UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Tesi di Laurea Magistrale in Ingegneria del Software II

Esperimenti per il confronto dell'efficacia di tecniche di testing Capture and Replay, Sistematiche ed Ibride nel contesto di App Android

Anno Accademico 2016/2017

Relatore

Ch.mo Prof. Porfirio Tramontana

Correlatore
Ing. Nicola Amatucci

Candidato
Daniele Ioviero
matr. M63000276

Alla mia famiglia
A Jessica Duarte
A chi è sempre onesto

Indice

Introduzione

Nell'ultimo decennio abbiamo assistito ad una crescita esponenziale della tecnologia in termini di funzioni, qualità, design, interoperabilità, interfacciamento e prestazioni. Oggi la tecnologia è semplice da usare e capire, ricca di servizi da offrirci ed in grado di renderci la vita molto più semplice. Basti pensare ai telefoni cellulare divenuti SmartPhone; essi ci ascoltano, ci parlano, ci guidano, ci aiutano, possono effettuare alcune commissioni al posto nostro, ad esempio possono risparmiarci la lunghissima coda alla posta od in banca. Tutto questo sia grazie alla potenza elaborativa degli attuali calcolatori sia ai linguaggi di programmazione ed i tools, a loro annessi, sempre più avanzati e più vicini al linguaggio umano, dando possibilità anche a meno esperti, ma con geniali idee, di poter creare un piccolo programma in grado di facilitargli qualche compito. Anche una persona affetta da paralisi, se in grado di muovere la testa, attraverso l'uso di uno speciale software ed una webcam, può muovere il cursore del mouse su una tastiera virtuale e scrivere un programma; un cieco può "digitare" delle lettere su un dispositivo attraverso l'uso della voce. Restando in tema SmartPhone, oggi si parla sempre più di applicazioni che di software, più comunemente chiamate App. Un'App può rappresentare un gioco, una calcolatrice, un foglio di disegno, una sveglia, la nostra agenda personale, insomma c'è un'App per tutto. Attraverso i vari store (magazzini virtuali), accessibili sia da SmartPhone che da PC, è possibile scaricare un'infinità di

applicazioni, se consideriamo il punto di vista del cliente. Uno sviluppatore di App, sfrutta lo store come vetrina per vendere o regalare il suo prodotto. Uno store, però, non dev'essere visto solo come un semplice magazzino virtuale in quanto permette di inserire commenti e dare un voto alle App, ragion per cui, un utente, può sapere a priori se ha trovato ciò che cercava o, se lo ha trovato, qual è il grado di soddisfazione da aspettarsi. Lato sviluppatore, il rating ed i commenti vengono utilizzati come spunto per apportare migliorie all'App, rilasciando poi gli aggiornamenti necessari attraverso lo store stesso. Cosa molto importante è l'attività svolta dai gestori degli store, i quali, generalmente, controllano le App prima di pubblicarle, in modo tale da ridurre la pubblicazione di App "malintenzionate". Pubblicare un'applicazione significa, quindi, rispettare quelle regole stipulate dai gestori degli store, verificate in primis dagli sviluppatori attraverso attività di Testing. Il progetto di sviluppo di questa Tesi riguarda il confronto tra le diverse tecniche di Testing atte a verificare la copertura del codice relativo alle App. In poche parole, facendo uso di alcune App preselezionate, verranno illustrate quelle tecniche di Testing che ci diranno, a fine confronto, quale di esse risulta essere più efficace attraverso operazioni di analisi statistica.

Nota bene, il processo di sviluppo di questa Tesi si preoccupa di ricercare, utilizzando delle App, le differenze, in termini di copertura ed efficacia, delle tecniche di testing senza scendere nel dettaglio delle App.

Di seguito verranno date informazioni relative al sistema operativo utilizzato, alle nozioni di base del Testing, alle diverse tecniche di Testing utilizzate, i tools di supporto al Testing utilizzati, agli obiettivi prefissati, i confronti tra le varie tecniche ed i risultati ottenuti.

I confronti tra le varie tecniche, sono necessari per comparare l'efficacia delle stesse e per poter validare o invalidare le ipotesi fatte nei confronti delle applicazioni coinvolte.

Capitolo 1: La Piattaforma Android

"Le migliori applicazioni per Android non sono ancora pronte, questo perché sarete voi, insieme ad altri sviluppatori come voi, a crearle." (Sergey Brin, Google Inc.). La fine del millennio scorso è stata sicuramente caratterizzata da Internet che ha rappresentato una vera e propria rivoluzione tecnologica e culturale. Infatti il fatto di dare la possibilità a chiunque di poter pubblicare informazioni accessibili da una qualunque parte del mondo ha permesso un notevole miglioramento nella divulgazione di informazioni, nella condivisione e nella distribuzione di dati di varia natura. Lo sviluppo di Internet ha portato ad una rivoluzione dal punto di vista informatico poiché ha permesso di passare dai programmi desktop in esecuzione su diversi PC ad applicazioni web accessibili da tutto il mondo grazie ai thin-client. Ed ora ci troviamo in una nuova rivoluzione: quella dei dispositivi mobili. Le informazioni che prima erano accessibili da un qualunque PC ora sono raggiungibili tramite dispositivi mobili che, con il trascorrere del tempo, stanno aumentando potenza, dimensioni e funzionalità. In questo contesto i principali costruttori di cellulari hanno messo a disposizione degli sviluppatori i propri sistemi operativi, ciascuno con il proprio ambiente di sviluppo, i propri tools e il proprio linguaggio di programmazione. Nessuno di essi si è affermato però come standard e quindi, data la varietà di ambienti tecnologici e tecnologie differenti, va sottolineata la difficoltà nella realizzazione di applicazioni per dispositivi mobili. A seconda del tipo di sistema operativo lo sviluppatore dovrà acquisire la conoscenza di un ambiente, di

una piattaforma e di un linguaggio di programmazione. È necessaria dunque una standardizzazione verso la quale si sono diretti Google e l'OHA (Open Handset Alliance) con la creazione di Android. In particolare la piattaforma Android non riguarda un semplice linguaggio di programmazione, ma un vero e proprio insieme di strumenti e librerie volte alla realizzazione di applicazioni mobili. Ha come obiettivo quello di fornire agli sviluppatori ciò di cui hanno bisogno per realizzare le proprie applicazioni che poi altri utenti potranno scaricare; non stiamo parlando di piccole cifre, secondo un report redatto da Sensor Tower, ci sono stati **11,1 miliardi di download di App dal Google Play Store** nel primo trimestre del 2016, con un forte aumento rispetto allo stesso periodo dell'anno precedente; il tutto a fronte di circa 2 miliardi di contenuti disponibili, tra applicazioni e giochi. A differenza degli altri ambienti di sviluppo, Android ha la fondamentale caratteristica di essere Open Source. C'è da dire che Android non nasce grazie a Google, come in molti credono, ma nasce nel 2003 da una startup californiana, Android Inc., acquisita nel 2005 da Google. L'evoluzione della piattaforma è tuttavia curata dall' Open Handset Alliance (http://www.openhandsetalliance.com), nato nel 2007. Del gruppo, oltre a Google, fanno parte numerose società, tra cui Intel, Motorola, Samsung, Sony-Ericsson, Texas Instruments, LG e molti altri. La licenza scelta dalla Open Handset Alliance è la Open Source Apache License 2.0, che permette ai diversi vendor di costruire su Android le proprie estensioni anche proprietarie senza legami che ne potrebbero limitare l'utilizzo.

1.1 Descrizione della piattaforma

Android non è un linguaggio di programmazione, ma un vero e proprio insieme di strumenti e librerie per la realizzazione di applicazioni mobili. Un aspetto fondamentale del successo di Android è che il linguaggio da esso utilizzato non è nuovo, altrimenti gli sviluppatori sarebbero stati obbligati a imparare da zero, stiamo parlando del bel noto Java descritto dalle specifiche di Sun Microsystems utilizzate dal 1995.

1.1.1 La Delvik Virtual Machine (DVM) e Android Run Time (ART)

La decisione più importante è stata quella di adottare una nuova Virtual Machine (VM), diversa da quella Sun, ottimizzata per l'esecuzione di applicazioni in ambienti a memoria ridotta come nel nostro caso, sfruttando al massimo le caratteristiche del sistema operativo ospitante. La scelta è caduta sulla Dalvik Virtual Machine (DVM) in grado di eseguire codice contenuto all'interno di file con estensione .dex ottenuti a partire dal byte-code Java. La scelta di non utilizzare nemmeno il byte-code Java nasce dall'esigenza di risparmiare quanto più spazio possibile per la memorizzazione ed esecuzione delle applicazioni. Per comprendere di quanto lo spazio richiesto diminuisca facciamo un esempio: se un'applicazione Java è descritta da codice contenuto all'interno di un archivio .jar di 100 KB, la stessa potrà esser contenuta all'interno di un file di dimensioni di circa 50 KB se trasformato in .dex. Questa diminuzione del 50% avviene nella fase di trasformazione dal byte-code Java al byte-code per la DVM. La DVM non elimina il Garbage Collector (GC) che troviamo per un'applicazione Java, ovvero quella modalità automatica di gestione della memoria, mediante la quale un sistema operativo, o un compilatore e un modulo di run-time, liberano le porzioni di memoria che non dovranno più essere successivamente utilizzate dalle applicazioni, poiché, togliendo questo processo e quindi porre la gestione della memoria a carico del programmatore avrebbe complicato lo sviluppo delle applicazioni e inoltre avrebbe anche aumentato la probabilità di bug e memory-leak (consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa). Vediamo infatti che la DVM, già dalla versione 2.2, implementa il Just In Time (JIT) compiler, un meccanismo attraverso il quale la Java Virtual Machine (JVM)[1] riconosce determinati pattern di codice Java, traducendoli in altrettanti frammenti di codice nativo per un'esecuzione più efficiente. La DVM migliora il meccanismo di generatore del codice, qui è detto register based (orientato all'utilizzo dei registri),

[1] La macchina virtuale Java, detta anche Java Virtual Machine (JVM), è il componente della piattaforma Java che esegue i programmi tradotti in byte-code dopo una prima compilazione. Il processo può essere schematizzato con *Codice Java : compilazione : byte-code : VM → esecuzione reale del programma*.

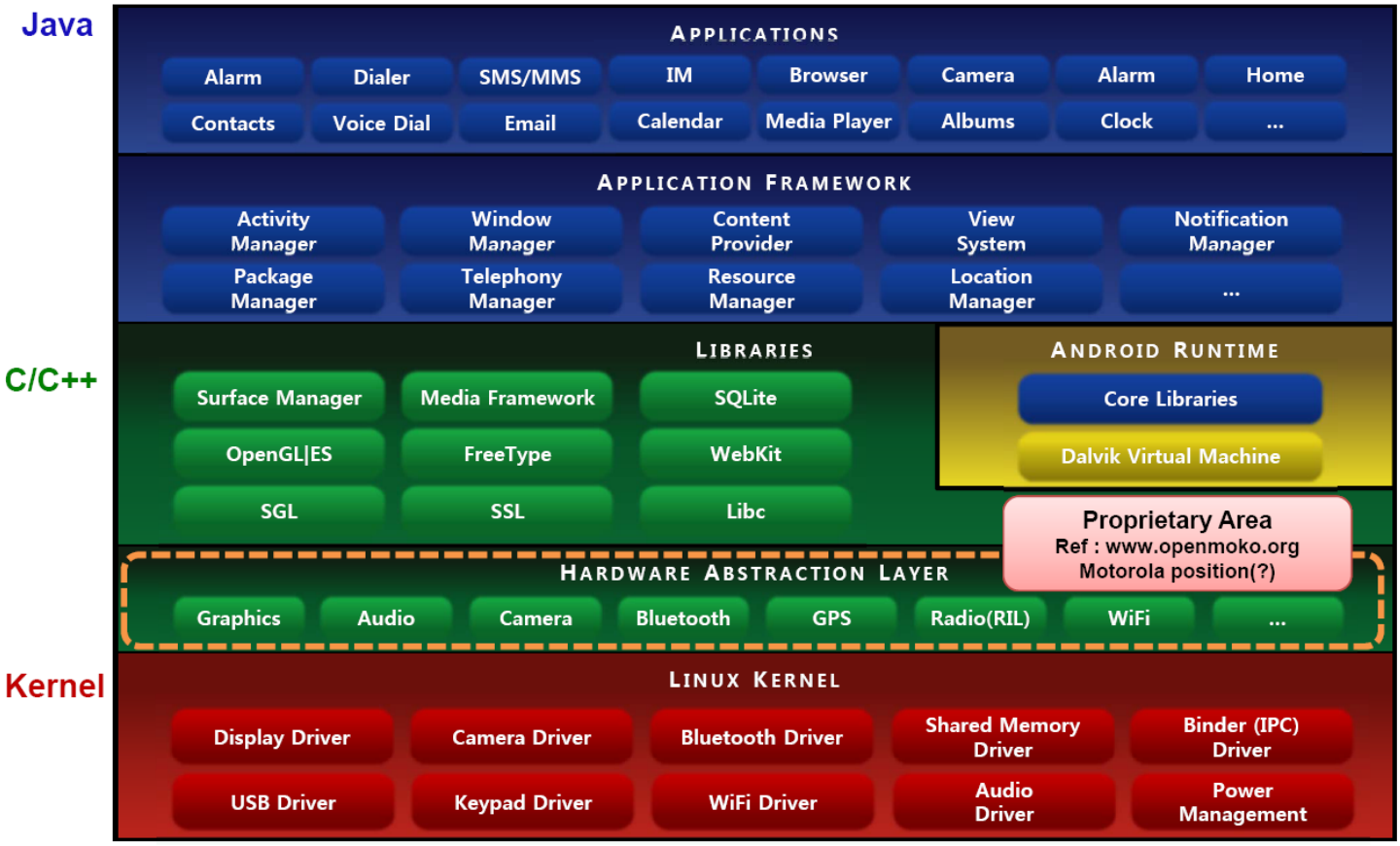
invece quello della JVM è detto stack based (orientato all'utilizzo di stack). In questo modo ci si aspetta che, a parità di codice Java, le operazioni che necessitano di essere eseguite si riducano del 30%. Notiamo infatti che se ad esempio volessimo eseguire un'espressione di somma tra due operandi, per quanto riguarda il meccanismo stack based, prima di eseguire l'operazione e inserire nello stack il risultato, dovremmo prima caricare gli addendi. Invece per quanto riguarda il meccanismo register based dovremmo solo caricare gli addendi in zone diverse di un registro e poi memorizzare il risultato nel registro stesso. Vediamo dunque che si ottiene un minor tempo di esecuzione delle istruzioni, al prezzo di una maggiore elaborazione in fase di compilazione. La compilazione JIT permette di ottenere un buon compromesso tra velocità d'esecuzione e portabilità del codice. Nella fase di compilazione del byte-code è eseguita la maggior parte del "lavoro pesante", ovvero tutte quelle operazioni che richiedono molto tempo per essere eseguite, come l'analisi sintattica e semantica del codice sorgente e una prima fase di ottimizzazione; la compilazione da byte-code a codice nativo è invece molto più veloce. I compilatori da byte-code a codice macchina sono più semplici da scrivere perché la maggior parte del lavoro è già stata compiuta dal compilatore che ha prodotto il byte-code; questo, inoltre, rende i programmi in byte-code più facilmente portabili su nuove architetture. Dobbiamo pensare poi che stiamo trattando coi dispositivi mobili e quindi alleggerire il carico in fase di esecuzione è sicuramente un aspetto positivo. Inoltre la DVM ottimizza l'esecuzione di più processi in contemporanea e, anche questo, è un aspetto fondamentale nei dispositivi mobili.

1.2 **Architettura**

Possiamo quindi dire che Android è un’architettura che comprende l’insieme degli strumenti utili alla creazione delle applicazioni per dispositivi mobili, tra cui un sistema operativo, un insieme di librerie native per le funzionalità principali della piattaforma, un’implementazione della VM e un insieme di librerie Java. Si tratta di

un'architettura a livelli dove gli strati inferiori offrono servizi a quelli superiori offrendo un più alto grado di astrazione (Fig. 1.1).

Facciamo una panoramica di ciò che compone tale architettura:



(Figura1.1)

1.2.1 **Il livello Applications**

Il livello più alto dello stack è costituito dalle applicazioni: è interessante osservare che non vengono considerate soltanto le App native (come per esempio il sistema di gestione dei contatti, l'applicazione per l'invio di SMS, il calendario), ma anche quelle provenienti da altre fonti, come ad esempio quelle scaricate dallo store, garantendo alle App gli stessi privilegi (o quasi).

1.2.2 **Il livello Application Framework**

Tutte le librerie nei livelli sottostanti vengono poi utilizzate da un insieme di componenti di più alto livello che costituiscono l'Application Framework.

Quest'ultimo mette a disposizione degli sviluppatori la possibilità di utilizzare in modo immediato componenti riutilizzabili per lo sviluppo delle proprie applicazioni. Tali componenti, inoltre, sono facilmente sostituibili con implementazioni personalizzate. Sono presenti i seguenti moduli:

- ***Activity Manager*:** modulo che gestisce tutto il ciclo di vita delle Activity.
  Le Activity sono entità associate ad una schermata, rappresentano quindi l'interfaccia verso l'utente. Il suo compito è quello di mostrare le varie Activity sul display del terminale e di organizzarle in uno stack in base all'ordine di visualizzazione sullo schermo;
- ***Package Manager*:** modulo che gestisce i processi di installazione e rimozione delle applicazioni dal sistema;
- ***Telephony Manager*:** modulo che gestisce l'interazione con le funzioni tipiche di un cellulare;
- ***Content Provider*:** modulo che gestisce la condivisione di informazioni tra i vari processi attivi. Il suo utilizzo è simile a quello di un repository comune nel quale i vari processi possono leggere e scrivere informazioni;
- ***Resource Manager*:** modulo deputato alla gestione delle informazioni relative ad una applicazione (file di configurazione, file di definizione dei layout, immagini utilizzate, …);
- ***View System*:** gestisce l'insieme delle viste utilizzate nella costruzione dell'interfaccia verso l'utente (bottoni, griglie, text boxes, …);
- ***Location Manager*:** modulo che mette a disposizione dello sviluppatore una serie di API che si occupano della localizzazione. Esistono due provider per la localizzazione: GPS e NETWORK. GPS utilizza i satelliti geostazionari per il posizionamento geografico. NETWORK utilizza punti dei quali si conosce la posizione geografica, come ad esempio celle GSM oppure reti wireless geo-localizzate;

- ***Notification Manager*:** mette a disposizione una serie di meccanismi utilizzabili dalle applicazioni per notificare eventi al dispositivo che intraprenderà delle particolari azioni in conseguenza della notifica ricevuta.

1.2.3 **Android Runtime**

L'Android Runtime si compone delle Core Libraries e della Dalvik Virtual Machine, quest'ultima già trattata in precedenza. Le Core Libraries forniscono molte delle funzionalità delle analoghe librerie disponibili per il linguaggio di programmazione Java. Per le applicazioni, in fase di compilazione, si avrà bisogno del file "android.jar", contenente le classi di sistema relative alla versione Android su cui si basa l'applicazione, per la creazione del bytecode Java, mentre in esecuzione il device metterà a disposizione la versione "dex" del runtime che costituisce appunto la core library. Non dimentichiamo che la DVM è ottimizzata per macchine con risorse ridotte e quindi risulta ad-hoc per i dispositivi mobili. Una caratteristica molto importante della DVM è quella di consentire un'efficace esecuzione di più processi contemporaneamente; infatti ciascuna applicazione è in esecuzione nel proprio processo Linux e questo comporta dei vantaggi dal punto di vista delle performance.

1.2.4 **Libraries**

È interessante sapere che le librerie native non sono scritte in Java. Le applicazioni Android si sviluppano (principalmente) in Java, ma la maggior parte del codice che verrà eseguito sarà codice nativo in C/C++. Questo per il semplice fatto che molti dei componenti della piattaforma sono semplicemente delle interfacce Java di componenti nativi. Il loro scopo principale è supportare l'Application Framework. Alcune di queste sono create appositamente per Android, altre sono prelevate dalla community open source per completare il sistema operativo.

Tra le varie librerie native troviamo:

- ***Surface Manager (SM)*:** di notevole importanza poiché ha il compito di gestire le

componenti dell'interfaccia grafica, controllando e gestendo le diverse finestre visibili a schermo. Ad esempio impedendo la sovrapposizione disordinata in caso di più finestre aperte contemporaneamente;

- ***OpenGL ES e Scalable Graphics Library (SGL)*:** per gestire grafica 2D e 3D all'interno di una stessa applicazione;
- ***Media Framework*:** contenente le librerie necessarie per gestire molti formati audio e video comuni quali MPEG4, H.264, MP3, AAC, JPG, e PNG in continuo aggiornamento in base alle novità presenti;
- ***FreeType*:** per la gestione dei font;
- ***SQLite*:** il DBMS (Database Management System) utilizzato da Android; sistema relazionale progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database, molto compatto.
- ***WebKit*:** un browser-engine, open source, che può essere integrato in qualunque applicazione sotto forma di finestra browser;
- ***SSL*:** librerie per gestire i Secure Socket Layer e quindi tutti i problemi legati alla sicurezza;
- ***Libc*:** implementazione della libreria standard C, ottimizzata per i dispositivi basati su versioni Linux embedded;

1.2.5 **Livello Hardware Abstraction**

È un livello, interposto fra il livello *Libraries* ed il livello *Kernel.* L'accesso ai driver dei dispositivi normalmente non è molto standardizzato quindi Android effettua un'astrazione di ciascuno di essi con una libreria nativa condivisa. La libreria è un oggetto condiviso aderente a un'interfaccia comune che supporta ogni principale driver di hardware. Questo comporta che ogni produttore deve implementare una libreria comune e astrarre le complicazioni relative al design del suo specifico dispositivo.

1.2.6 **Linux Kernel**

Il livello più basso è rappresentato dal kernel Linux. Esso garantisce gli strumenti di basso livello per l'astrazione dell'hardware sottostante attraverso la definizione di diversi driver ad esempio per la gestione del display, della connessione Wi-Fi, del Bluetooth. E' da notare anche la presenza di un driver dedicato alla gestione della comunicazione tra processi diversi (IPC); la sua importanza è fondamentale per far comunicare componenti diversi in un ambiente in cui ciascuna applicazione viene eseguita all'interno di un proprio processo. Fornisce, inoltre, gestione della memoria, impostazioni di sicurezza, gestione del consumo.

1.3 **Le applicazioni (App) del mondo Android**

Come già accennato in precedenza, le App Android sono scritte per lo più in Java; questo perché esiste la possibilità di scriverle in C/C++. E' importante sapere che Android mette a disposizione di tutti il Software Development Kit (SDK), il quale fornisce tutti gli strumenti necessari allo sviluppo delle App. Ora non resta che conoscere quali sono le componenti principali che permettono ad un App di interagire con l'utente e, allo stesso tempo, comunicare con l'ambiente che la ospita:

- **Activity:** Le attività sono ciò che l'utente vede a schermo e, quindi, ciò che consentono all'utente di interagire con l'App attraverso i dispositivi di input messi a disposizione. Comunemente fanno uso di componenti grafici già pronti, come quelli presenti nel pacchetto android.widget. Un'applicazione può avere più activity, il cui ciclo di vita è gestito dall'Activity Manager, servizio presente nell'Application Framework, come visto in precedenza. L'Activity Manager è responsabile della creazione, distruzione e gestione delle activity.
- **Services:** I servizi, possono essere capiti in maniera molto semplice considerando quelle "attività", quei task, eseguiti in modo non visibile all'utente, come, ad esempio, quei micro-task utili ad eseguire un task più grande. Possono essere considerati come "attività" in background. Una delle differenze sostanziali, rispetto

alle Activity, è che il ciclo di vita dei servizi è nettamente differente in quanto possono solo essere avviati od interrotti.

- **Content Provider:** Sono interfacce per la condivisione di dati tra le applicazioni e costituiscono pertanto un canale di comunicazione tra le differenti applicazioni installate nel sistema. Meccanismo necessario affinché ciascuna applicazione sia eseguita in una propria sandbox, isolata dalle altre per motivi di sicurezza.
- **Broadcast Receiver:** Rappresentano l'implementazione di un meccanismo publish/subscribe a livello di sistema. Il receiver è un codice che viene attivato solo in concomitanza di un evento, al quale il receiver stesso è iscritto. Lo stesso sistema esegue il broadcast di eventi continuamente. Un esempio di evento può essere la ricezione di un messaggio o di una chiamata, ciò comporta l'attivazione di diversi receiver, scatenando diverse azioni, avviando, ad esempio, un'activity o un service.
- **Intent e Intent Filter:** attraverso un intent (tradotto come "intenzione"), un'applicazione può dichiarare la volontà di compiere una particolare azione senza pensare a come questa verrà effettivamente eseguita. Ciascuna activity può dichiarare l'insieme degli intent che è in grado di esaudire attraverso quelli che si chiamano *intent filter*. Se un'Activity ha tra i propri intent filter, ad esempio, quello relativo alla scelta di un contatto dalla rubrica, quando tale intent viene richiesto, essa verrà visualizzata per permettere all'utente di effettuare l'operazione voluta. Tale attività sarà la stessa per ogni applicazione senza che occorra definirne una propria. Questo utilizzo degl'intent riguarda la comunicazione tra più Activity. Possiamo distinguere tra Intent esplicito ed implicito, il primo tipo permette il passaggio da un'Activity ad un'altra specificando esplicitamente la classe Java dell'Activity che si vuole mandare in esecuzione, con il secondo tipo, invece, non viene specificata l'Activity da mandare in esecuzione, ma vengono forniti i dati da elaborare e/o una descrizione dell'azione da svolgere: sarà il sistema operativo ad inoltrare la richiesta.

1.3.1 **I widget**

Abbiamo parlato delle componenti principali che compongono un'App, ma come interagisce effettivamente l'utente con un'activity? I widget sono quei componenti di base per l'interazione con l'utente, come i bottoni, le check-box, le liste, i campi di testo e così via. I widget predefiniti di Android estendono tutti (direttamente o indirettamente) la classe *View* (classe utile alla realizzazione dell'interfaccia utente differente dalla classe *ViewGroup* utile come base per la sottoclasse chiamata *layout*, la quale offre appunto differenti tipi di layout), hanno responsabilità ben precise e fanno parte del package android.widget. Alcuni di questi sono:

- **TextView:** permette di mostrare del testo all'utente;
- **EditText:** permette all'utente di modificare il testo mostrato;
- **Button:** realizza un bottone che l'utente può cliccare;
- **ImageView:** componente che permette di mostrare un'immagine;
- **ImageButton:** componente che realizza un bottone a partire da un'immagine;
- **CheckBox:** questo componente realizza una check-box.

Chiarito il concetto dei widget, c'è bisogno di capire come questi ultimi attivano porzioni di codice per soddisfare una richiesta utente. La gestione avviene attraverso particolari listener o con override di metodi che il sistema richiama automaticamente. Per poter gestire gli input generati dall'utente è necessario definire un evento di ascolto e registrarlo alla View. Quest'ultima contiene una collezione di interfacce chiamate *On<something>Listener*, ognuna delle quali con un metodo di callback denominato appunto *On<something>()*. Ci sono ad esempio le seguenti interfacce: *View.OnClickListener* per la gestione dei "click" sulla View, *View.OnTouchListener* per la gestione degli eventi di touch screen e *View.OnKeyListener* per gestire la pressione dei tasti del device. Così se vogliamo che la View ci notifichi quando si verifica un evento di "click", ad esempio la pressione di un bottone, basta implementare l'interfaccia OnClickListener ed il metodo di callback *onClick ()* e registrare lo stesso

alla View utilizzando *setOnClickListener ( )*; o ancora effettuare l'override di un metodo di callback già esistente.

1.4 Il ciclo di vita delle Activity

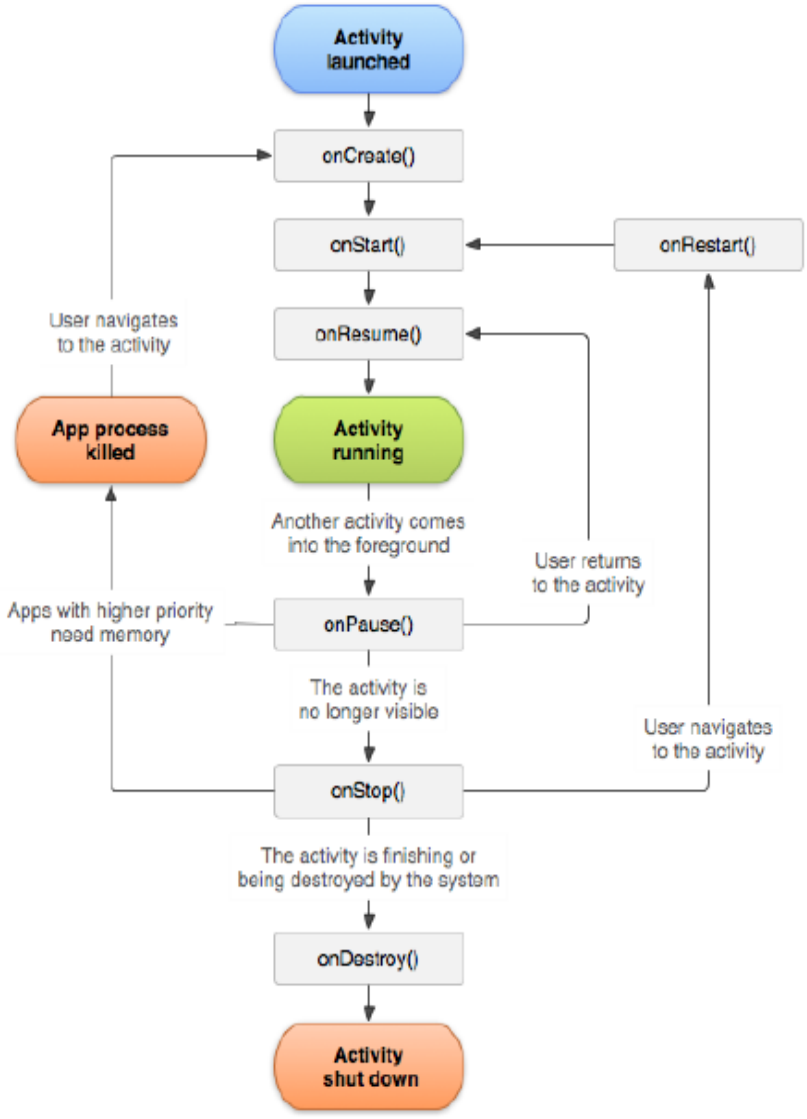
L'importanza di capire qual è il ciclo di vita delle activity, nasce da ciò che è stato detto in precedenza riguardo le activity stesse, i servizi e lo scambio di informazioni tra activity. Consideriamo i 4 stati principali:

- **ACTIVE:** è lo stato più vicino all'utente, riguarda lo stato in cui un'activity risulta attiva sullo schermo (in esecuzione).
- **PAUSED:** si riferisce al caso di Activity non attive ma ancora visibili per la trasparenza di quelle superiori o perché queste non occupano tutto lo spazio a disposizione. In questo stato vengono mantenute tutte le informazioni di stato ma può essere eliminata dal sistema in situazioni estreme basse memoria.
- **STOPPED:** si riferisce al caso di Activity non attive né visibili e non più sensibili agli eventi dell'utente è sarà tra le prime candidate ad essere eliminata per necessità di memoria.
- **INACTIVE:** l'Activity si trova in questo stato quando viene eliminata oppure prima di essere creata.

Per passare da uno stato all'altro vengono utilizzati i seguenti metodi

- **onCreate():** si tratta dell'operazione invocata in corrispondenza della creazione dell'Activity e che, nella maggior parte dei casi, contiene le principali operazioni di inizializzazione.
- **onRestart():**Chiamato dopo che l'Activity è stata arrestata, prima che venga avviato di nuovo. Sempre seguito da onStart().
- **onStart():** Chiamato quando l'Activity diventa visibile all'utente. Seguito da onResume() se l'Activity va in primo piano, o da OnStop() se diventa nascosta.

- **onResume():** Chiamato non appena l'utente inizia l'interazione con l'Activity Sempre seguito da onPause ().
- **onPause():** Chiamato quando sopraggiunge un altro evento come la chiamata ad un'altra Activity.
- **onStop():** Chiamato se l'activity non è più visibile all'utente. Provvede, in mancanza di memoria, ad eliminare le Activity in sospeso non necessarie.
- **onDestroy():** Se l'Activity viene terminata (dall'utente o per mancanza di memoria) viene chiamato il metodo che chiude le Activity.



(Figura 1.2: Il ciclo di vita delle Activity)

Capitolo 2: Testing

In questo capitolo verranno date le informazioni di base per capire al meglio di cosa si parla quando ci si occupa di Testing, verranno spiegate alcune delle tecniche esistenti e verranno mostrate in maniera esaustiva le tecniche di Testing utilizzate sulla piattaforma Android. Vedremo, inoltre, i vari frameworks e tools di sostegno impiegati.

2.1 Testing

Gli standard IEEE definiscono il Testing come quel processo utile, una volta osservati o registrati gli esiti, per scopi di valutazione di prefissate caratteristiche, relative ad un sistema o un componente sotto osservazione in specifiche condizioni iniziali (*Standard IEEE 610.12-1990*). Nel nostro caso, andremo a testare non un sistema né un componente, intesi come qualcosa di fisico, ma andremo a testare del software, le famose App, per cui con lo *Standard IEEE 729-1983* sappiamo che il Testing può essere considerato come quel processo che analizza tutte, o quasi, le componenti di un software in modo tale da ottenere informazioni relative a delle condizioni specifiche con il fine di confrontarle con quelle pre-stabilite ed effettuare, di conseguenza, le valutazioni necessarie per poter dichiarare esito positivo o negativo sulla base dei risultati ottenuti. Come già detto in precedenza, un'App deve rispettare norme ben precise per poter essere pubblicata sugli store, il Testing entra in gioco in quanto facilita il compito di *"Verifica"* delle specifiche, questo perché,

alla base, ci sono 2 domande che il Testing propone, le quali, se sfruttate come linee guida, permettono di ottenere un buon prodotto ed un ottimo feedback:

1. Stiamo creando il giusto prodotto? (Attività di Verifica)
2. Lo stiamo facendo nel modo giusto? (Attività di Validazione)

L'importanza che si verifichino entrambe le attività sta nel fatto che la (1) punta alla conformità delle specifiche del software, ossia che deve soddisfare i requisiti funzionali e non funzionali specificati, la (2) mira a capire se il software sotto test è ciò che il cliente realmente vuole e che quindi soddisfa le aspettative del cliente stesso. Nel processo di verifica e validazione, due sono gli approcci complementari al controllo e all'analisi del sistema:

- **Ispezione del software**: tecnica statica dove non è necessaria l'esecuzione, in cui sono analizzati il documento dei requisiti, i diagrammi di progettazione e ispezionato il codice sorgente del programma. E' possibile l'uso di analizzatori statici automatici.
- **Test del software**: tecnica dinamica in quanto richiede l'esecuzione del programma in esame attraverso i casi di test sottomessi. Si esaminano gli output e il suo comportamento operativo.

2.2 Test

"Il testing non può dimostrare l'assenza di difetti, ma solo la loro presenza" (Dijkstra). Per quanto detto in precedenza, sappiamo che, dinamicamente, attraverso il testing è possibile dimostrare l'affidabilità e le prestazioni di un software, abbiamo quindi bisogno di una esecuzione reale, con dati simili a quelli reali, in modo tale da poter visionare il corretto funzionamento. C'è da dire però che se il programma viene eseguito, non vuol dire che sia stato scritto in modo corretto,

stiamo parlando dei così detti *"bug"*, termine ormai diffuso grazie alla rete, ovvero della manifestazione di *"errori"*, per lo più umani, attraverso comportamenti del software non attesi, tali bug portano il sistema in uno stato di "failure" anch'esso, quindi, di natura dinamica, ottenibile solo mediante esecuzione.

Il testing, attraverso attività di debugging, permette di semplificare il processo di localizzazione e correzione dei difetti.

Caratteristiche di qualità:

- **Un test deve aiutare a localizzare i difetti:** tanti test semplici consentono una più semplice localizzazione rispetto a pochi test complessi
- **Un test dovrebbe essere ripetibile:** Potrebbe non essere possibile se l'esecuzione del caso di test influenza l'ambiente di esecuzione senza la possibilità di ripristinarlo. Inoltre potrebbe non essere possibile se nel software ci sono degli elementi indeterministici (dipendenti da input non controllabili)
- **Un test dovrebbe essere preciso:** La valutazione del risultato deve essere univoca
- **Affidabilità e Validazione:** scelto un criterio di selezione di test C, questo risulta *affidabile* per un programma P se, per ogni coppia di test-suite (T1;T2), selezionati da C, T1 ha successo, quindi anche T2 e viceversa. Diciamo, invece, che C è *valido* per P se, esiste almeno una T selezionata da C che ha successo per P. Se un criterio C è *affidabile* e *valido* allora qualsiasi test-suite generata da C ha successo.
- **Efficacia ed Efficienza:** Descritti dettagliatamente nel capitolo 5

La caratteristica di ripetibilità è importante per i test di regressione: in seguito all'attività di debugging è importante poter rieseguire i test precedenti per verificare che le modifiche non abbiano introdotto nuovi errori.

Per quanto riguarda i problemi e le limitazioni:

- **La correttezza di un programma è un problema indecidibile:** risulta impossibile riprodurre sempre tutte le possibili configurazioni di valori di input in corrispondenza di tutti i possibili stati interni di un sistema software
- **Garanzia:** non è possibile sapere che se alla n-esima prova un modulo od un sistema abbia risposto correttamente (ovvero non sono stati più riscontrati difetti), altrettanto possa fare alla (n+1)-esima
- **Valori di input:** Nel campo del software si ha a che fare con sistemi discreti, per i quali piccole variazioni nei valori d'ingresso possono portare a risultati scorretti.

2.3 Principali famiglie di tecniche di Testing

Diversi sono i criteri di selezione dell'attività di testing da utilizzare, tutti riconducibili a due macro categorie che rappresentano, quindi, non due tecniche di testing ma due famiglie di tecniche utilizzabili:

- **Testing Funzionale:** Richiede l'analisi degli output generati dal sistema (o da suoi componenti) in risposta ad input definiti sulla base della sola conoscenza dei requisiti del sistema (o di suoi componenti). Questo tipo di testing viene spesso realizzato in modalità **Black-Box**, ovvero senza accedere in alcun modo alla struttura interna del software. Vien da se che, facendo testing black-box, il software è acceduto unicamente attraverso la sua interfaccia. C'è da precisare, però, che non bisogna semplicemente considerare questa modalità come: *"metto l'input, valuto l'output",* in quanto esistono diverse tecniche, per citarne qualcuna: Testing basato sui requisiti; basato sugli scenari dei casi d'uso; con classi di equivalenza; basato sui valori limite; a partire dalle tabelle di decisione.
- **Testing Strutturale:** Fondato sulla conoscenza della struttura del software, ed in particolare del codice, degli input associati e dell'oracolo (ciò che conosce i

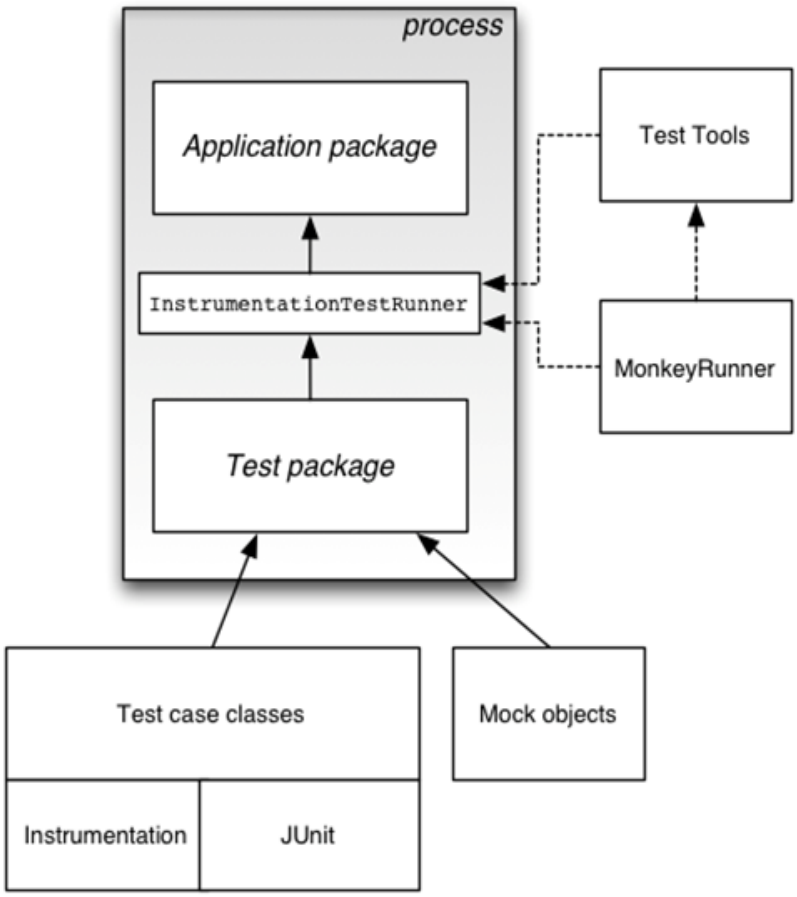
risultati attesi, in genere è umano), per la definizione dei casi di prova. Questo modo di fare testing richiede l'accesso al codice in quanto c'è la necessità di conoscere quali input o quali condizionamenti particolari attivano determinati porzioni di codice, se presenti. Questa modalità è chiamata **White-Box** e risulta molto utile al debugger in quanto fornisce informazioni maggiori sulla posizione degli errori. A differenza della prima modalità, c'è bisogno di stabilire un criterio di copertura in quanto il tester può essere interessato a coprire, ad esempio, semplici righe di codice, intere porzioni, determinate condizioni e decisioni.

Ai fini della stesura di questa tesi è stato necessario svolgere attività di testing sfruttando ambedue le modalità sopracitate, questo perché, come dalla premessa iniziale, andremo ad effettuare i dovuti confronti tra le diverse tecniche di testing, sfruttando diverse metriche (es. la quantità di codice eseguito rispetto a quello eseguibile).

2.4 Android Testing

Il sistema di sviluppo Android mette a disposizione un Framework, basato su JUnit, necessario ai fini del testing in quanto fornisce gli strumenti necessari per testare tutti gli aspetti di un'App. Il framework viene esteso da classi specifiche per i vari tipi di componenti dell'application framework in modo tale da permettere di svolgere diverse operazioni come il controllo del ciclo di vita del componente.

Iniziamo con l'illustrazione del diagramma che rappresenta il framework:



(Figura 2.1: Modello concettuale del Framework)

Così come per JUnit classico, i metodi di test vengono organizzati in classi, test cases/suite, eseguite dall'*InstrumentationTestRunner*. La presenza di questo componente è importante in quanto permette di eseguire separatamente la AUT (Application Under Test=Applicazione Sotto Test) ed i test case in thread differenti, per non influenzare il sistema, anche se appartenenti allo stesso processo. A tal proposito diciamo che le operazioni di instrumentazione fanno da gancio verso la AUT, permettendo di gestirne il ciclo di vita, i vari stati e di richiamare tutti i metodi di callback necessarie per lavorare con le activity dell'AUT (es. onCreate(); onResume(); onPause()).

2.4.1 **Test Automation**

A causa del numero di casi di test necessari per un testing efficace, l'operazione di progettazione manuale dei casi di test può essere molto onerosa. Automatizzare i processi di generazione ed esecuzione parziale/totale dei casi di test significa ridurre i tempi ed i costi del testing. È un rafforzamento delle metodologie e dei processi utilizzati nel test manuale, anche se non li sostituisce del tutto, poiché alcune verifiche (ad esempio verifiche sul layout grafico e sull'usabilità del prodotto) restano più efficaci se eseguite manualmente. Automatizzare i casi di test, in modo da essere eseguiti senza l'intervento umano, ha un certo costo iniziale che viene ripagato nel tempo a seguito delle numerose ri-esecuzioni dei casi di test evitando, inoltre, procedure noiose, ripetitive e potenzialmente soggette a errori umani: i test case possono essere generati automaticamente come nelle tecniche per la generazione automatica di casi di test per il testing black-box partendo dall'analisi delle sessioni utente (*User Session*), ovvero delle sequenze dei valori di input immessi e di output ottenuti in utilizzi reali del software. In pratica, vengono utilizzati strumenti in grado di mantenere un log di tutte le interazioni utente-App da testare e l'applicazione stessa (fase di *Capture*); a partire da tali dati, vengono formalizzati casi di test che replicano le interazioni "catturate" (fase di *Replay*). In questo modo è possibile ottenere casi di test che siano rappresentativi dei reali utilizzi dell'applicazione da parte dei suoi utenti. Il risultato ottenuto (in termini di output e stato) durante la fase di Capture può rappresentare l'oracolo per un futuro Replay. La registrazione (*Recording*) dei vari "passi" compiuti dall'utente reale, è possibile per mezzo di alcuni strumenti che catturano gli eventi generati dall'interazione utente-activity. Per raggiungere gli scopi previsti da questo lavoro di tesi, si è fatto ampio uso di *Robotium Recorder*, software di supporto in grado di offrire i processi di capture e replay sopracitati e quindi di interagire direttamente con le App, presentate nel prosieguo, per generare casi di test basati su sessioni utente. Da non trascurare gli aspetti negativi inerenti il capture-replay, stiamo parlando di quelle situazioni in cui casi limite non vengono testati per errore umano

(scarsa efficacia), scarsa efficienza con test case ridondanti, tempi di capture troppo lunghi per esecuzioni significative.

2.4.2 **Testing e GUI**

Avere conoscenze sulla GUI (*Graphical User Interface*) è importante in quanto è proprio grazie ad essa che l'utente interagisce con un'App, di conseguenza creare test case per testare le App vuol dire simulare l'interazione utente-dispositivo. Il testing GUI-based prevede componenti in grado di rilevare e riconoscere gli elementi della GUI (es. bottoni), esercitare eventi su di essi, fornire degli input a questi elementi (ad esempio editando campi di testo), o ancora controllare le descrizioni della GUI per verificare se sono consistenti con quelle attese. Ciò rende il testing GUI-based particolarmente difficoltoso e la sua implementazione strettamente dipendente dalla tecnologia utilizzata. Gli eventi di cui parliamo sono quelli legati all'azione scaturita cliccando un bottone, selezionando un elemento, scorrendo l'activity attiva eccetera, l'evento scaturito, quindi, rappresenta l'intenzione dell'utente, ci sarà pertanto un gestore degli eventi che soddisferà, se possibile, quella intenzione. In genere il passaggio tra un'activity e l'altra avviene proprio grazie allo scaturirsi di questi eventi, ragion per cui, per rappresentare tutte le possibili interazioni eseguibili dall'utente, viene utilizzato il modello descrittivo chiamato FSM (Macchina a Stati Finiti), dove lo "stato" rappresenta l'insieme degli elementi costitutivi dell'interfaccia (ad esempio finestre, widget, layout) corrispondente a uno stato dell'automa. Gli eventi sull'interfaccia, come la selezione di un elemento in una lista, sono modellati come ingressi che innescano transizioni nell'automa. La sequenza di input corrisponde a una sequenza di stati visitati. Ottenere la FSM che descrive l'interfaccia su cui effettuare i test case spesso non è semplice. Essa può essere prodotta in fase di sviluppo dell'applicazione oppure ricostruita tramite Reverse Engineering a partire dall'interfaccia stessa. Come già accennato in precedenza, Robotium Recorder effettua attività di capture e replay; in sostanza nelle attività di capture va a registrare proprio questi eventi sulle

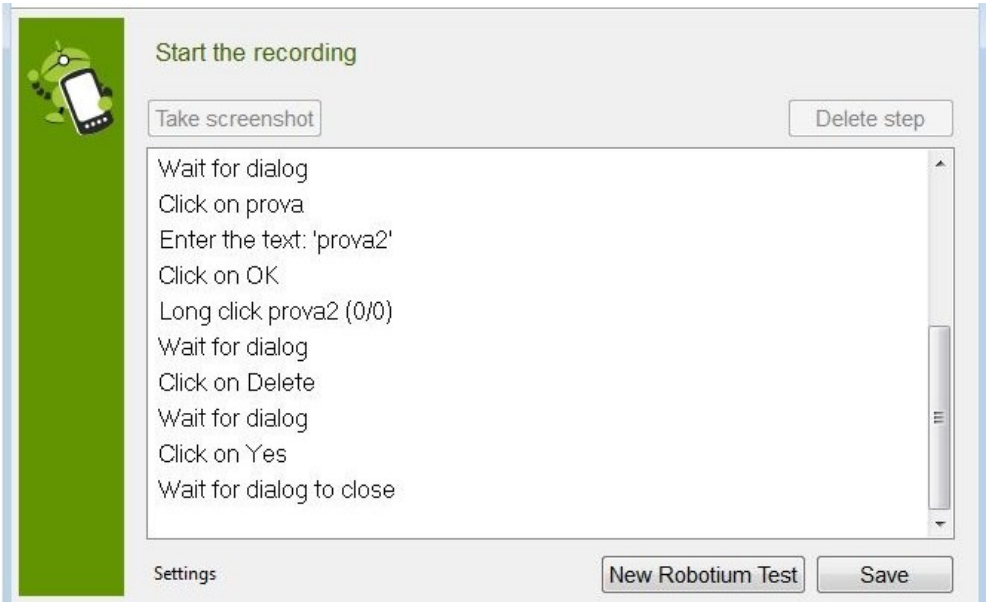
componenti della GUI, riproponendole, ad ogni attività di replay di quel test case, modificabile all'occorrenza.

2.4.3 **Robotium Recorder**

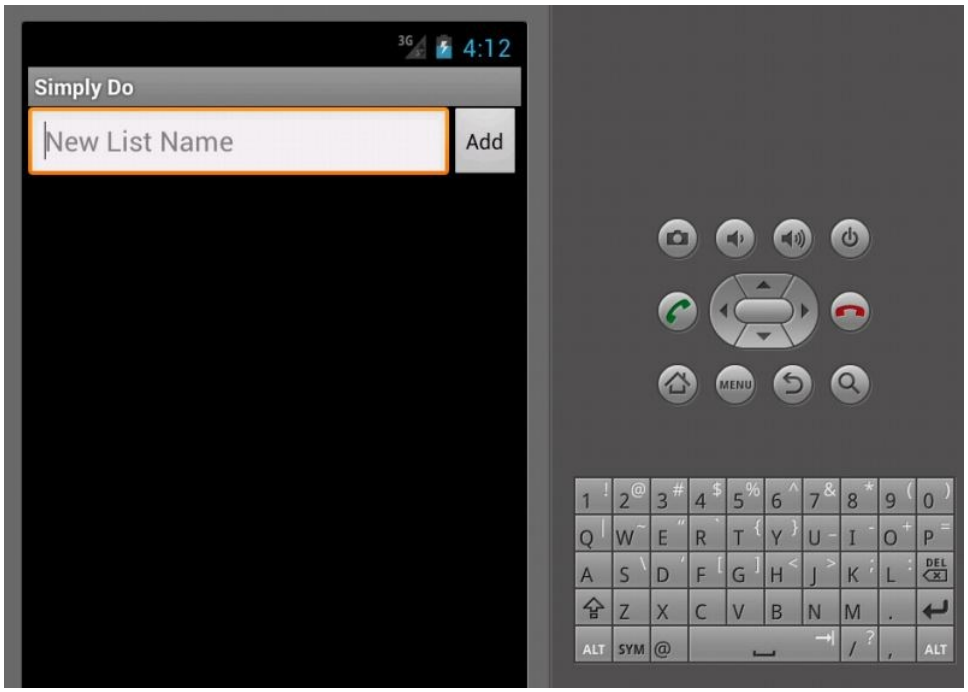
Robotium è uno strumento a supporto dell'automazione di test per applicazioni Android. Robotium è stato sviluppato nel 2010 da Renas Reda, un'autorità internazionale nell'ambito dell'automazione dei test. Questo framework consente, semplificandolo, automatizzandolo e velocizzandolo, il testing black box dell'interfaccia grafica dell'AUT, richiedendo una minima conoscenza del suo funzionamento e offrendo delle API per garantire l'interazione con essa. I test case prodotti hanno la proprietà di essere facilmente leggibili e modificabili. Il funzionamento di Robotium è basato sull'utilizzo di un oggetto denominato "solo" :

Solo solo = new Solo(getInstrumentation(),getActivity());

Tramite esso è possibile interrogare e modificare i widget della UI, eventualmente anche senza conoscerne l'identificativo. Robotium Recorder, invece, può essere visto come una estensione di Robotium in quanto si basa su di esso e quindi sulle sue API. I servizi principali che offre, Robotium Recoder, sono quelli relativi alle attività di Capture e Replay già discussi in precedenza. Nel package di test troveremo una classe che contiene codice Java e fa uso dell'oggetto "`Solo`". Questo oggetto fa parte della libreria "`com.robotium.solo`" che è indispensabile per questi progetti e contiene tutto il supporto necessario per Views, WebViews, Activities, Dialogs, Menus, Context Menus e instrumentazione. Vediamo qualche esempio in modo da capire facilmente di cosa stiamo parlando: (Figura 2.2: esempio di capture con Robotium Recorder)



Come mostrato in (Figura 2.2), Robotium Recorder sta effettuando un'attività di capture, registrando tutte le interazioni utente-dispositivo, interazioni possibili grazie al dispositivo virtuale messo a disposizione dall' AVD (Android Virtual Device), strumento dell'SDK (Figura 2.3). Tutte le azioni registrate andranno a formare la classe di test, inserite in un metodo chiamato testRun(), insieme ai metodi setUp(), tearDown() ed il costruttore (Figura 2.4).



(Figura 2.3: Dispositivo virtuale generato dall'AVD)

```
package kdk.android.simplydo.test;

import kdk.android.simplydo.SimplyDoActivity;
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;


public class SimplyDoActivityTest1BackRest extends ActivityInstrumentationTestCase2<SimplyDoActivity> {
    private Solo solo;

    public SimplyDoActivityTest1BackRest() {
        super(SimplyDoActivity.class);
    }

    public void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation());
        getActivity();
    }

    @Override
    public void tearDown() throws Exception {
        solo.finishOpenedActivities();
        super.tearDown();
    }

    public void testRun() {
        // Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
        solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
        // Enter the text: 'lista1'
        solo.clearEditText((android.widget.EditText) solo.getView(kdk.android.simplydo.R.id.AddListEditText));
        solo.enterText((android.widget.EditText) solo.getView(kdk.android.simplydo.R.id.AddListEditText), "lista1");
```

(Figura 2.4 test generato da Robotium Recorder)

Le sezioni da evidenziare sono:

- **Package kdk.android.simplydo.test**: Tipicamente Robotium Recorder suggerisce come best practice di usare come package della classe di test quello dell'AUT con l'aggiunta del suffisso ".test"
- **Extends ActivityInstrumentationTestCase2**: ActivityInstrumentationTestCase2 è una classe per il testing delle activity che ha metodi protected "setUp()", tearDown() e runTest ()
- **Metodi setUp() e tearDown()** per le fasi di inizializzazione e chiusura di ciascun test. Nel setUp() c'è l'inizializzazione dell'oggetto "Solo"
- **Metodo testRun():** deputato all'esecuzione del test

2.4.4 **Emma**

Emma è un altro tool di supporto, molto utilizzato in questo lavoro in quanto, attraverso operazioni di instrumentazione del codice sorgente, permette di valutare l'effettiva copertura del codice a seguito di una esecuzione. L'esito restituito da questo tool può essere in diversi formati (es. html, xml), quest'ultimo permette di effettuare una stima quantitativa dell'efficacia del test. In (Figura 2.5) mostriamo un

report in formato html in cui è possibile riconoscere le porzioni di codice dell'AUT attraversate dal test (colore verde) e quelle non attraversate (colore rosso):

```
161     @Override
162     public void fetchLists()
163     {
164         Log.v(L.TAG, "CachingDataView.fetchLists(): Entered");
165         ViewerTask task = new ViewerTask();
166         task.taskId = ViewerTask.FETCH_LISTS;
167         doTaskAndWait(task);
168         Log.v(L.TAG, "CachingDataView.fetchLists(): Exited");
169     }
170
171     public ListDesc fetchList(int listId)
172     {
173         ListDesc rv = null;
174
175         synchronized (viewerLock)
176         {
177             for(ListDesc list : listData)
178             {
179                 if(listId == list.getId())
180                 {
181                     rv = list;
182                     break;
183                 }
```

(Figura 2.5: esempio copertura EMMA)

EMMA Coverage Report (generated Thu Jul 16 00:52:37 CEST 2015)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [kdk.android.simplydo]

name	class, %	method, %	block, %	line, %
kdk.android.simplydo	96% (44/46)	83% (203/246)	81% (4449/5523)	79% (1030,2/1310)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
L.java	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/2)
SimpleDataViewer.java	0% (0/1)	0% (0/21)	0% (0/231)	0% (0/78)
ListListSorter.java	100% (3/3)	100% (7/7)	73% (69/95)	84% (16/19)
ItemListSorter.java	100% (4/4)	92% (12/13)	76% (170/224)	81% (39,7/49)
CachingDataViewer.java	100% (3/3)	93% (38/41)	77% (1153/1502)	79% (267,9/337)
SettingsActivity.java	100% (1/1)	100% (6/6)	81% (221/273)	81% (46/57)
RestoreActivity.java	100% (6/6)	94% (16/17)	84% (268/319)	77% (67/87)
ListDesc.java	100% (1/1)	80% (8/10)	85% (39/46)	80% (16/20)
SimplyDoActivity.java	100% (15/15)	86% (59/69)	86% (1287/1495)	83% (289,9/351)
DataManager.java	100% (2/2)	89% (16/18)	91% (613/673)	92% (132/143)
ListPropertiesAdapter.java	100% (1/1)	100% (5/5)	93% (78/84)	90% (18/20)
MoveToAction.java	100% (4/4)	87% (13/15)	93% (295/316)	90% (62,7/70)
ItemPropertiesAdapter.java	100% (1/1)	100% (5/5)	95% (116/122)	93% (27/29)
DeleteInactiveAction.java	100% (2/2)	100% (7/7)	100% (87/87)	100% (26/26)
ItemDesc.java	100% (1/1)	100% (11/11)	100% (53/53)	100% (22/22)

(Figura 2.6: Percentuale di copertura del codice dell'AUT)

Capitolo 3: Android Ripper

In questo capitolo verrà descritto il Ripper, un software sviluppato da REvERSE, gruppo di ricerca dell'Università Federico II di Napoli, capace di testare in modo completamente automatico tutta la struttura della GUI dell'AUT, generando input ed eventi fornendo in output tutto il necessario per poter effettuare le misure necessarie per rispondere ai quesiti imposti durante la fase di progettazione.

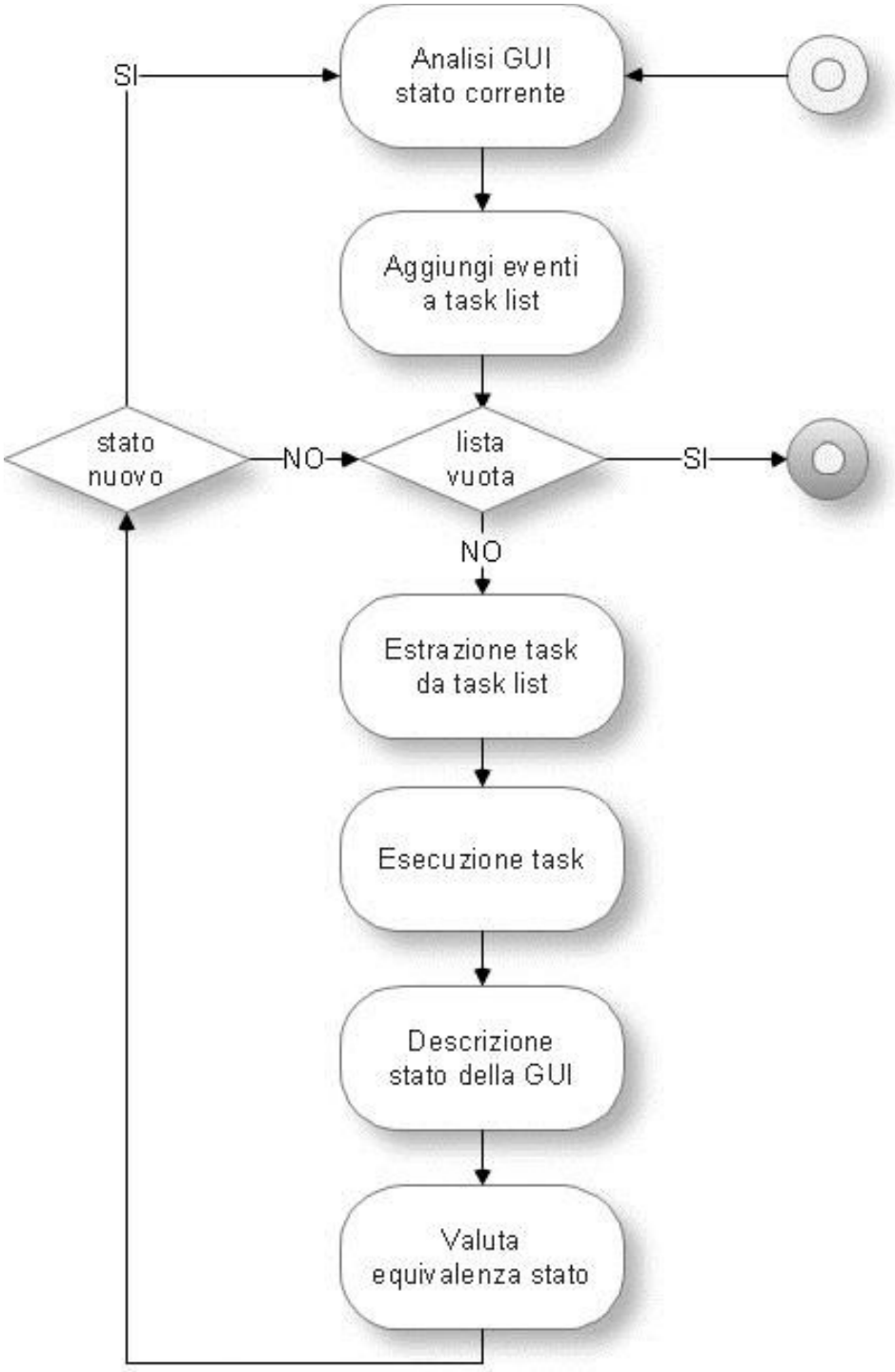
3.1 Ripper

Come specificato nell'introduzione del capitolo, il Ripper permette di effettuare attività di testing automatico generando input ed eventi, virtualizzando quindi l'interazione utente-dispositivo toccando, premendo, ruotando le activity sulla GUI ed interagendo automaticamente con i widget presenti sull'activity attiva. Nel capitolo precedente abbiamo parlato di testing e GUI, adesso è importante specificare le tecniche utilizzate per il testing della GUI dividendole in tre categorie:

- **Tecniche Model Learning:** in cui l'AUT, senza la conoscenza pregressa della struttura, viene esplorata seguendo una strategia di navigazione
- **Tecniche Random:** dove gli eventi vengono generati in maniera del tutto casuale
- **Tecniche Model Based:** dove esiste una descrizione dell'AUT utile alla generazione automatica dei test case.

Questa specifica è importante in quanto il Ripper è capace di adottarne le prime due.

L'algoritmo di base, chiamato *Processo di Ripping*, adoperato per esplorare l'AUT, può essere compreso attraverso il seguente activity diagram:



(Figura 3.1: Activity Diagram del processo di Ripping)

Prima di descrivere testualmente la figura sopracitata è importante conoscere alcuni concetti utili alla comprensione del processo di Ripping:

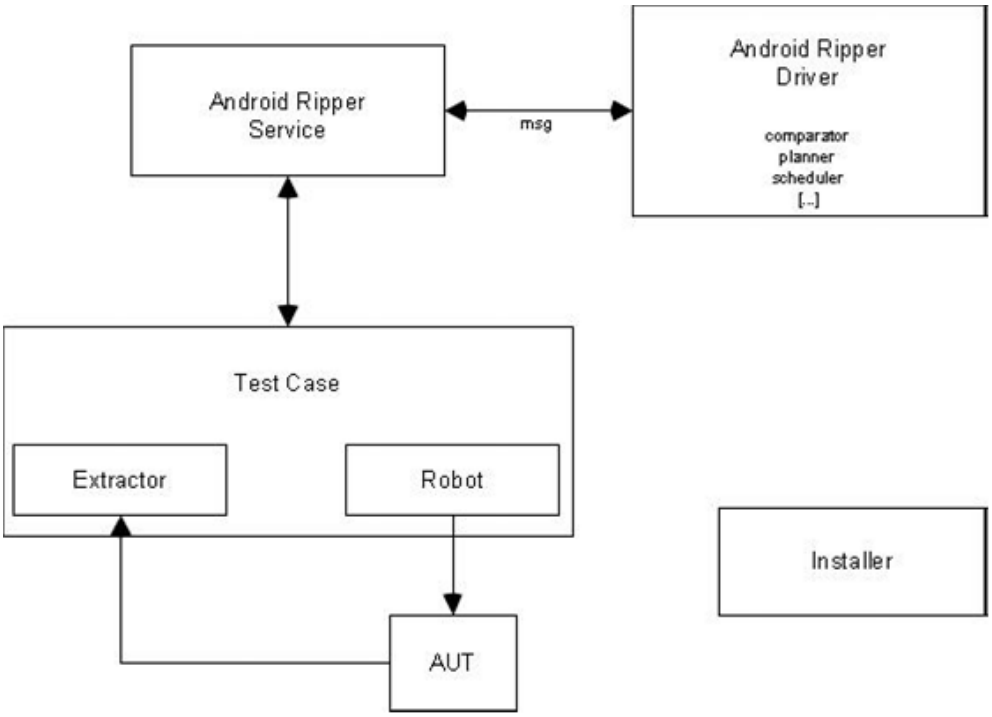
- **Activity State**: rappresenta l'Activity correntemente visualizzata (running activity), in termini dei suoi parametri e dei widget in essa contenuti; i widget saranno a loro volta descritti in termini di uno o più attributi
- **Evento**: un azione eseguita su uno dei widget della GUI in grado di determinare il passaggio da una schermata ad un'altra
- **Input**: un'interazione con la GUI dell'applicazione sotto test che non provoca cambiamenti di stato. In pratica, tutte le interazioni che non sono eventi, saranno considerate di input
- **Azione**: la sequenza ordinata di un evento e di tutti gli input che lo precedono. Essa rappresenta le operazioni necessarie ad indurre un passaggio di stato nella GUI dell'applicazione sotto test
- **Task**: coppia (Azione, GUI State) che rappresenta una azione eseguita sull'interfaccia
- **Plan**: è l'insieme dei task che descrivono le possibili azioni con le quali è possibile interagire per innescare ulteriori transizioni e proseguire l'esplorazione
- **Trace**: un rapporto sull'esecuzione di un task, corrispondente ad una traccia di esecuzione. Per ogni azione componente il task, il trace dovrà contenere lo stato dell'Activity all'inizio, la sequenza degli input, l'evento e lo stato dell'Activity alla fine della transizione
- **Sessione**: l'insieme di tutte le operazioni eseguite dal ripper, cominciando con la prima inizializzazione fino alla generazione dei file di output.

Torniamo al nostro processo di Ripping mostrato in (Figura 3.1). Inizialmente viene esplorato la stato della GUI, che rappresenta l'Activity State corrente in cui si trova l'applicazione, descritta da ciò che è contenuto in essa, come ad esempio i widget,

che a loro volta saranno descritti da propri attributi. Una volta ottenuta la descrizione dello stato, è possibile ottenere l'elenco delle azioni (eventi) scatenabili sull'Activity, filtrati in base a diversi criteri, e che sono in grado di scatenare il passaggio ad una diversa interfaccia dell'applicazione. Per ogni elemento dell'elenco viene generato un task contenente una successione ordinata di input ed eventi i quali vengono aggiunti ad un piano di test detto TaskList. Se il piano di test contiene task da eseguire, ne viene estratto uno secondo la strategia configurata e si passa alla sua esecuzione, altrimenti il processo termina rendendo, eventualmente, disponibili i suoi output. Dopo l'esecuzione del task, viene fornito un rapporto di esecuzione (trace) e dallo stato della GUI raggiunto viene estratta la descrizione. Quest'ultima è confrontata con le descrizioni degli stati già visitati; se lo stato è già presente tra quelli visitati, si passa all'estrazione del task successivo, altrimenti si analizza lo stato corrente in cerca di nuovi task da generare.

3.1.1 **Architettura del Ripper**

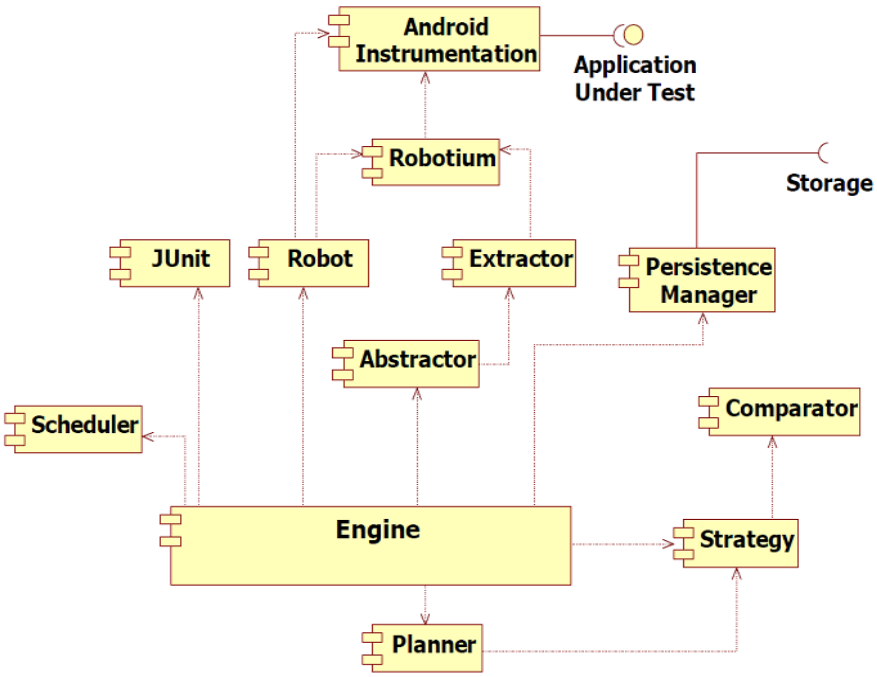
Volendo esporre l'architettura in forma concettuale, si individuano quattro componenti principali che compongono il Ripper, come mostrato in figura:



(Figura 3.2: Componenti del Ripper)

- **Driver:** realizzato come progetto Java eseguibile, in grado di gestire l'esecuzione di un esperimento coinvolgendo il GUI Ripper, ed è realizzato come applicazione Java per interagire con i dispositivi Android per mezzo di strumenti inclusi in Android Development Toolkit (ADT), come Android Debug Bridge (ADB). In particolare, mediante la componente Driver è possibile eseguire l'ambiente di test e l'applicazione sotto test sui dispositivi Android.
- **Ripper Test Case**: interagisce con la AUT per eseguire gli eventi ed estrarre le descrizioni della GUI.
- **Service**: è predisposto per mediare la comunicazione tra le altre due componenti tramite IPC (Inter-Process Communication) e le socket mediante il protocollo TCP (Transmission Control Protocol).
- **Installer**: è la componente predisposta all'installazione dell'Android Ripper. Con essa è possibile modificare i vari settaggi, le scelte sul modello di esecuzione e i dettagli riguardanti le configurazioni dell'AVD e dell'applicazione da testare.

Scendendo più nel dettaglio è opportuno mostrare la composizione architetturale del Ripper. Andiamo quindi a specificare quali sono e cosa fanno le principali componenti: (Figura 3.3: Architettura del Ripper)



Descriviamo quindi le componenti nel dettaglio:

- **Engine**: è il controllore centralizzato che racchiude la business logic del ripper. Esso gestisce il processo di ripping in ogni sua fase, supervisionando il flusso di esecuzione e garantendo lo scambio di messaggi fra le componenti
- **Scheduler/Dispatcher**: è il componente che decide l'ordine di esecuzione dei task generati per l'esplorazione dell'applicazione sotto test. Si occupa inoltre di memorizzare i task in attesa di essere eseguiti in un'opportuna struttura dati e fornisce all'Engine, di volta in volta, il task da eseguire
- **Robot:** è il componente che si occupa dell'interfacciamento con l'applicazione, ed esegue i task pianificati per esplorarla. In pratica agisce riproducendo le interazioni che un utente reale eserciterebbe per svolgere il compito descritto nel task da eseguire
- **Extractor:** ha la responsabilità di estrarre le informazioni che determinano lo stato dell'applicazione, ed in particolare la descrizione dell'interfaccia grafica dell'Activity attiva. A seguito dell'elaborazione, l'extractor mette a disposizione del Ripper una Activity Description, che contiene informazioni sulla struttura dell'interfaccia. Questa descrizione sarà una collezione di riferimenti ad oggetti del framework Android presenti nel contesto della JVM. Esempi di oggetti estratti da questo componente sono l'Activity corrente ed i widget presenti in essa
- **Abstractor:** crea e fornisce al Ripper un modello esportabile dell'interfaccia correntemente visualizzata dall'applicazione, senza fare riferimento agli oggetti effettivamente istanziati nella JVM, al fine di rendere possibile la costruzione di un modello dell'applicazione la cui validità si estenda oltre la durata della singola sessione di ripping
- **Strategy:** effettua il confronto fra lo stato corrente dell'applicazione e gli stati già visitati in precedenza. In caso di equivalenza lo stato corrente non necessita di ulteriore esplorazione da parte del ripper; Inoltre ha il compito di prendere le decisioni che guidano il flusso di esecuzione del Ripper in base al risultato del

confronto fra stati, ai dati forniti dall'Abstractor, alla descrizione dell'ultimo task eseguito ed eventualmente ad ulteriori informazioni interne al componente, quali il tempo di esecuzione del software ed il livello di profondità raggiunto

- **Planner**: ha il compito di generare il test plan che definisce le modalità di esplorazione dell'applicazione a partire dall'Activity corrente. I nuovi task saranno generati in base all'analisi del risultato dell'esecuzione del task che ha portato l'applicazione in quello stato, e solo se di tale stato è stata richiesta l'esplorazione da parte dello Strategy. Inoltre esaminando la struttura dell'istanza di interfaccia visualizzata, seleziona quei widget che in base a regole prestabilite o definite dal tester sono ritenuti in grado di innescare una transizione di stato nell'applicazione
- **Persistence Manager**: provvede a tutte le operazioni da e verso le memorie di massa (lo storage interno o la scheda SD del dispositivo) come l'apertura, la chiusura, la copia e la cancellazione di file.

Nella (Figura 3.1) si è concettualmente dimostrato il processo di Ripping, ma effettivamente qual è il suo algoritmo? Partiamo col dire che questa tecnica è divisa in 2 fasi principali: la fase di "*Inizializzazione"* e la fase di *"Setup"*. Durante la fase di *Inizializzazione* le varie componenti del ripper vengono istanziate ed inizializzate; in quella di *Setup*, invece, il Ripper si interfaccia con l'applicazione da esplorare e ne esamina lo stato iniziale popolando la lista dei task da eseguire che inizialmente è vuota. Lo stato dell'Activity iniziale, opportunamente elaborato, viene memorizzato nella lista degli stati visitati per successivi confronti, ed inviato al Planner per la compilazione di un piano di **esplorazione** contenente l'elenco dei primi task generati, che andranno successivamente eseguiti nella fase di esplorazione.

Algoritmo Setup

Input: *androidApplication* = the application under test
robot.bindApplication(*androidApplication*)
baseActivity ← *robot*.getCurrentActivity()
activityDescription ← *extractor*.describe(*baseActivity*)
activityState ← *abstractor*.abstract(*activityDescription*)
strategy.storeVisitedState(*activityState*)
plan ← *planner*.createPlan(*activityState*)
scheduler.addPlan(*plan*)

(Figura 3.4: Algoritmo di Setup)

Nella fase di *Esplorazione*, i task estratti dallo scheduler, vengono eseguiti, elaborandone i risultati e generando eventualmente nuovi task. L'algoritmo prevede che inizialmente, l'Extractor e l'Abstractor forniscano una descrizione dell'Activity corrente, ovvero quella ottenuta al termine dell'esecuzione del task; tale descrizione viene poi inviata allo Strategy per effettuarne la comparazione con gli stati visitati in precedenza. Se lo stato corrente non è equivalente a nessuno di quelli presenti nell'Activity List, allora dovrà essere aggiunto. A questo punto, lo Strategy individua se una transizione ha effettivamente avuto luogo alla fine del task eseguito: ad esempio, la pressione di un elemento di menu disattivato non produce alcun risultato. In questo caso, le operazioni riportate di seguito non vengono eseguite e l'esecuzione riprende dall'inizio del ciclo con l'estrazione di un nuovo task. In caso contrario l'esecuzione prosegue con la generazione del trace che descrive l'esecuzione del task appena terminato. Esso viene poi passato al Persistence Manager per essere memorizzato su disco. L'insieme di questi trace costituirà l'output della sessione, dal quale sarà in seguito possibile estrarre un modello a stati dell'applicazione sotto test. Lo Strategy dovrà ora decidere se lo stato corrente è passibile di ulteriore esplorazione. Nel caso più semplice, tale decisione equivale all'esito del confronto appena effettuato: lo stato verrà esplorato se e solo se non è mai stato esplorato in precedenza. Infine, si controlla se uno dei

criteri di terminazione è verificato. In tal caso, la sessione viene chiusa e si esce dal ciclo di iterazione, altrimenti si procede con l'esecuzione di un nuovo task, se disponibile.

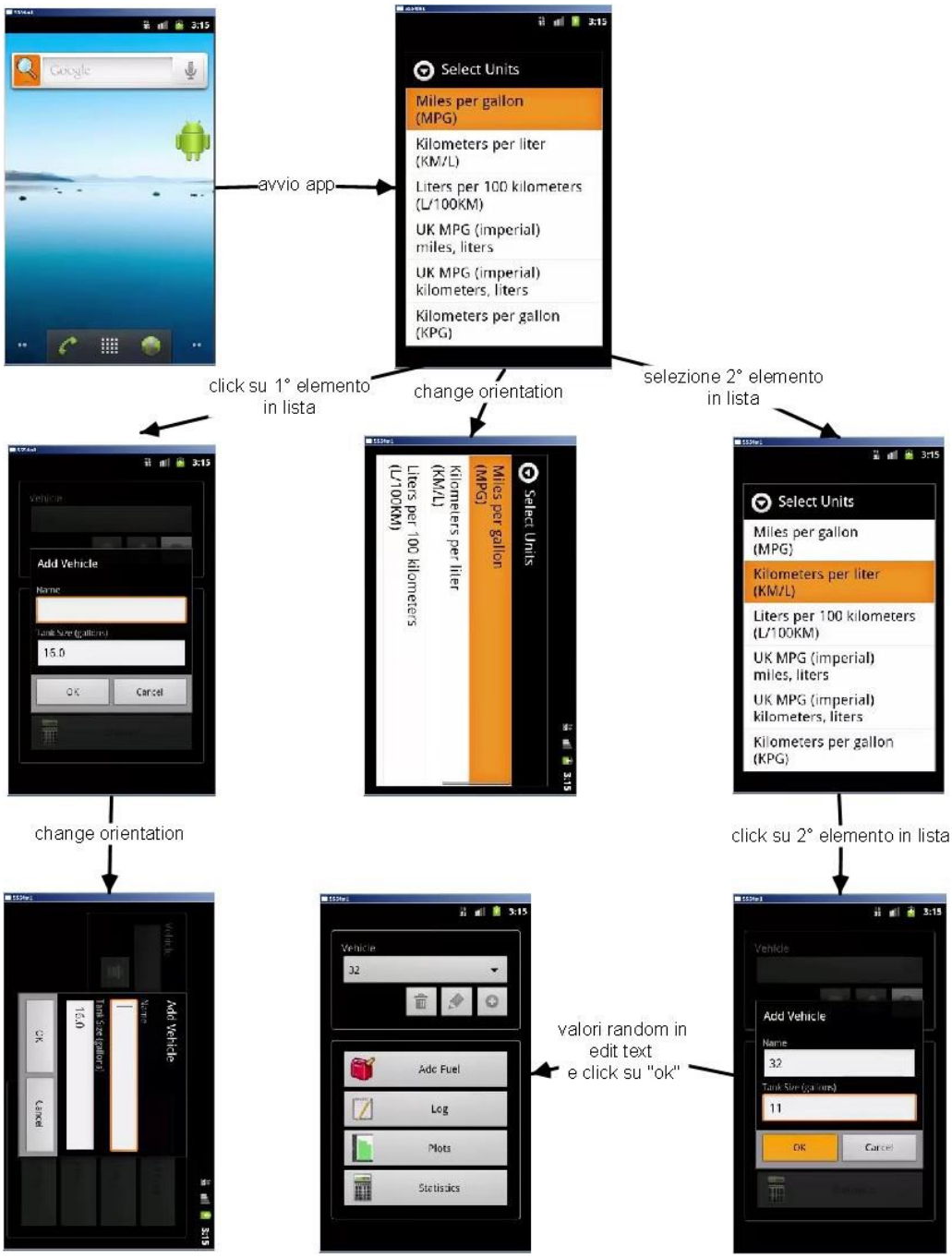
Algoritmo Exploration

```
while scheduler.hasMoreTasks() do
  task ← scheduler.getNextTask()
  strategy.setCurrentTask(task)
  robot.process(task)
  currentActivity ← robot.getCurrentActivity()
  activityDescription ← extractor.describe(currentActivity)
  activityState ← abstractor.abstract(activityDescription)
  isStateNew ← strategy.compareState(activityState)
  if isStateNew then
    strategy.storeVisitedState(activityState)
  end if
  if strategy.transitionOccurred() then
    trace ← abstractor.createTrace (task, activityState)
    persistence.addTrace(trace)
    if strategy.explorationNeeded() then
      plan ← planner.createPlan(activityState)
      scheduler.addPlan(plan)
    end if
    if strategy.sessionTermination() then
      close the session
      break the loop
    end if
  end if
end while
```

(Figura 3.5: Algoritmo di Exploration)

Al termine del processo di ripping otteniamo il seguente output:

- **I file ".bin":** non sono altro che la lista degli stati e dei task utilizzati in fase di elaborazione ed aggiornati durante l'esecuzione
- **Directory "coverage":** file di copertura generati con Emma
- **Directory "junit":** risultati dell'esecuzione dei casi di test, in formato xml
- **Directory "logcat":** è il tracciamento cronologico delle operazioni
- **Directory "model":** contiene la lista degli stati visitati ed info sugli eventi eseguiti.



(Figura 3.6: Esempio processo ripping)

Ogni stato della GUI viene rappresentato in un GUI Tree come nodo di un albero e gli eventi che provocano transizioni di stato sono rappresentati come archi. Ogni percorso che parte dall'interfaccia iniziale e segue le transizioni eventualmente, ma non necessariamente, fino ad un nodo foglia, rappresenta una traccia di esecuzione.

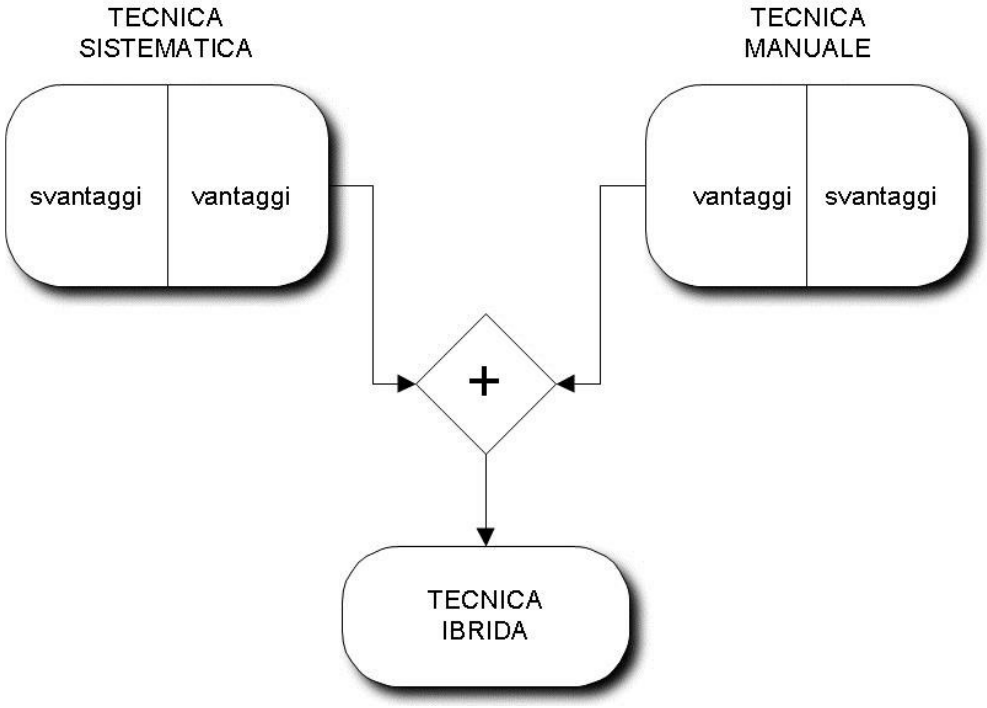
Capitolo 4: Android Ripper Ibrido

In questo capitolo si va a presentare quella che può essere definita una estensione del Ripper tradizionale presentato nel capitolo precedente. Stiamo parlando del "Ripper Ibrido" che si differenzia da quello tradizionale in quanto, quest'ultimo, opera in modo del tutto automatico mentre, in questa estensione, c'è un comportamento "ibrido" in quanto ingloba sia la parte automatica che quella manuale, quindi, con il supporto a sessioni registrate, rese possibili grazie al tool Robotium Recorder già presentato precedentemente. Ciò che resta invariato sono le operazioni relative all'estrazione e confronto delle activity description, alle operazioni di pianificazione, schedulazione ed esecuzione dei task, in poche parole lo scheletro dell'architettura già presentata resta invariato, c'è solo qualche piccola aggiunta. La differenza sostanziale sta nel modo di ragionare del ripper ibrido e quindi, cambiando la logica di funzionamento, cambia la procedura di elaborazione degli output.

4.1 Cosa si intende per "Ibrido"?

Come specificato nel capitolo 3, esistono diverse tecniche da utilizzare per il testing della GUI, tra cui, quelle di maggior interesse: Model Learning (ML) e User Based (UB). La risposta alla domanda: *"*Cosa si intende per Ibrido?*"*, la si può ricercare attraverso le caratteristiche peculiari di queste tecniche. Soffermandoci su alcune delle caratteristiche delle tecniche UB ed ML; per quanto concerne la UB abbiamo

uno svantaggio in quanto questa tecnica richiede risorse umane e quindi i test case risultanti possono essere limitati all'esperienza in merito del tester umano ottenendo così percentuali di copertura non sempre elevate ma, rispetto ad un tipo di testing che genera test case in modalità random, per esempio, è più probabile che non si ottengano risultati ridondanti. Per quanto concerne la ML, invece, pur non garantendo una massima copertura, c'è il vantaggio di operare completamente in automatico. Quindi, volendo rispondere alla domanda sopracitata, vogliamo una tecnica ibrida che sappia sfruttare a pieno le caratteristiche vantaggiose delle tecniche ML ed UB: tecnica manuale/sistematica.



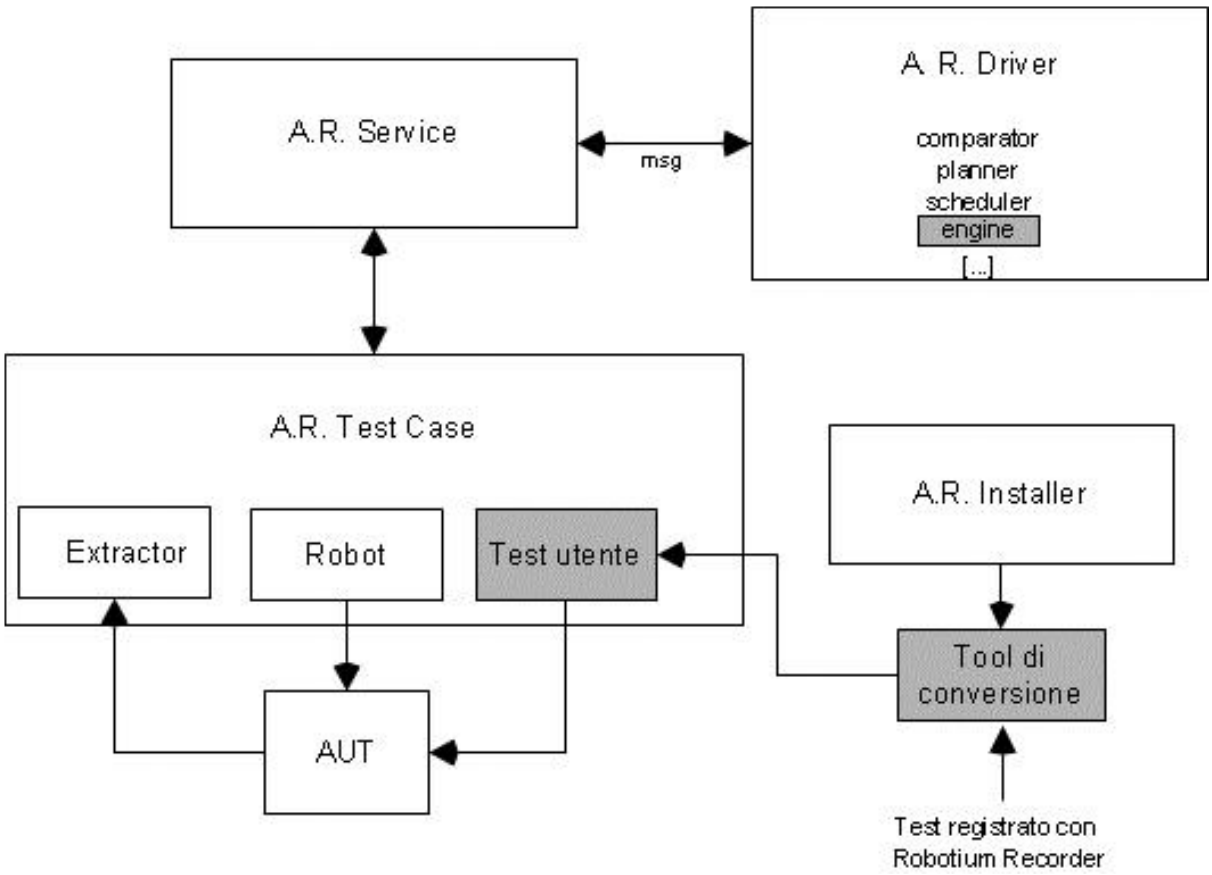
(Figura 4.1: Modello concettuale della tecnica ibrida)

Dalla (Figura 4.1) è possibile comprendere il termine "ibrido", questa estensione mette insieme le due tecniche sopracitate con lo scopo di ottenere un software più efficace. C'è da precisare che la componente sistematica altro non è che il ripper tradizionale utilizzato in quanto flessibile e predisposto a subire modifiche in quanto è suddiviso in blocchi funzionali ben identificati. Per la componente manuale,

invece, è stato necessario un software in grado di fornire servizi di capture e replay, motivo per cui la scelta è caduta su Robotium Recorder già discusso in precedenza.

4.2 Architettura

Come indicato nell'introduzione di questo capitolo, lo scheletro dell'architettura resta lo stesso del ripper tradizionale, vengono solamente aggiunti dei blocchi di elaborazione ed esecuzione dei test manuali:



(Figura 4.2: Modello concettuale dell'architettura del Ripper Ibrido)

Ciò che risulta evidente, è l'aggiunta dei tre blocchi in grigio: **Engine, Tool Di Conversione, Test Utente**. **L'engine** non è altro che il motore del software, molto più complesso rispetto a quello del ripper tradizionale, questo perché gestire la tecnica ibrida vuol dire avere un processo di funzionamento totalmente differente e più complicato. Durante la fase di *installazione*, quando in input viene posto il test

case utente, registrato con Robotium Recorder, il **tool di conversione** intercetta e rielabora il test case il quale viene eseguito, successivamente, dalla componente **Test Utente** e quindi dal ripper.

4.2.1 **Tool di conversione**

Questo tool è di fondamentale importanza in quanto fa da perno tra la tecnica manuale e quella sistematica. Esso prende in input il test case utente e, una volta controllata la presenza della classe di test specificata dall'utente nel file "ripper.properties", prima che sia eseguito dal ripper, rielabora il test rendendo il ripper ibrido in grado di conoscere tutti gli stati che è possibile attraversare con il test case corrente. In pratica crea una nuova classe di test chiamata *"RobotiumTest.java"* in cui sono presenti i **tagli** effettuati sul test. Questi tagli, variabili rispetto a ciascun test case, vengono effettuati in corrispondenza di eventi ben precisi i quali potrebbero far scoprire nuovi stati al ripper ibrido. Una volta identificati i tagli, si procede alla composizione dei diversi metodi testRun. Ciascun metodo possiede una nuova struttura. Nel nome del metodo testRunX (Solo solo), la X è un numero progressivo da 1 al numero di tagli, mentre il parametro "solo" di tipo "Solo" è necessario per l'invocazione del metodo stesso tramite reflection. Il corpo del metodo testRunX contiene tutte le istruzioni dall'inizio del metodo testRun originale fino all'istruzione identificata come taglio X. Quindi in definitiva si passa dalla registrazione originale, racchiusa in un solo test case del metodo "testRun ()", alla creazione di una molteplicità di test case. La presenza dell'istruzione "solo.sleep(1000)" al termine di ciascun metodo "testRunX" (assente nel metodo originario "testRun()") si è resa necessaria per la corretta estrazione dell'activity description durante il processo di cattura delle informazioni sulla schermata. Il valore "1000" rappresenta i millisecondi di attesa. Per quelle applicazioni con una GUI particolarmente complessa è necessario un aumento di tale valore. Il compito del tool si conclude smistando il "nuovo" file di test, contenente i tagli, in una specifica cartella del ripper in modo tale da poter essere reperito facilmente da esso:

(\Release\AndroidRipperInstaller\AndroidRipper\src\it\unina\android\ripper).

È opportuno conoscere quali sono gli eventi\stringhe che permettono i tagli:

- ***"solo.goBack()"*:** simula la pressione del tasto "back"
- ***"solo.clickOnView()"*:** simula il click o il tocco su una specifica view
- ***"solo.clickInList()"*:** simula il tocco su un elemento di una lista visualizzata
- ***"solo.waitForActivity()"*:** attende che un'Activity richiesta venga visualizzata

Prendendo come esempio una parte del codice di test generato, tramite Robotium Recorder, da uno degli studenti del corso di Ingegneria del Software II dell'Università degli studi di Napoli Federico II, che hanno testato l'App "FillUp", mostriamo come il tool di conversione crea i tagli e genera il file di test adattato per il Ripper Ibrido:

```
public void testRun() {
    // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
    solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
    // Wait for dialog
    solo.waitForDialogToOpen(5000);
    // Sleep for 9960 milliseconds
     //9960);
    // Click on Kilometers per liter (KM/L)
    solo.clickOnView(solo.getView(android.R.id.text1, 1));
    // Wait for activity: 'com.github.wdkapps.fillup.MainActivity'
    assertTrue("com.github.wdkapps.fillup.MainActivity is not found!",
    solo.waitForActivity(com.github.wdkapps.fillup.MainActivity.class));
```

(Figura 4.3.1: Parte di codice di Test generato con Robotium Recorder)

Questa parte di codice non fa altro che attendere l'apertura dell'Activity di StartUp dell'applicazione, successivamente si attende che si apra una finestra di dialogo che permetta di far selezionare all'utente l'unità di misura preferita (in questo caso KM/L), infine c'è da soddisfare un *Assert* che risulta vero nel caso in cui non si apra l'Activity principale dell'App (*MainActivity.class*).

RobotiumTest.java

```java
package it.unina.android.ripper;

import com.github.wdkapps.fillup.StartupActivity;
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;


public class RobotiumTest extends ActivityInstrumentationTestCase2<StartupActivity> {
    private Solo solo;

    public RobotiumTest() {
        super(StartupActivity.class);
    }

    public void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation());
        getActivity();
    }

    @Override
    public void tearDown() throws Exception {
        solo.finishOpenedActivities();
        super.tearDown();
    }

//il numero tc = al numero tagli; da restituire al driver
public static final int num_tc = 198;

    public void testRun1(Solo solo){
        this.solo = solo;
        // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
        solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
        solo.sleep(1000);
    }

```

(Figura 4.3.2: esempio di test case rielaborato dal tool di conversione)

```java
public void testRun1(Solo solo){
    this.solo = solo;
    // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
    solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
    solo.sleep(1000);
}

public void testRun2(Solo solo){
    this.solo = solo;
    // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
    solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
    // Wait for dialog
    solo.waitForDialogToOpen(5000);
    // Sleep for 9960 milliseconds
     //9960);
    // Click on Kilometers per liter (KM/L)
    solo.clickOnView(solo.getView(android.R.id.text1, 1));
    solo.sleep(1000);
}

public void testRun3(Solo solo){
    this.solo = solo;
    // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
    solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
    // Wait for dialog
    solo.waitForDialogToOpen(5000);
    // Sleep for 9960 milliseconds
     //9960);
    // Click on Kilometers per liter (KM/L)
    solo.clickOnView(solo.getView(android.R.id.text1, 1));
    // Wait for activity: 'com.github.wdkapps.fillup.MainActivity'
    assertTrue("com.github.wdkapps.fillup.MainActivity is not found!", solo.waitForActivity(com.github.wdkapps.fillup.MainActivity.class));
    solo.sleep(1000);
}
```

(Figura 4.3.3: esempio di test case rielaborato dal tool di conversione)

Le figure (4.3.2 e 4.3.3) mostrano come il tool abbia adattato il file di test. Dalla (4.3.2), oltre ai metodi di **"SetUp"** e **"TearDown"** per l'apertura e la chiusura delle Activity, si nota il numero "**num_tc**" dei tagli effettuati dal tool. Dalla (4.3.3) è possibile vedere i vari metodi **"testRun"**, tanti quanto il numero dei tagli nel file completo. In questo esempio sono mostrati tre tagli effettuati in corrispondenza delle seguenti istruzioni (presenti nella figura 4.3.1):

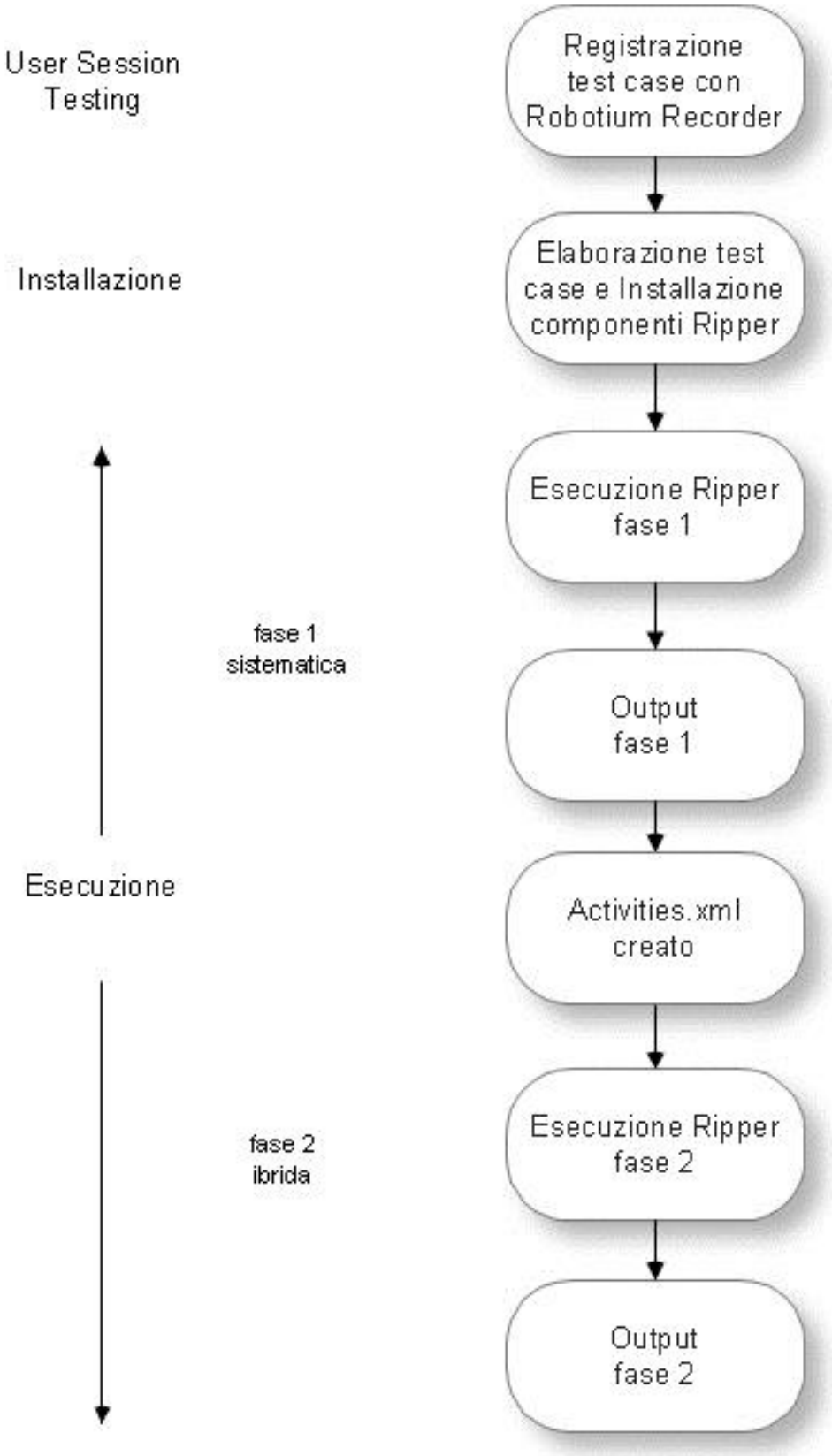
- *solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);*
- *solo.clickOnView(solo.getView(android.R.id.text1, 1));*
- *solo.waitForActivity(com.github.wdkapps.fillup.MainActivity.class));*

È importante notare come ad ogni incremento dei "testRun" si riparte dalla prima istruzione, del file di test non adattato, fino all'aggiunta dell'istruzione che ha causato il taglio in quel punto.

4.3 Processo di esecuzione

Il Ripper tradizionale prevede una singola esecuzione così come per il Ripper Ibrido, la differenza sta nel fatto che per quest'ultimo, la singola esecuzione, è composta da due fasi distinte, senza sovrascrittura dei file prodotti tra una fase e l'altra, sempre nel rispetto del requisito di non richiedere alcuna operazione da parte dell'utilizzatore del software. La discriminante tra le due fasi è la presenza del file "Activities.xml". Nel caso questo sia presente nella cartella di output "model", il Ripper comprende che si tratta della seconda fase, scatenando una serie di operazioni aggiuntive rispetto a quelle eseguite nella prima fase.

Vediamo ora il processo generale che governa la logica di esecuzione dei test del ripper ibrido attraverso il seguente modello concettuale:



(Figura 4.4: Processo generale di funzionamento del ripper ibrido)

In figura possiamo identificare tre sezioni. La prima sezione riguarda la registrazione del test case con Robotium Recorder; è l'unica sezione che richiede il supporto dell'utente, le altre saranno svolte in maniera automatica.

La seconda sezione riguarda l'installazione e l'elaborazione automatica della classe di test secondo una procedura discussa in seguito.

La terza sezione racchiude il funzionamento del Ripper Ibrido, diviso in due fasi. La prima fase corrisponde all'esecuzione del Ripper tradizionale e consente di effettuare una prima esplorazione dell'applicazione, con conseguente collezione degli stati trovati. Uno stato è un'activity description, una descrizione dell'istanza di interfaccia corrente. In particolare uno degli output di tale fase è il file "Activities.xml" che contiene appunto gli stati e rappresenta l'elemento in base al quale sono diversificate le due fasi.

Nella prima fase l'esplorazione è totalmente sistematica, a differenza della seconda, il cuore della tecnica ibrida. Proprio in quest'ultima fase è previsto l'utilizzo del test registrato. La classe di test non è utilizzata direttamente ma deve essere elaborata in modo da essere eseguito in maniera congiunta al Ripper. La registrazione rappresenta un unico test case ottenuto attraverso l'interazione con la GUI dell'applicazione. È pertanto composta da tutti gli eventi e input effettuati sui singoli widget. Come visto nel capitolo 3, si parla di "evento" quando l'interazione causa un cambiamento di stato; in caso contrario si parla semplicemente di "input". Dopo aver analizzato tutte le possibili interazioni con i widget previsti dal framework Robotium (come ad esempio click su menu, pressione tasto back, click in caselle di testo) sono stati selezionati tra queste gli eventi, elencati successivamente. Il singolo test case registrato, pertanto, è una collezione di statement corrispondenti ad input o a eventi. Gli statement considerati come eventi saranno presi in considerazione per suddividere automaticamente il singolo test case iniziale in un insieme di test case. Si individuano pertanto nel test case tante sezioni o "tagli" quanti sono gli statement corrispondenti ai soli eventi. La struttura dei test case risultanti verrà analizzata nei

paragrafi successivi; l'importante è sottolineare che la suddivisione nasce dalla necessità di riconsiderare le sequenze più o meno lunghe di test generate dagli utenti, come composte da tanti test aventi una precondizione: racchiudere tutti gli statement precedenti al taglio considerato a cui si aggiunge come statement finale un evento (statement di taglio). In base a questa caratteristica, l'esecuzione di ogni test case conduce sicuramente a uno stato a partire dal quale si valuta l'eventuale copertura. Solo quando lo stato in cui si è giunti non è equivalente a nessuno degli stati già scoperti, l'idea è di avviare una nuova esplorazione sistematica, ma a partire da quello specifico stato trovato, incrementando in tal caso la copertura rispetto a quella già ottenuta e potenzialmente scoprendo automaticamente nuovi stati. La componente manuale viene pertanto sfruttata per raggiungere porzioni dell'applicazione inesplorate (sfruttando uno dei vantaggi della tecnica manuale) lasciando poi il campo all'esplorazione metodica (vantaggio questo della ML), per eseguire un'attività di test dell'applicazione più approfondita rispetto a quella ottenibile dalle tecniche manuale e sistematica considerate singolarmente.

4.3.1 **Funzionamento**

Passiamo alla descrizione dettagliata delle 2 fasi principali del ripper ibrido.

1° fase: Prevede la procedura di installazione attraverso il comando:

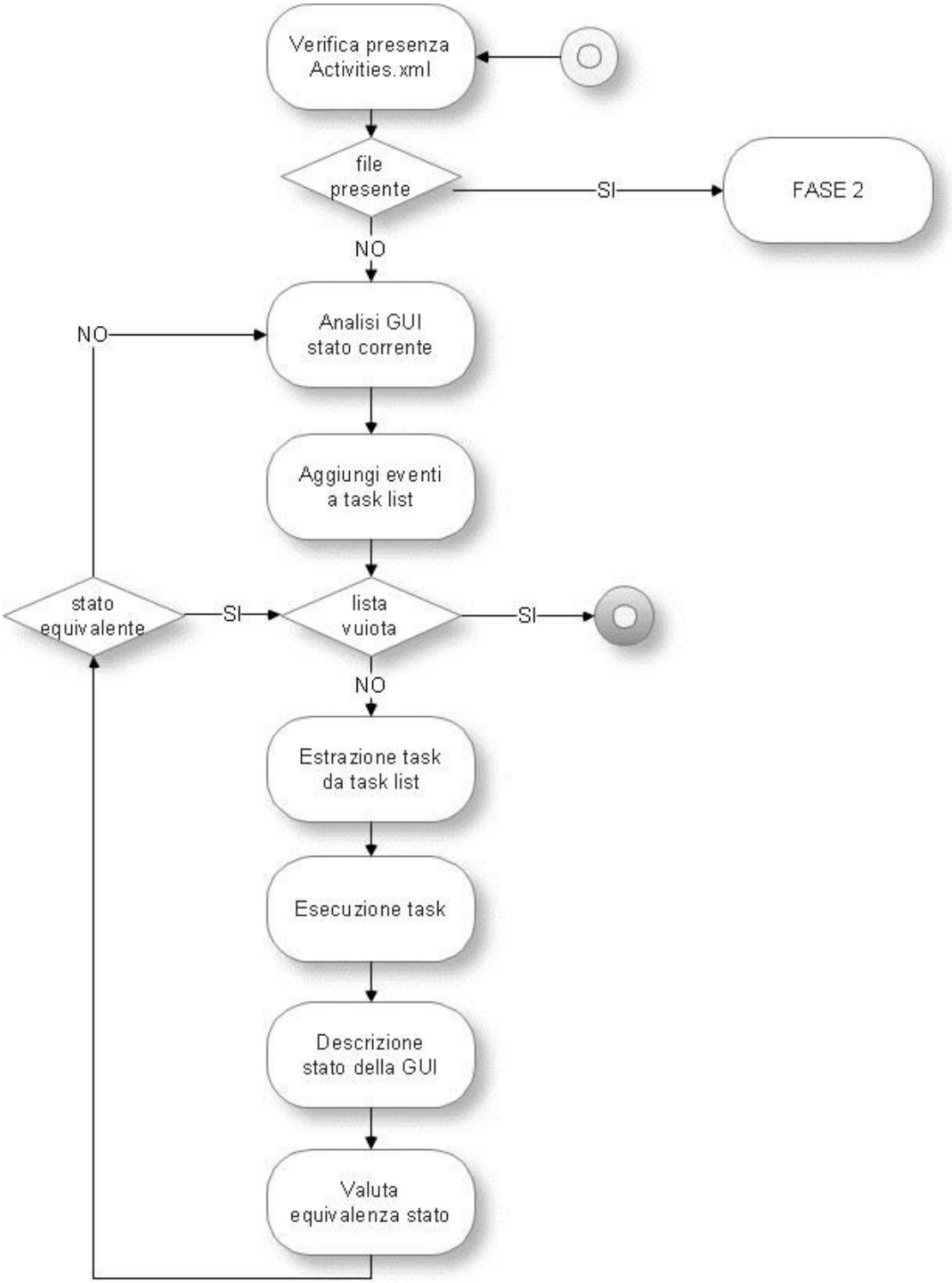
"java -jar AndroidRipperInstaller.jar"

cui segue l'elaborazione vera e propria attraverso il comando:

"java -jar AndroidRipper.jar s systematic.properties".

Questa prima fase è esattamente la stessa del Ripper tradizionale consentendo pertanto di generare gli stessi output (i file di coverage, log, e Activities.xml) modificati nel nome per evitare sovrascritture. Questa prima fase consente di esplorare automaticamente la GUI dell'applicazione, collezionando una serie di activity description nel file "Activities.xml". L'algoritmo di esecuzione esamina di

volta in volta gli stati scoperti. Per ciascuno stato nuovo verrà generata una task-list di eventi che andranno a sollecitare lo stato stesso. Il processo terminerà quando saranno esaminate tutte le activity scoperte e la task-list sarà vuota:



(Figura 4.5: Ripper Ibrido, 1° fase)

2° fase: Al termine della fase 1, otteniamo in output gli stessi file del Ripper tradizionale. Per ottenere tale risultato sono state settate delle condizioni booleane e inseriti statement di controllo in più parti del codice.

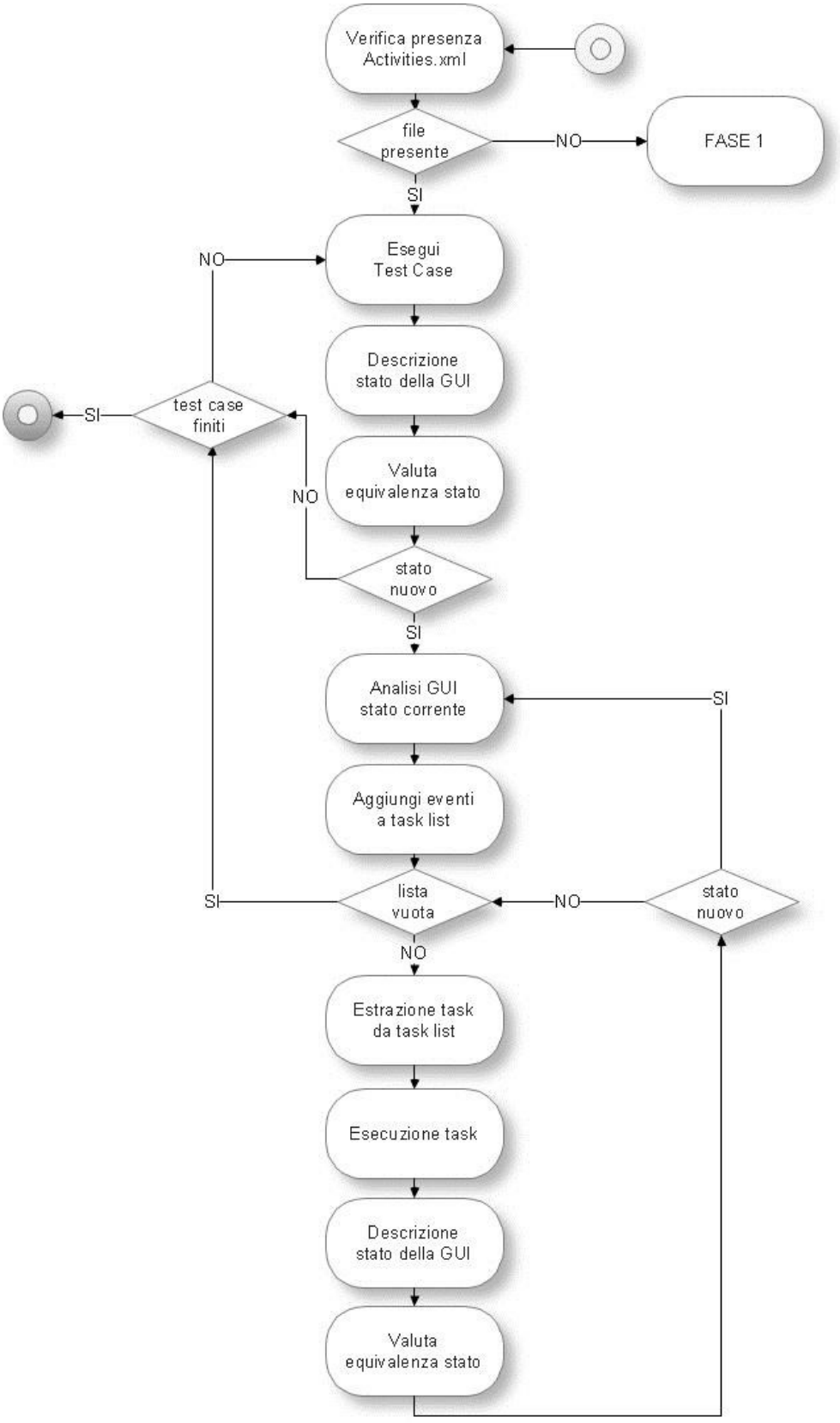
La fase 2 vede una nuova esecuzione del Driver attraverso lo stesso comando

"java -jar AndroidRipper.jar s systematic.properties".

Sebbene il comando sia il medesimo della fase 1, il comportamento atteso è differente. In questo caso, infatti, il Ripper vede l'esistenza del file "Activities.xml" creato nella prima fase ed effettua una serie di operazioni totalmente differente.

N.B. Rimandiamo all'Appendice B le operazioni di inizializzazione delle componenti del ripper.

L'activity diagram seguente sintetizza le operazioni appena descritte:



(Figura 4.6: Ripper Ibrido, 2° Fase)

La macro-attività "FASE1" rappresenta l'activity diagram mostrato in (Figura 4.5). Nel diagramma è possibile identificare due cicli innestati. Il ciclo esterno governa l'esecuzione dei test case e ha come criterio di terminazione il raggiungimento del numero totale di questi, previsto dalla registrazione. Il ciclo interno ha come criterio di terminazione la task list vuota come visto nella fase 1, in quanto rappresenta il processo di esplorazione del Ripper tradizionale, innescato tante volte quanti sono gli stati nuovi scoperti dai metodi della classe di test. Rifacendoci alla (Figura 4.2: Architettura del Ripper Ibrido), analizziamo nel dettaglio le attività del processo di ripping: nella seconda fase, il "Driver" invia un messaggio, per conoscere il numero totale di test case utente il quale rappresenterà il criterio di terminazione del ciclo esterno e quindi della intera fase 2 (terminati i test case, l'esecuzione terminerà). Fin quando tale numero non è raggiunto, il driver invia un messaggio, appositamente creato per l'esecuzione dei test case utente, al componente "Test Case", il quale, esegue le operazioni e invia in risposta l'activity description sempre incapsulata in un messaggio. Il Driver, ottenuta l'activity description, la confronta con tutte quelle prelevate dal file "Activities.xml". Nel caso sia già presente, si passa direttamente al successivo test, altrimenti viene eseguito il processo di ripping ma a partire dal nuovo stato scoperto. Supponiamo di eseguire il test case i-esimo. Al termine dell'esecuzione viene estratta l'activity description corrispondente e confrontata con quelle già presenti. Consideriamo nel dettaglio i due casi:

- **Stato equivalente**: Nel caso in cui lo stato estratto a seguito dell'i-esimo test case risulta essere equivalente a uno degli stati già visitati, il ciclo interno non viene eseguito e si considera direttamente il test case successivo, se presente
- **Stato non equivalente**: L'activity description viene salvata nel file Activities.xml ed inizia nuovamente il processo di ripping ma a partire dallo stato nuovo scoperto. L'applicazione sarà forzata in questo stato dal driver, previo scambio di un ulteriore messaggio, prima della pianificazione dei task

e della conseguente esecuzione degli stessi. È possibile che, a partire dallo stato considerato, il Ripper scopra nuovi stati in maniera automatica. Terminato il processo di ripping, si passerà al test case successivo.

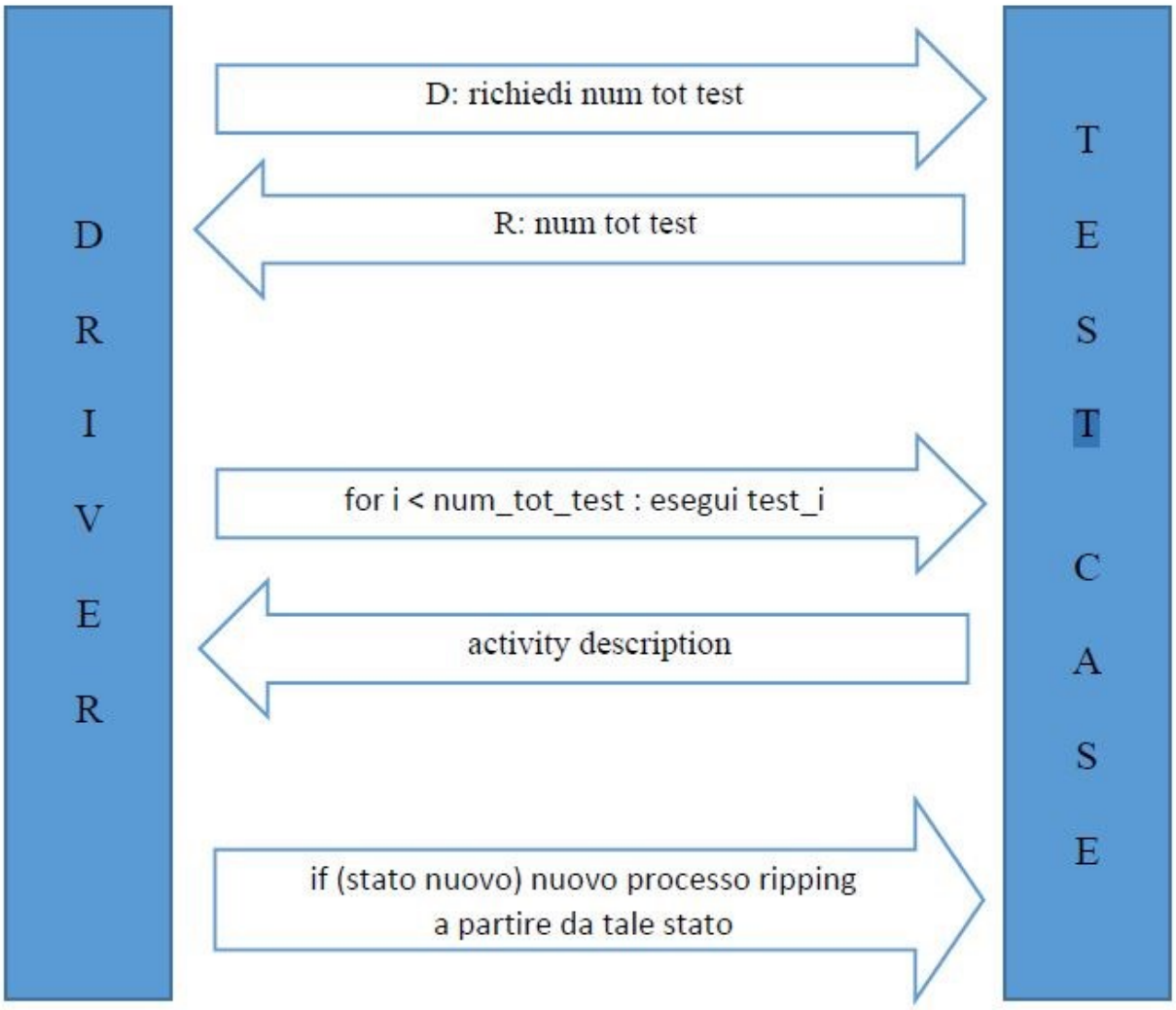
La classe che racchiude il principio di funzionamento è "SystematicDriver.java", profondamente modificata nel Ripper Ibrido, di cui mostriamo un metodo in particolare *"protected void rippingLoop( )"*, in cui sono racchiusi i due cicli "do-while" innestati che governano il processo di ripping:

```
 // ciclo su controllo test_case finiti do {
 // loop ripper
   do {
   (…)
} while (running && this.terminationCriterion.check() == false);
       // condizioni di uscita ripper
closeStateDescriptionFile();
       //controllo su raggiungimento num totale di test case
if  (num_tc_corrente  <  num_tot_testcase  &&  esecuzione1  ==  false)  {
              bootstrap = false; // devo rientrare nel bootstrap perchè
                                 // num_tc_corrente<num_tot_testcase
               System.out.println("Test utente non terminati");
               } else {
                     tc_finiti = false; //test utente terminati
       System.out.println("Test utente terminati");
                      break; // esci dal loop; testcase finiti
               }
   } while (tc_finiti == true);
   // loop fin quanto i test case non sono terminati
```

Il ciclo interno (*while (running && this.terminationCriterion.check() == false);*) ha come condizione di terminazione la task list vuota dopo aver esaminato tutti gli stati trovati, come visto nella fase 1. Il ciclo esterno (*while (tc_finiti == true);*) ha come criterio di terminazione il raggiungimento del numero dei test previsti, basato sulla variabile di controllo booleana "tc_finiti", impostata inizialmente a "true". Sarà impostata a "false", con conseguente uscita dal ciclo, quando è stato raggiunto il numero totale di test case oppure nel caso di fase 1 (quest'ultima condizione è identificata nel codice con la variabile booleana "esecuzione1" impostata a "true").

4.3.2 **Messaggi**

Come visto nel paragrafo precedente, il processo di ripping è governato da uno scambio di messaggi tra componente Driver e componente Test Case. In particolare, la figura seguente illustra tale scambio relativamente alla seconda fase del Ripper, per la quale sono stati creati tre nuovi messaggi oltre a quelli esistenti:



(Figura 4.7: scambio di messaggi tra le componenti del Ripper Ibribo, fase 2)

Nello specifico, nella classe “MessageType.java” abbiamo la stringa identificativa dei tre messaggi, rispettivamente per eseguire il test case i-esimo, per conoscere il numero totale di test case e infine per forzare l’applicazione all’eventuale stato nuovo trovato:

```
public static final String TEST_UTENTE_MESSAGE = "TEST";
public static final String NUM_TOT_TEST_CASE = "NUMTC";
public static final String ESEGUI_TEST_CASE_PER_DRIVER = "ETC";
```

I messaggi non sono altro che degli HashMap contenenti delle stringhe, difatti la classe "Message.java" estende la classe HashMap:

```
public class Message extends HashMap<String, String>{
(…)
// messaggio per exec test case
public static Message getRunMessage(String i)
{
	Message msg = new Message(MessageType.TEST_UTENTE_MESSAGE);

	msg.addParameter("test", i);
	(è la coppia chiave-valore)
	return msg;
}
// messaggio per num tot test case
public staticMessagegetNumTestCaseMessage()
{
return new Message(MessageType.NUM_TOT_TEST_CASE);
}
// messaggio per forzare stato
public static Message getTestCasePerDriver(String string)
{
	Message msg = new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
	msg.addParameter("runner", string);
	return msg;
}
// messaggio per forzare stato
public static Message getTestCasePerDriver()
{
	Message msg = new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
return new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
	}
(…)
 }
```

I valori restituiti, ossia il numero totale di test case e l'activity description sono anch'essi incapsulati in messaggi inviati dal componente Test Case e letti dal Driver. La classe che regola lo scambio nel Driver (invio - ricezione) è "RipperServiceSocket.java". La componente Test Case resta in attesa dei messaggi. A seconda della tipologia ricevuta (Message Type) esegue specifiche operazioni.

4.3.3 **Output**

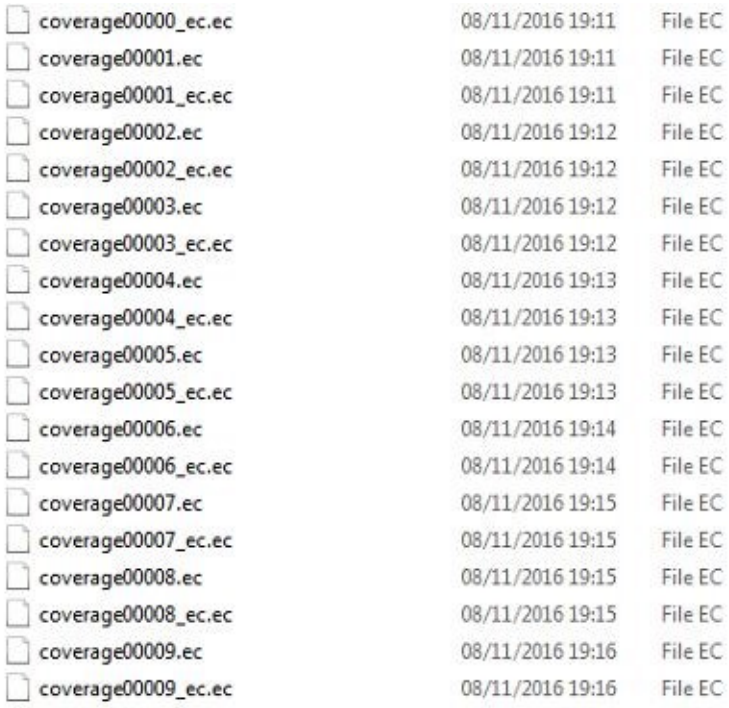
Diamo una breve descrizione di ciò che viene restituito in output sia al termine della prima che della seconda fase. Seguendo il path:

...\AndroidRipper_Ibrido\Release\AndroidRipperDriver

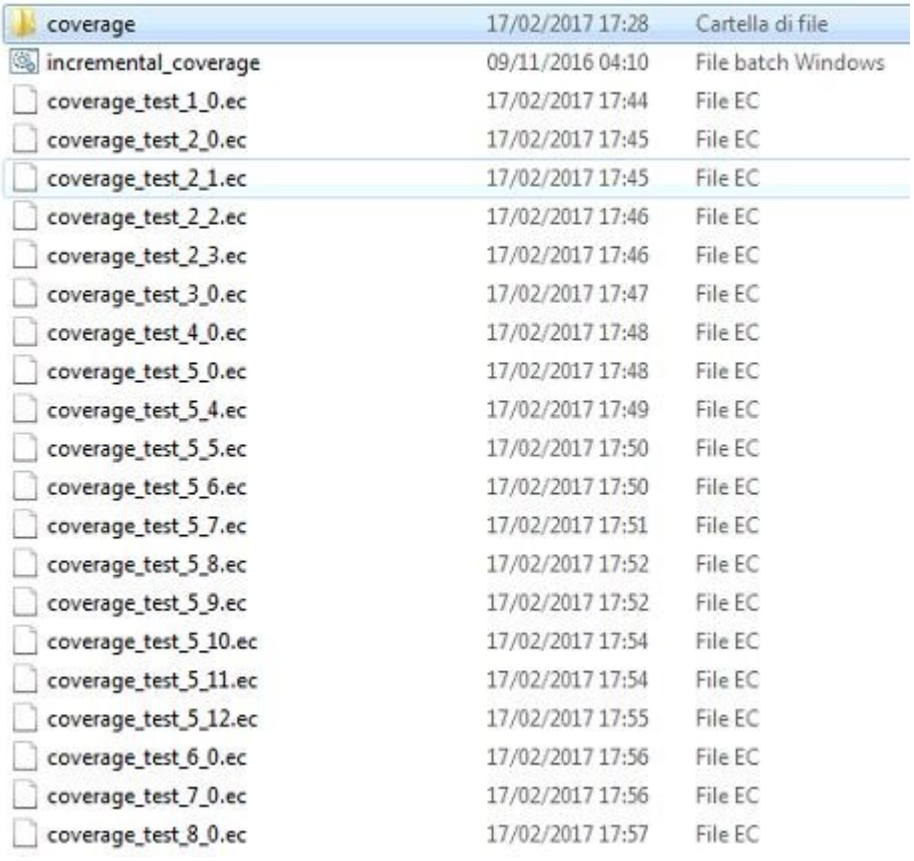
al termine di una delle due fasi, è possibile avere accesso alle seguenti directory:

- **Coverage:** è la directory che ha maggior rilevanza in quanto, in essa, è possibile trovare tutte le informazioni di copertura.
- **Junit:** directory contenete tutti i file *Junit-log-test*
- **Logcat:** file non importanti ai fini dell'analisi
- **Model:** contiene il file activities.xml ed altri file di log

C'è da precisare che il file activities.xml è lo stesso di cui si è discusso in precedenza, cioè quel file contenente informazioni parziali rispetto alla fase 1, in cui non sono riportati i widget della GUI analizzata e le relative caratteristiche. Concentriamo di più l'attenzione sulla prima directory menzionata, quella Coverage: prendiamo come esempio l'output generato al termine della fase 1 e 2 dell'applicazione MunchLife.

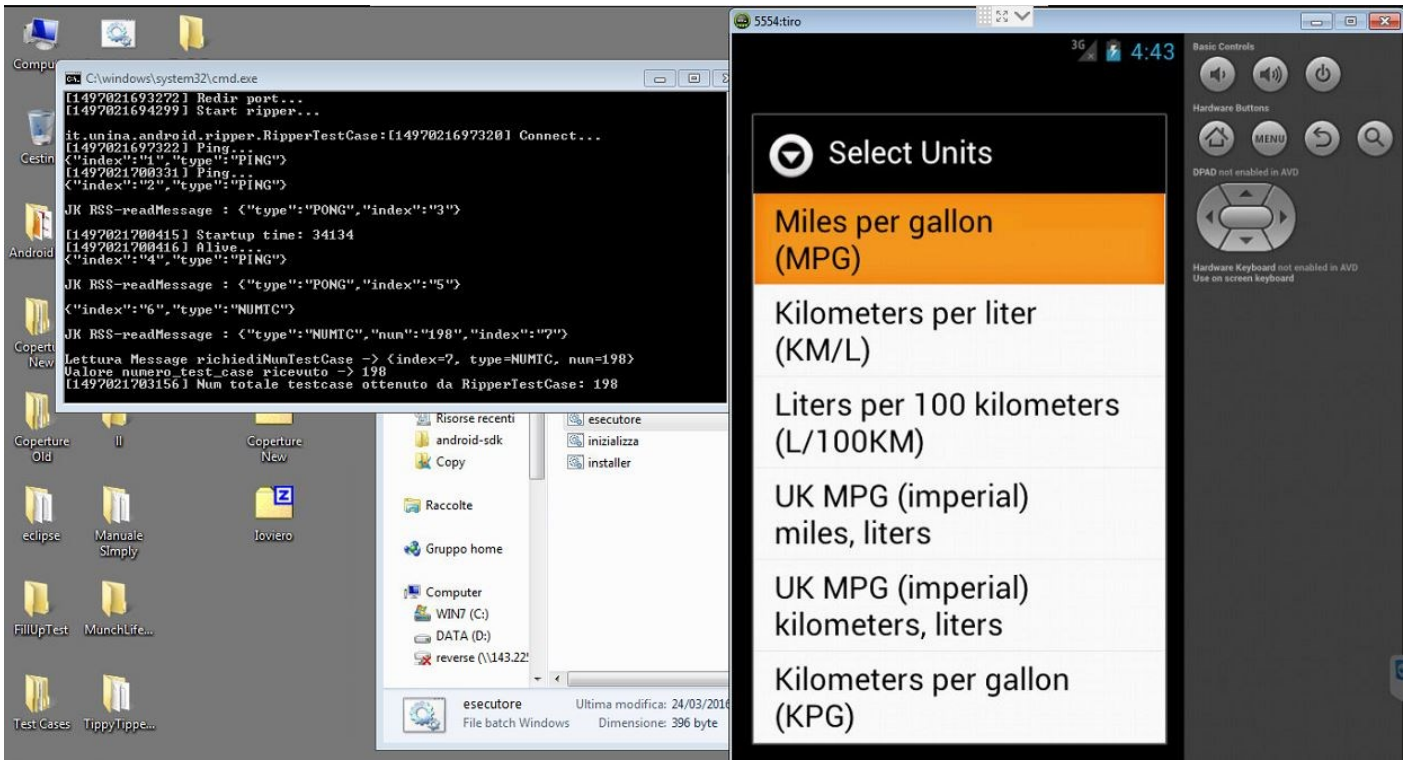


(Figura 4.8: Esempio di files di coverage al termine della fase 1)

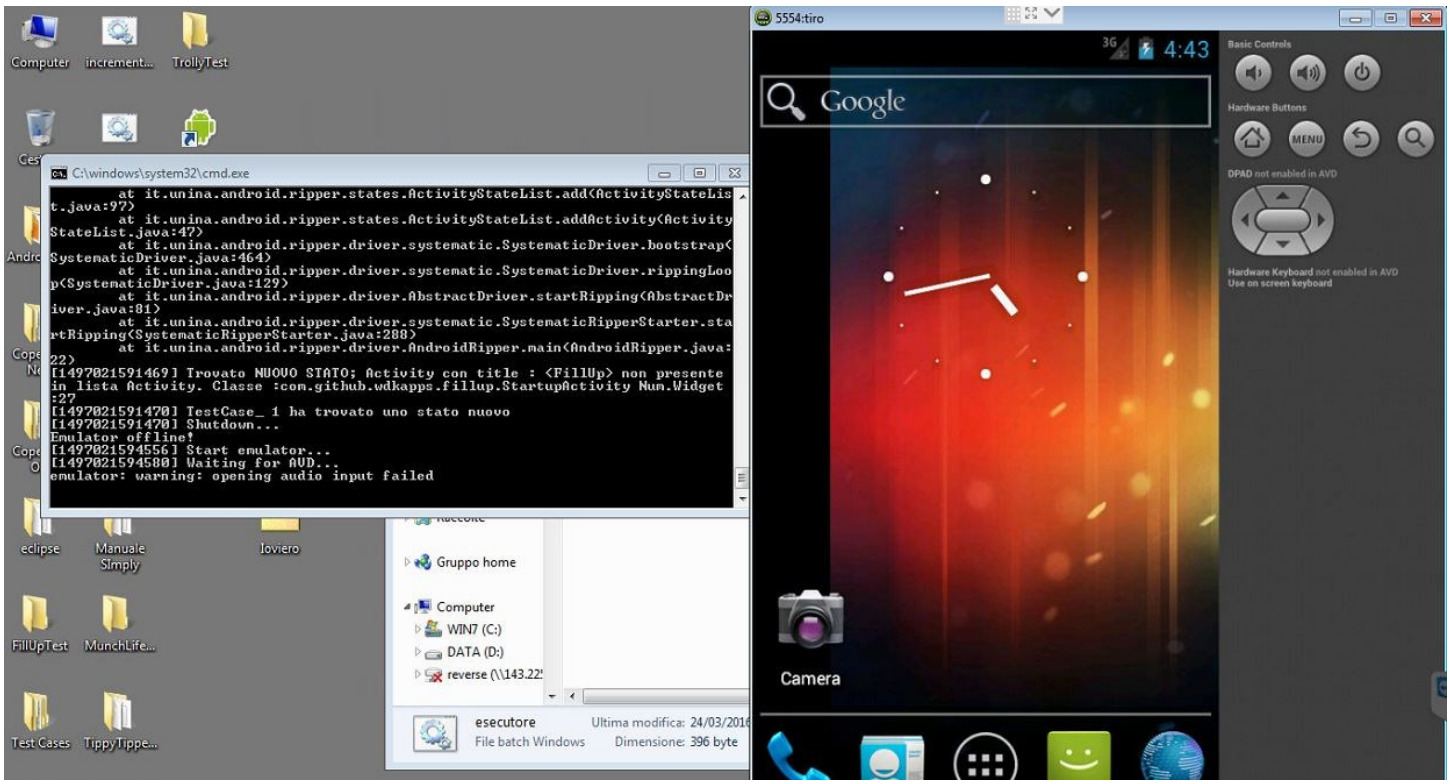


(Figura 4.9: Esempio di files di coverage al termine della fase 2)

È importante sottolineare che nella cartella coverage, al termine della fase 2, troveremo sia i file di coverage della (Figura 4.8: fase 1) che quelli della (Figura 4.9: fase 2) in quanto, a monte della seconda fase, è importante lasciare l'output della fase 1 nella cartella AndroidRipperDriver altrimenti si perderebbe il concetto di ibrido. Sarà quindi necessario creare una copia di backup dell'output della prima fase in modo tale da poter avviare la seconda senza dover ripetere ogni volta il testing sistematico per quell'AUT (es. gli esiti del testing ibrido tra il testing sistematico e tutti i test manuali raccolti da alcuni studenti ed utilizzati ai fini di questa tesi). Dando un'occhiata alla struttura dei nomi usati per i file di coverage (es. "coverage_test_2_1.ec") il primo numero ("2" nell'esempio) è il numero di test case scatenante la scoperta del nuovo stato; il secondo numero ("1" nell'esempio) rappresenta il numero di evento schedulato automaticamente dal Ripper a partire dal nuovo stato (ad esempio "back" o "change orientation"). L'analisi della correttezza dell'output ha invece rilevato una errata elaborazione del file Activities.xml per alcune applicazioni. In particolare gli stati nuovi venivano rilevati ma il processo di estrazione della descrizione non veniva completato correttamente. Il problema è stato individuato nel tempo necessario per il prelievo di tutte le informazioni riguardanti i widget costituenti la schermata al termine di ciascun test case. Come soluzione è stato impostato un tempo di 1000 millisecondi per consentire il corretto prelievo. A seguito di questa modifica, anche il file Activities.xml risulta essere correttamente costruito. Per la maggior parte delle applicazioni testate, questo valore si è dimostrato sufficiente. Per concludere, è stato creato uno script ad-hoc (CoverageScriptGeneratorSeparate), che fa uso di EMMA, per mettere insieme tutti i file di coverage e creare un unico file di coverage generale, in formato html, che ci dà la possibilità di visionare le percentuali di copertura del codice e quali parti di esso sono coperte (es. Figura 2.5 e Figura 2.6, del paragrafo 2.4.4, figure relative all'output di emma EMMA).



(Figura 4.10: Il Ripper, a monte della fase 2, riceve il numero di test case dopo che il test manuale è stato processato dal tool di conversione)



(Figura 4.11: Il Ripper Ibrido scopre un nuovo stato)

Capitolo 5: Esperimento di confronto tra tecniche di testing

In questo capitolo sarà presentata l'intera fase di sperimentazione, in particolare si analizzeranno tutti i dati raccolti, posti in forma tabellare, con cui, attraverso analisi statistiche, sfruttando le opportune metriche, possiamo rispondere alle research question. Verranno, quindi, messe in discussione l'analisi qualitativa e quella quantitativa in modo da giustificare i risultati ottenuti e trarre delle conclusioni.

5.1 Goal

Date 5 applicazioni e 13 studenti, i quali, indipendentemente, hanno testato le App e generato codice di test manuali, attraverso l'uso di Robotium Recorder, è necessario stabilire quale delle tecniche di testing, tra quelle utilizzate, risulta essere più efficiente. Le tecniche di testing messe a confronto sono:

- **Testing Manuale:** attraverso l'uso di Robotium Recorder
- **Android Ripper**: testing sistematico di tipo ML (Model Learning)
- **Android Ripper Ibrido**: testing manuale + ML
- **Unione:** (sistematico)U(manuale)

5.2 Research Question

Le *research question* sono molto utilizzate in attività di ricerca in quanto rappresentano metodologicamente un punto di partenza per questo tipo di attività. Grazie a questa metodologia è possibile definire il tipo di studio che lo scrittore

effettua permettendo, inoltre, di definire gli obiettivi ed ottenere esiti in forma netta. È importante sottolineare che, prima di sviluppare le research question, è fondamentale definire il tipo di studio da voler affrontare (qualitativo, quantitativo, misto). Sulla base degli obiettivi prefissati, per questa attività di sperimentazione è stato necessario operare secondo il tipo di studio misto in quanto, per valutare comparativamente le prestazioni delle diverse metodologie di testing provate, è stato ritenuto opportuno effettuare sia un'analisi quantitativa che un'analisi qualitativa.

Le research question proposte sono:

RQ 1) Il Ripper Ibrido consente di migliorare il livello di efficacia rispetto a quello ottenuto dall'utilizzo del solo Testing Manuale?

RQ 2) Il Ripper Ibrido consente di migliorare il livello di efficacia rispetto a quello ottenuto dall'utilizzo della sola tecnica sistematica (ML)?

RQ 3) Il Testing Manuale risulta più efficace rispetto alla tecnica ML?

RQ 4) L'unione tra la tecnica Manuale e quella Sistematica risulta essere più efficace delle due tecniche di cui è composta se considerate singolarmente?

RQ 5) L'unione tra la tecnica Sistematica e quella Manuale risulta essere più efficace rispetto alla tecnica ibrida?

Di seguito discuteremo i risultati ottenuti in modo da rispondere a queste domande.

5.3 Configurazione

Come spiegato nel paragrafo **5.1**, tutta l'attività di ricerca si è svolta all'interno del DIETI (Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione), dell'Università degli studi di Napoli Federico II, utilizzando un'unica macchina in modo tale da essere sicuri di sfruttare tutte le risorse, tutte le entità, necessarie al raggiungimento degli obiettivi, sotto le stesse condizioni, configurazioni e caratteristiche della macchina stessa.

La macchina utilizzata presenta le seguenti caratteristiche:

- **Produttore:** ASUSTeK COMPUTER INC.
- **Modello:** ASUS Desktop PC CM6870 Series
- **Sistema Operativo:** Windows 7 Home Premium SP1
- **Tipo Sistema:** Sistema Operativo a 64 bit
- **Processore:** Intel Core i5-3330 3.00GHz
- Memoria RAM: 4,00 Gb

Per la riesecuzione dei test case manuali e l'esecuzione del Ripper Ibrido è stato utilizzato lo stesso dispositivo virtuale di Android con le seguenti caratteristiche:

- **Device:** 4.0" WVGA (480*800: hdpi)
- **Target:** Android 4.0.3 – API Level 15
- **CPU/ABI:** ARM (armeabi-v7a)
- **Skin:** Skin with hardware controls
- VM Heap: 32
- **RAM:** 700 Mb
- Internal Storage: 400 MiB
- **SD Card:** 128 MiB
- Snapshot: Yes

Sul PC, inoltre, sono installati:

- JDK ver. 1.8.0_60
- Android SDK Manager Revision 22.3
- Android SDK Platform Tools ver. 19.0.2
- Android Build Tools ver. 18.1.1
- Libraries for Android 4.0.3 (API 15)
- Apache-ANT-1.9.6

5.3.1 **Le AUT (Application Under Test)**

Le applicazioni utilizzate, per i test manuali e per confrontare le varie tecniche di test, sono cinque, tutte disponibili su internet o su Google Play. Diamo ora una breve descrizione delle App in modo da capirne, in parte, le funzionalità:

- **SimplyDo**, ver. 0.9.2: è una semplice applicazione Android che consente di gestire azioni da svolgere, organizzate in liste.
- **Tippy Tipper**, ver.1.2: è un'applicazione Android per il calcolo della mancia da lasciare per il servizio in un ristorante.
- **Trolly** ver.1.4: è un'applicazione Android per la gestione di liste della spesa.
- **MunchLife** ver.1.4.4: è una semplice applicazione utile a tener conto del livello raggiunto da un personaggio e dei bonus (gear) da esso accumulati nel contesto di una partita al gioco di ruolo Munch.
- **FillUp** ver.1.7: è una applicazione con cui è possibile gestire i consumi di carburante della propria automobile pianificando un percorso.

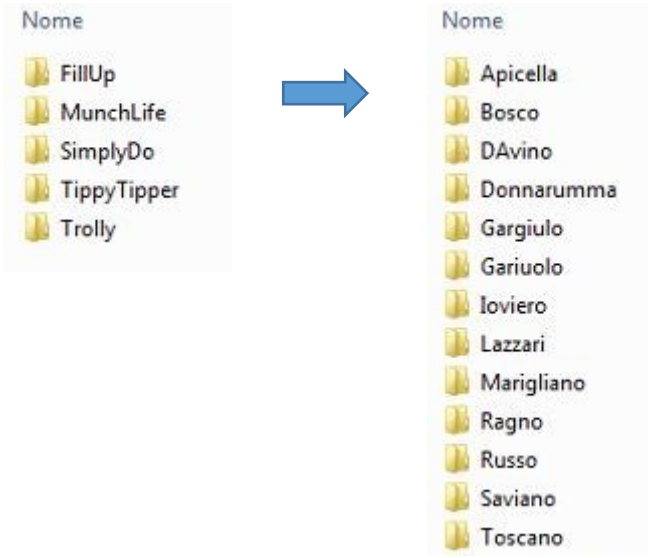
	Classi	Metodi	Blocchi	LOCs	Activities
SimplyDo	46	246	5523	1281	3
TippyTipper	42	225	4253	999	6
Trolly	19	64	1888	364	2
MunchLife	10	28	841	184	2
FillUp	105	669	18096	3807	10

(Figura 5.1: Informazioni relative al codice sorgente delle 5 App utilizzate)

5.3.2 **Test Case Manuali**

I test case manuali sono stati ottenuti attraverso esperimenti messi in atto da 13 studenti i quali, in veste di Tester, hanno fatto parte del corso di Ingegneria del Software II, della laurea magistrale dell'Università degli studi di Napoli Federico II, che, per il loro elaborato di esame, hanno testato le applicazioni Android presentate

raccogliendo test con lo strumento Robotium Recorder, avendo come obiettivo la copertura di tutti gli scenari di esecuzione che riuscivano ad individuare con una ispezione delle applicazioni in esecuzione. Di seguito è possibile osservare l'organizzazione delle App e degli studenti sotto forma di cartelle, in cui saranno memorizzati tutti i file relativi alle coperture:

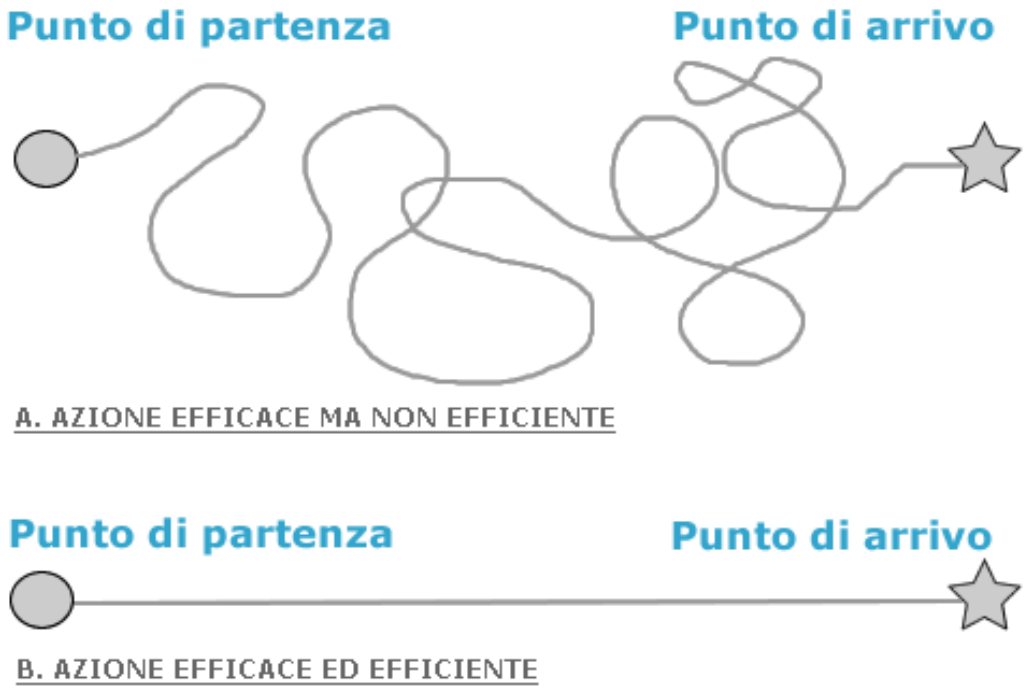


5.4 Efficacia ed Efficienza

I termini **efficacia** ed **efficienza**, spesso usati indistintamente come sinonimi, riflettono in realtà due concetti ben distinti. In un contesto generico:

- **L'efficacia** indica la capacità di raggiungere l'obiettivo previsto
- **L'efficienza** valuta l'abilità di farlo impiegando le risorse minime indispensabili

È possibile mostrare tramite un semplice disegno la differenza sostanziale tra i due:



(Figura 5.2: Esempio di Efficacia ed Efficienza)

Come si evince dall'immagine, nel caso dell'efficacia è importante solo raggiungere l'obiettivo indipendentemente dall'energia spesa, nel caso dell'efficienza, invece, risulta evidente che l'energia spesa per raggiungere il punto di arrivo dev'essere minima. Detto ciò è possibile affermare che: *"Siamo molto efficaci e molto efficienti quando raggiungiamo il massimo spendendo il minimo"*. Nel nostro caso, rendere valida quest'affermazione, risulta molto difficile in quanto, comunemente, aumentare una delle misure vuol dire influenzare negativamente l'altra. In ambito informatico, nello specifico quello di Testing, la differenza tra queste due misure risulta essere molto sensibile in quanto, anche se usate insieme, permettono di effettuare valutazioni diverse, ad esempio permettono di rispondere a research question diverse, tutto si basa sulle variabili e sulle metriche con cui vogliamo rappresentare la misura **dell'efficacia** e **dell'efficienza**. Nel nostro caso è possibile definire le due misure come:

- **Efficacia:** indica la scoperta di quanti più difetti possibili;
- **Efficienza:** valuta l'abilità di farlo impiegando le risorse minime indispensabili

Definite in questo modo, l'efficacia e l'efficienza, rappresentano delle regole di base su come dev'essere un'attività Testing. Volendo, tramite esempio, caratterizzare le misure attraverso alcune delle possibili metriche a disposizione, definiamo:

- **Efficacia:** numero di difetti scoperti / numero di difetti esistenti
- **Efficienza:** numero di casi di test rilevanti difetti / numero di casi di test esistenti

Se ipotizziamo di aumentare ambedue le misure, provando ad aumentare il numero di test case eseguiti per aumentare l'efficacia, non stiamo facendo altro che aumentare le risorse a disposizione, siamo quindi in contrasto con la definizione di efficienza. Allo stesso modo, se diminuiamo il numero di test case, ad esempio considerando solo test corrispondenti a situazioni critiche, in modo tale da aumentare l'efficienza, non stiamo facendo altro che diminuire la possibilità di scoprire quanti più difetti possibili e quindi diminuendo la probabilità di raggiungere gli obiettivi prefissati, in contrasto con la definizione di efficacia. Con ciò abbiamo dimostrato che risulta difficile validare l'affermazione sopracitata, si rende quindi necessario trovare il giusto compromesso tra i due, per cui risulta conveniente sapere che l'efficacia va privilegiata quando si vuole un software affidabile, ad esempio per applicazioni critiche), l'efficienza, invece, va privilegiata quando si vuole ridurre lo sforzo allocato nelle attività di testing, in particolare se non può essere eseguita automaticamente. In conclusione, sulla base dell'ultima definizione di efficacia ed efficienza, bisogna precisare che, tranne sé iniettati di proposito in ambito sperimentale, possiamo conoscere solo i difetti scoperti e non quelli presenti, altrimenti l'attività di Testing non avrebbe senso, per cui ci si affida a misure indirette legate alla copertura del codice sorgente del software, ad esempio:

Sia **LOC** = Lines Of Code = Linee di codice

Sia **Cov** = Coverage = Copertura

- **CovLOC:** LOC coperte da almeno un caso di test / Totale LOC
- **CovMetodi:** Metodi coperti da almeno un caso di test / Totale Metodi
- **CovClassi:** Classi coperte da almeno un caso di test / Totale Classi

La copertura può essere interpretata come una "speranza" di verifica, nel senso che se una riga difettosa non viene eseguita, sicuramente non si riscontrerà il fallimento, in caso contrario c'è la possibilità, non certa, di riscontrarlo. Per rispondere alle research question imposte, si è fatto ampio uso di queste metriche indirette mostrate in forma tabellare nel proseguo di questo capitolo. Infine, si evidenzia che, ai fini di questo lavoro di tesi, ci occuperemo principalmente di efficacia più che di efficienza.

5.5 Variabili e Metriche

Riprendendo il discorso del paragrafo precedente, per misurare l'efficacia delle tecniche è opportuno utilizzare delle misure indirette in quanto non è possibile sapere quanti difetti sono presenti all'interno del codice sorgente di un'App. Per effettuare i dovuti confronti ci siamo basati su una metrica ben precisa, stiamo parlando della copertura del numero delle classi, dei metodi, e delle linee di codice (LOC coperte), copertura ottenuta grazie all'utilizzo del tool EMMA. In questo lavoro di tesi discuteremo dell'efficacia misurata principalmente in termini di copertura delle LOC, per essere precisi, si è fatto riferimento al totale delle LOC eseguite rispetto alle LOC complessive delle AUT, esempio con SimplyDO:

APP	TESTER	TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
SimplyDo	Ripper (Fase1)	700	526	919,9	1281	0,7181108509	72
	Apicella	4,7	90	906,3	1281	0,7074941452	71
	Ibrido(Apicella)	600	294	1070	1281	0,8352849336	84
	(Apicella)U(Sistematico)	-	-	1062,6	1281	0,8295081967	83
	Bosco	5,16	140	1071,1	1281	0,8361436378	84
	Ibrido (Bosco)	405	267	1077,1	1281	0,8408274785	84
	(Bosco)U(Sistematico)	-	-	1076,1	1281	0,8400468384	84

(Figura 5.3: Esempio di utilizzo della metrica di copertura delle LOC)

In questa immagine è possibile vedere la colonna ELOC e quella TLOC, rispettivamente Executed LOC e Total LOC, utilizzate in modo tale da ottenere la percentuale di copertura raggiunta, riportata in colonna (% COV), dalle diverse tecniche di testing. A tal proposito possiamo esprimere l'efficacia come:

$$\varepsilon = \left(\frac{Executed\ LOC}{Total\ LOC}\right) * 100$$

precisiamo che ELOC/TLOC rappresenterà l'insieme delle distribuzioni da mettere a confronto, sfruttate successivamente per attività di analisi statistica. Per quanto riguarda le variabili è importante definire sia le variabili indipendenti che quelle dipendenti. L'unica variabile indipendente a disposizione è rappresentata dalla tecnica, quelle dipendenti, invece, subiscono mutazioni dovute all'utilizzo della variabile indipendente, e vengono rappresentate dalla copertura (coverage) e dal tempo di esecuzione delle tecniche di test (figura 5.3). È importante sottolineare che il tempo di esecuzione, relativo al testing manuale, viene fornito direttamente da EMMA al termine della sua attività, mentre, per ottenere il tempo di esecuzione relativo alla tecnica ibrida, è stato necessario controllare il tempo di creazione dei file coverage_test, cioè i file contenenti le coperture parziali, generati dal ripper ibrido passo dopo passo, precisamente si va a consultare la colonna "Ultima modifica" all'interno della cartella coverage del ripper ibrido affinché si possa

conoscere il tempo trascorso dalla generazione del primo file di copertura all'ultimo, infine il tempo, per lo più espresso in ore, è stato convertito in minuti in modo tale normalizzare i valori di questo parametro sotto la stessa unità di misura.

5.5.1 **Unione di più tecniche**

Una delle variabili indipendenti in (Figura 5.3), presente nella colonna TESTER, non ancora descritta, è la tecnica dell'Unione [es. (Apicella)U(Sistematico)]. Questa tecnica permette di procedere con un'attività di unione tra la misura dell'efficacia ottenuta con la tecnica sistematica e la misura dell'efficacia ottenuta con la tecnica manuale. Questa misura è ottenuta effettuando un merge a line di comando, utilizzando EMMA, dei file di coverage ottenuti con l'utilizzo delle due tecniche citate attraverso il seguente comando:

java emma report -r txt,html -in coverage[manuale].ec -in coverage[sistematico].ec

Questo comando restituisce in output due file di copertura, uno in formato .txt e un altro in .html, ottenuti facendo il merge dei due file di coverage.ec posti in ingresso. È importante tenere in considerazione che, data l'unione dei due valori di efficacia, il valore ottenuto da questa unione non può essere né minore dei due valori di efficacia di partenza né maggiore del valore di efficacia ottenuto con la tecnica ibrida, per essere più chiari:

- Sia $\mathcal{E}_S$ il valore dell'efficacia ottenuto con la tecnica sistematica;
- Sia $\mathcal{E}_M$ il valore dell'efficacia ottenuto con la tecnica manuale;
- Sia $\mathcal{E}_I$ il valore dell'efficacia ottenuto con la tecnica Ibrida;
- Il valore rappresentativo dell'efficacia ottenuto con l'unione delle due tecniche deve essere contenuto nel seguente range di valori:

$$\varepsilon_U \in [\min(\varepsilon_S; \varepsilon_M), \max(\varepsilon_S; \varepsilon_M; \varepsilon_I)]$$

5.6 Metodologia di analisi

Una volta raggruppati sulla macchina ed organizzati in base al cognome dello studente, i test manuali hanno subito un primo processo di riesecuzione, attraverso Eclipse con annesso dispositivo virtuale così, in caso di esito positivo, cioè corretta esecuzione, i test sono stati controllati a livello di codice in quanto bisognava soddisfare il sospetto di trovare codice aggiunto manualmente dagli studenti e quindi non generato automaticamente da Robotium Recorder. Questa seconda fase di controllo risulta essere necessaria in quanto il tool di conversione dei test trova difficoltà ad effettuare tagli lì dove sono presenti costrutti iterativi (per lo più sono stati trovati cicli For nei test degli studenti, risolti "sciogliendo" i cicli e copiando le serie di istruzioni presenti tante volte quanto indicato dagli indici), questo perché i tagli vengono effettuati su precise istruzioni generate da Robotium Recorder (vedere paragrafo 4.2.1). Superata questa seconda fase si è passati ad una seconda riesecuzione dei test manuali, in modo tale da essere certi che il test modificato producesse gli stessi effetti del test privo di modifiche. Grazie a quest'attività di riesecuzione dei test case degli studenti è stato possibile rilevare la caratteristica di alcuni di essi di incorporare, in un unico file, sia il codice di test relativo al testing di tipo Black-Box che quello relativo al testing White-Box. Questa scoperta risulta essere di fondamentale importanza in quanto, per effettuare i dovuti confronti tra le varie tecniche di testing, è necessario che i test manuali siano eseguiti tutti in modalità Black-Box, ovvero, per dire che un tester umano porta a risultati migliori/peggiori rispetto ad uno automatico, c'è bisogno che il tester non conosca il codice, altrimenti partirebbe "in vantaggio". Questo ha portato ad isolare la maggior parte dei test White-Box mentre, per uno studente, non è stato chiaro il tipo di testing effettuato causa assenza di meta-dati. Una volta rieseguiti tutti i test manuali in modalità Black-Box, attraverso l'utilizzo di EMMA, è stato possibile misurare le coperture in termini di linee di codice ed ottenere così i primi risultati. Il passo successivo è stato quello di sfruttare i file di test manuali, opportunamente

modificati, per far sì che il ripper ibrido potesse essere eseguito. Dato che il ripper dev'essere configurato ogni volta che si cambia applicazione, è stato ritenuto opportuno configurarlo una sola volta per App, quindi, eseguendo prima il ripper tradizionale, ottenendo la copertura della tecnica sistematica relativa all'AUT (Ripper Fase 1), ed eseguendo in successione la Fase 2 per tutte le App partendo ogni volta dai risultati ottenuti dalla Fase 1. Terminata l'esecuzione di tutte le Fasi, attraverso l'apposito tool, è stato possibile ricavare tutti i dati di copertura relativi alla tecnica ibrida. A questo punto, avendo a disposizione i file di copertura sia della tecnica sistematica che quella manuale, attraverso attività di merge dei file, si sono ricavati gli esiti di copertura dell'unione tra le due tecnica citate. Infine, raccolti i dati, organizzati successivamente in forma tabellare, è stato possibile avere una prima impressione dei comportamenti delle varie tecniche sulla base delle percentuali di copertura, è stata quindi effettuata una prima attività di **analisi quantitativa** tramite cui si sono fatte le prime supposizioni. Si è passato poi ad una **analisi quantitativa** più dettagliata della precedente, utilizzando tecniche di analisi statistica. Scendendo ancor di più nel dettaglio, attraverso attività di **analisi qualitativa**, è stato possibile estrarre ulteriori informazioni riguardo le differenze delle tecniche di testing riuscendo ad avere un quadro preciso della situazione utile per rispondere alle research question. Di seguito mostreremo nel dettaglio tutte le attività di analisi svolte.

5.6.1 **Risultati ottenuti**

In questo paragrafo sarà mostrata e discussa tutta la tabella in cui sono stati raccolti i dati relativi alle percentuali delle coperture delle App, per tutti gli studenti, utilizzando le tecniche di testing già discusse in precedenza. Per quanto concerne lo schema tabellare è stato utilizzato un prodotto di casa Microsoft chiamato Excel facente parte del pacchetto Office. Grazie all'utilizzo del foglio elettronico e delle funzioni di supporto, messe a disposizione da Excel stesso, è stato possibile organizzare i dati, effettuare piccole operazioni matematiche ed estrapolare

informazioni per trarre le prime conclusioni riguardo le differenze tra le tecniche:

App SimplyDo:

APP	TESTER	TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
SimplyDo	Ripper (Fase1)	700	526	919,9	1281	0,7181108509	72
	Apicella	4,7	90	906,3	1281	0,7074941452	71
	Ibrido(Apicella)	600	294	1070	1281	0,8352849336	84
	(Apicella)U(Sistematico)	-	-	1062,6	1281	0,8295081967	83
	Bosco	5,16	140	1071,1	1281	0,8361436378	84
	Ibrido (Bosco)	405	267	1077,1	1281	0,8408274785	84
	(Bosco)U(Sistematico)	-	-	1076,1	1281	0,8400468384	84
	Davino	2,2	53	960,5	1281	0,74980484	75
	Ibrido (DAvino)	320	177	1051	1281	0,8204527713	82
	(DAvino)U(Sistematico)	-	-	1048,8	1281	0,818735363	82
	Donnarumma	6,15	161	1048,7	1281	0,818657299	82
	Ibrido (Donnarumma)	1395	439	1080,1	1281	0,8431693989	84
	(Donnarumma)U(Sistematico)	-	-	1078,1	1281	0,8416081187	84
	Gargiulo	3,19	85	1023,8	1281	0,7992193599	80
	Ibrido (Gargiulo)	560	321	1075,4	1281	0,8395003903	84
	(Gargiulo)U(Sistematico)	-	-	1073	1281	0,837626854	84
	Gariuolo	6	128	974,7	1281	0,7608899297	76
	Ibrido (Gariuolo)	1440	478	1079,1	1281	0,8423887588	84
	(Gariuolo)U(Sistematico)	-	-	1079,1	1281	0,8423887588	84
	Ioviero	5,37	141	973,3	1281	0,7597970336	76
	Ibrido (Ioviero)	610	425	1079	1281	0,8423106948	84
	(Ioviero)U(Sistematico)	-	-	1039,7	1281	0,8116315379	81
	Lazzari	5,3	156	1045,1	1281	0,8158469945	82
	Ibrido (Lazzari)	1780	807	1079,8	1281	0,8429352069	84
	(Lazzari)U(Sistematico)	-	-	1071,4	1281	0,8363778298	84
	Marigliano	5,3	121	1047,2	1281	0,8174863388	82
	Ibrido (Marigliano)	645	553	1079	1281	0,8423106948	84
	(Marigliano)U(Sistematico)	-	-	1073,7	1281	0,8381733021	84
	Ragno	2,51	61	805,4	1281	0,6287275566	63
	Ibrido (Ragno)	285	228	967,7	1281	0,7554254489	76
	(Ragno)U(Sistematico)	-	-	967,6	1281	0,7553473849	76
	Russo	3,28	94	943	1281	0,7361436378	74
	Ibrido (Russo)	785	300	972,3	1281	0,7590163934	76
	(Russo)U(Sistematico)	-	-	972,3	1281	0,7590163934	76
	Saviano	4,44	111	1076,6	1281	0,8404371585	84
	Ibrido (Saviano)	820	222	1080,1	1281	0,8431693989	84
	(Saviano)U(Sistematico)	-	-	1078,6	1281	0,8419984387	84
	Toscano	8,1	65	956,8	1281	0,7469164715	75
	Ibrido(Toscano)	600	298	1077,1	1281	0,8408274785	84
	(Toscano)U(Sistematico)	-	-	1034,7	1281	0,8077283372	81

(Figura 5.4: Dati di copertura raccolti per l'App SimplyDo)

App MunchLife:

		TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
MunchLife	Ripper (Fase1)	20	26	135,7	184	0,7375	74
	Apicella	4,44	78	154,5	184	0,839673913	84
	Ibrido(Apicella)	90	83	164,9	184	0,8961956522	90
	(Apicella)U(Sistematico)	-	-	164,9	184	0,8961956522	90
	Bosco	2,3	61	165,8	184	0,9010869565	90
	Ibrido (Bosco)	65	66	172,8	184	0,9391304348	94
	(Bosco)U(Sistematico)	-	-	171,8	184	0,9336956522	93
	Davino	1,32	53	173	184	0,9402173913	94
	Ibrido (DAvino)	65	58	175	184	0,9510869565	95
	(DAvino)U(Sistematico)	-	-	175	184	0,9510869565	95
	Donnarumma	1,3	84	155,5	184	0,8451086957	85
	Ibrido (Donnarumma)	145	89	169,9	184	0,9233695652	92
	(Donnarumma)U(Sistematico)	-	-	169,9	184	0,9233695652	92
	Gargiulo	1,45	53	160,5	184	0,8722826087	87
	Ibrido (Gargiulo)	60	58	167,8	184	0,9119565217	91
	(Gargiulo)U(Sistematico)	-	-	167,8	184	0,9119565217	91
	Gariuolo	4,22	120	160,5	184	0,8722826087	87
	Ibrido (Gariuolo)	265	125	169,8	184	0,922826087	92
	(Gariuolo)U(Sistematico)	-	-	169,8	184	0,922826087	92
	Ioviero	5,2	295	156,5	184	0,8505434783	85
	Ibrido (Ioviero)	840	300	166,9	184	0,9070652174	91
	(Ioviero)U(Sistematico)	-	-	166,9	184	0,9070652174	91
	Lazzari	1,34	49	154,5	184	0,839673913	84
	Ibrido (Lazzari)	40	54	164,9	184	0,8961956522	90
	(Lazzari)U(Sistematico)	-	-	164,9	184	0,8961956522	90
	Marigliano	2,33	74	141,4	184	0,7684782609	77
	Ibrido (Marigliano)	160	74	149,8	184	0,8141304348	81
	(Marigliano)U(Sistematico)	-	-	149,8	184	0,8141304348	81
	Ragno	1,5	25	146,5	184	0,7961956522	80
	Ibrido (Ragno)	20	30	163,8	184	0,8902173913	89
	(Ragno)U(Sistematico)	-	-	163,8	184	0,8902173913	89
	Russo	2,4	58	165	184	0,8967391304	90
	Ibrido (Russo)	65	63	174,9	184	0,9505434783	95
	(Russo)U(Sistematico)	-	-	174,9	184	0,9505434783	95
	Saviano	3,58	183	175	184	0,9510869565	95
	Ibrido (Saviano)	450	188	177	184	0,9619565217	96
	(Saviano)U(Sistematico)	-	-	177	184	0,9619565217	96
	Toscano	3,45	59	150,5	184	0,8179347826	82
	Ibrido(Toscano)	60	64	165,8	184	0,9010869565	90
	(Toscano)U(Sistematico)	-	-	165,8	184	0,9010869565	90

(Figura 5.5: Dati di copertura raccolti per l'App MunchLife)

App TippyTipper:

		TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
TippyTipper	Ripper (Fase1)	130	86	719,4	999	0,7201201201	72
	Apicella	5,26	130	832,7	999	0,8335335335	83
	Ibrido(Apicella)	400	130	837,7	999	0,8385385385	84
	(Apicella)U(Sistematico)	-	-	837,7	999	0,8385385385	84
	Bosco	4,5	156	863,7	999	0,8645645646	86
	Ibrido (Bosco)	530	156	865,7	999	0,8665665666	87
	(Bosco)U(Sistematico)	-	-	865,7	999	0,8665665666	87
	Davino	2,31	93	854,2	999	0,8550550551	86
	Ibrido (DAvino)	125	93	860,4	999	0,8612612613	86
	(DAvino)U(Sistematico)	-	-	860,4	999	0,8612612613	86
	Donnarumma	3,28	109	866,7	999	0,8675675676	87
	Ibrido (Donnarumma)	310	109	873,7	999	0,8745745746	87
	(Donnarumma)U(Sistematico)	-	-	873,7	999	0,8745745746	87
	Gargiulo	2,44	83	791,9	999	0,7926926927	79
	Ibrido (Gargiulo)	120	83	856,4	999	0,8572572573	86
	(Gargiulo)U(Sistematico)	-	-	856,4	999	0,8572572573	86
	Gariuolo	6,41	161	866,7	999	0,8675675676	87
	Ibrido (Gariuolo)	550	161	869,7	999	0,8705705706	87
	(Gariuolo)U(Sistematico)	-	-	869,7	999	0,8705705706	87
	Ioviero	5,23	115	855,4	999	0,8562562563	86
	Ibrido (Ioviero)	210	115	863,7	999	0,8645645646	86
	(Ioviero)U(Sistematico)	-	-	863,7	999	0,8645645646	86
	Lazzari	4,11	170	813,4	999	0,8142142142	81
	Ibrido (Lazzari)	370	170	864,7	999	0,8655655656	87
	(Lazzari)U(Sistematico)	-	-	864,7	999	0,8655655656	87
	Marigliano	3	85	853,4	999	0,8542542543	85
	Ibrido (Marigliano)	125	85	868,7	999	0,8695695696	87
	(Marigliano)U(Sistematico)	-	-	868,7	999	0,8695695696	87
	Ragno	2,38	83	844,7	999	0,8455455455	85
	Ibrido (Ragno)	115	83	856,7	999	0,8575575576	86
	(Ragno)U(Sistematico)	-	-	856,7	999	0,8575575576	86
	Russo	3,9	95	837,7	999	0,8385385385	84
	Ibrido (Russo)	165	95	858,7	999	0,8595595596	86
	(Russo)U(Sistematico)	-	-	858,7	999	0,8595595596	86
	Saviano	2,52	115	868,7	999	0,8695695696	87
	Ibrido (Saviano)	235	115	873,7	999	0,8745745746	87
	(Saviano)U(Sistematico)	-	-	873,7	999	0,8745745746	87
	Toscano	3,14	74	832,9	999	0,8337337337	83
	Ibrido(Toscano)	125	74	851,4	999	0,8522522523	85
	(Toscano)U(Sistematico)	-	-	851,4	999	0,8522522523	85

(Figura 5.6: Dati di copertura raccolti per l'App TippyTipper)

App Trolly:

		TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
Trolly	Ripper (Fase1)	45	67	232,5	364	0,6387362637	64
	Apicella	4,42	102	281,2	364	0,7725274725	77
	Ibrido(Apicella)	260	118	281,2	364	0,7725274725	77
	(Apicella)U(Sistematico)	-	-	281,2	364	0,7725274725	77
	Bosco	3,7	91	276,4	364	0,7593406593	76
	Ibrido (Bosco)	185	107	276,4	364	0,7593406593	76
	(Bosco)U(Sistematico)	-	-	276,4	364	0,7593406593	76
	Davino	1,58	29	276,4	364	0,7593406593	76
	Ibrido (DAvino)	30	29	291,4	364	0,8005494505	80
	(DAvino)U(Sistematico)	-	-	291,4	364	0,8005494505	80
	Donnarumma	3,38	89	289,4	364	0,7950549451	80
	Ibrido (Donnarumma)	160	105	289,4	364	0,7950549451	80
	(Donnarumma)U(Sistematico)	-	-	289,4	364	0,7950549451	80
	Gargiulo	2,27	67	276,3	364	0,7590659341	76
	Ibrido (Gargiulo)	120	83	276,3	364	0,7590659341	76
	(Gargiulo)U(Sistematico)	-	-	276,3	364	0,7590659341	76
	Gariuolo	4,9	87	291,3	364	0,8002747253	80
	Ibrido (Gariuolo)	200	103	291,3	364	0,8002747253	80
	(Gariuolo)U(Sistematico)	-	-	291,3	364	0,8002747253	80
	Ioviero	3,5	91	272,3	364	0,7480769231	75
	Ibrido (Ioviero)	215	107	272,3	364	0,7480769231	75
	(Ioviero)U(Sistematico)	-	-	272,3	364	0,7480769231	75
	Lazzari	7	200	284,3	364	0,781043956	78
	Ibrido (Lazzari)	670	216	285,3	364	0,7837912088	78
	(Lazzari)U(Sistematico)	-	-	285,3	364	0,7837912088	78
	Marigliano	3,12	90	274,3	364	0,7535714286	75
	Ibrido (Marigliano)	160	106	275,3	364	0,7563186813	76
	(Marigliano)U(Sistematico)	-	-	275,3	364	0,7563186813	76
	Ragno	1,57	56	274,3	364	0,7535714286	75
	Ibrido (Ragno)	70	72	274,3	364	0,7535714286	75
	(Ragno)U(Sistematico)	-	-	274,3	364	0,7535714286	75
	Russo	3,47	109	283,3	364	0,7782967033	78
	Ibrido (Russo)	260	115	286,3	364	0,7865384615	79
	(Russo)U(Sistematico)	-	-	286,3	364	0,7865384615	79
	Saviano	3,28	95	296,3	364	0,814010989	81
	Ibrido (Saviano)	240	111	296,3	364	0,814010989	81
	(Saviano)U(Sistematico)	-	-	296,3	364	0,814010989	81
	Toscano	5,2	75	280,7	364	0,7711538462	77
	Ibrido(Toscano)	260	91	281,7	364	0,7739010989	77
	(Toscano)U(Sistematico)	-	-	281,7	364	0,7739010989	77

(Figura 5.7: Dati di copertura raccolti per l'App Trolly)

App FillUp:

		TEMPO (min)	EVENTI	ELOC	TLOC	ELOC/TLOC	% COV
FillUp	Ripper (Fase1)	60	65	482,1	3807	0,1266351458	13
	Apicella	7,32	197	3158,2	3807	0,8295770948	83
	Ibrido(Apicella)	2210	454	3294,5	3807	0,865379564	87
	(Apicella)U(Sistematico)	-	-	3192,2	3807	0,8385080116	84
	Bosco	12,51	230	2950,3	3807	0,7749671657	77
	Ibrido (Bosco)	1980	466	3083,4	3807	0,809929078	81
	(Bosco)U(Sistematico)	-	-	2972,3	3807	0,7807459942	78
	Davino	4,21	137	3130,7	3807	0,8223535592	82
	Ibrido (D'Avino)	940	380	3260,2	3807	0,856369845	86
	(D'Avino)U(Sistematico)	-	-	3141,7	3807	0,8252429735	83
	Donnarumma	8,34	258	3201,5	3807	0,84095088	84
	Ibrido (Donnarumma)	1500	493	3264,3	3807	0,8574468085	86
	(Donnarumma)U(Sistematico)	-	-	3213,5	3807	0,8441029682	84
	Gargiulo	6,33	198	3027,7	3807	0,795298135	80
	Ibrido (Gargiulo)	1185	450	3235,7	3807	0,8499343315	85
	(Gargiulo)U(Sistematico)	-	-	3049,7	3807	0,8010769635	80
	Gariuolo	16,42	162	2758,9	3807	0,724691358	72
	Ibrido (Gariuolo)	2250	364	3048,7	3807	0,8008142895	80
	(Gariuolo)U(Sistematico)	-	-	2800,9	3807	0,7357236669	74
	Ioviero	13,31	132	2715,6	3807	0,7133175729	71
	Ibrido (Ioviero)	1375	381	3071,4	3807	0,8067769898	81
	(Ioviero)U(Sistematico)	-	-	2773,6	3807	0,7285526661	73
	Lazzari	9,8	250	3006,2	3807	0,7896506436	79
	Ibrido (Lazzari)	1990	495	3210,4	3807	0,8432886787	84
	(Lazzari)U(Sistematico)	-	-	3027,2	3807	0,795166798	80
	Marigliano	9,6	226	2968,8	3807	0,7798266351	78
	Ibrido (Marigliano)	1410	435	3014,3	3807	0,7917783031	79
	(Marigliano)U(Sistematico)	-	-	2989,8	3807	0,7853427896	79
	Ragno	4,22	226	2968,8	3807	0,7798266351	78
	Ibrido (Ragno)	1420	432	3013,3	3807	0,7915156291	79
	(Ragno)U(Sistematico)	-	-	2989,8	3807	0,7853427896	79
	Russo	7,53	195	2851,9	3807	0,749120042	75
	Ibrido (Russo)	2070	410	3063,8	3807	0,8047806672	80
	(Russo)U(Sistematico)	-	-	2869,9	3807	0,7538481744	75
	Saviano	9,17	292	3186,1	3807	0,8369057	84
	Ibrido (Saviano)	2640	532	3316	3807	0,8710270554	87
	(Saviano)U(Sistematico)	-	-	3208,1	3807	0,8426845285	84
	Toscano	5,4	137	3119,2	3807	0,819332808	82
	Ibrido(Toscano)	1285	361	3239,1	3807	0,8508274232	85
	(Toscano)U(Sistematico)	-	-	3157,2	3807	0,8293144208	83

(Figura 5.8: Dati di copertura raccolti per l'App FillUp)

Diamo ora una descrizione precisa degli elementi che compongono la tabella in modo da poterla interpretare nel modo corretto, partendo da sinistra, per le colonne:

- **Nome dell'AUT**
- **Tecnica utilizzata**
- **Misura del tempo (in minuti)**
- **Eventi:** per la tecnica sistematica, equivale al numero di file di coverage generati a fine fase 1. Per la tecnica manuale, equivale al numero di istruzioni, generate da Robotium Recorder, caratterizzanti il codice di test (è bastato prendere il numero di tagli effettuati dal tool di conversione). Per la tecnica ibrida, equivale al numero di file di coverage generati alla fine della fase 2.
- **ELOC:** Executed Lines Of Code = linee di codice eseguite
- **TLOC:** Total Lines Of Code = linee di codice totali
- **ELOC/TLOC:** misura dell'efficacia
- **%COV:** misura dell'efficacia espressa in percentuale

Per le righe:

- **Ripper (Fase 1):** riga dedicata alla tecnica sistematica
- **Cognome studente:** riga dedicata alla tecnica manuale
- **Ibrido(cognome_studente):** riga dedicata alla tecnica ibrida
- **(cognome_studente)U(sistematico):** riga dedicata all'unione

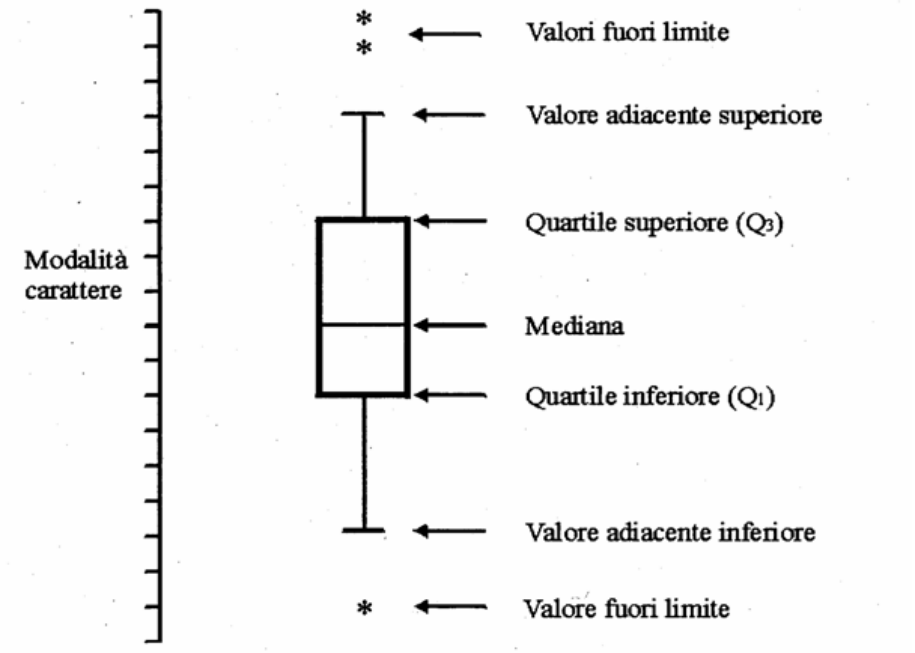
precisiamo che i valori della tecnica sistematica, denominata come Ripper fase1, vengono riportati una sola volta per App in quanto quel valore resta costante a prescindere dalle altre tecniche di testing, invece. Osservando e paragonando i valori di efficacia è possibile asserire che è sempre utile far ricorso al Ripper Ibrido in

quanto presenta una percentuale di copertura sempre maggiore rispetto alle altre tecniche, almeno rispetto a quella sistematica. Questi risultati hanno messo in luce un'altra importante caratteristica del ripper ibrido ovvero, nel caso in cui non vengono scoperti nuovi stati, l'efficacia del ripper ibrido è di valore pari al valore massimo tra l'efficacia ottenuta con la tecnica sistematica e quella ottenuta con la tecnica manuale ma, dato che non è possibile sapere a priori se il ripper ibrido trova o meno nuovi stati, è comunque necessario eseguire il ripper ibrido almeno per due o tre campioni. Inoltre, confrontando i risultati ottenuti dalla tecnica ibrida e l'unione, notiamo che spesso i loro valori di efficacia risultano essere uguali o molto vicini ma, sempre per ragioni legate al valore massimo di efficacia, si consiglia di usare il ripper ibrido.

5.6.2 **Analisi quantitativa e statistica**

Ciò che andremo a presentare nei prossimi paragrafi è la descrizione dell'analisi quantitativa utilizzando alcune soluzioni proposte dall'analisi statistica con lo scopo di ottenere informazioni aggiuntive rispetto a quanto detto nel paragrafo precedente. Per evidenziare alcune caratteristiche, appartenenti alle tabelle già presentate, si è fatto uso dei diagrammi Box-and-Whisker (scatola e baffi), più comunemente conosciuti come **box-plot**. La scelta è caduta su questo tipo di diagrammi in quanto permettono di evidenziare, graficamente, l'andamento delle nostre distribuzioni riassumendo le informazioni ricavate attraverso cinque valori contenuti nella distribuzione: la mediana, il primo ed il terzo quartile, il valore minimo e massimo. Un diagramma di questo tipo serve per rappresentare visivamente quattro caratteristiche fondamentali di una distribuzione statistica di dati campionari: la misura di tendenza centrale, attraverso la mediana e/o la media; il grado di dispersione o variabilità dei dati, rispetto alla mediana e/o alla media; la forma della distribuzione dei dati, in particolare la simmetria; la presenza di ogni valore anomalo o outlier. Volendo descrivere alcune caratteristiche di un box-plot, si osserva una linea orizzontale, interna alla "scatola", che rappresenta la mediana, ossia il valore centrale in un insieme ordinato di dati, una

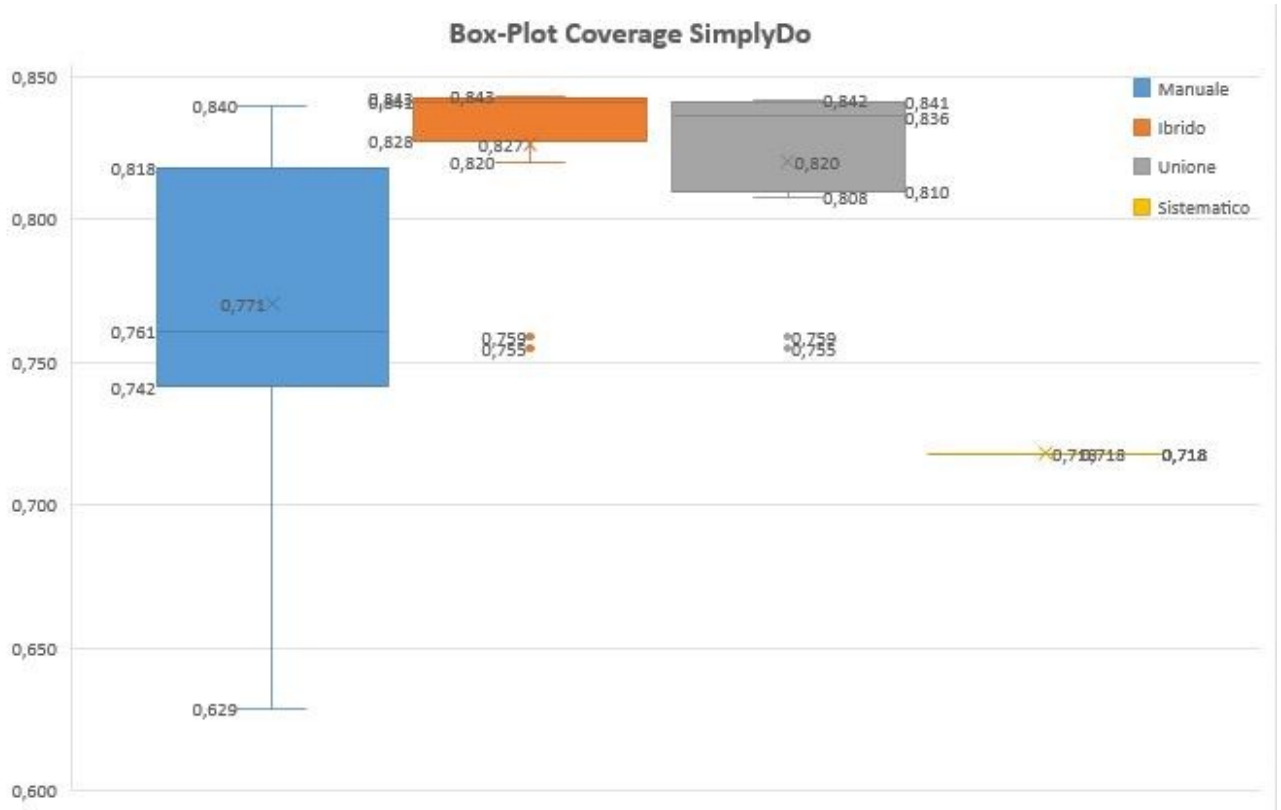
misura robusta, in quanto poco influenzata dalla presenza di dati anomali. La scatola (box) è delimitata da due linee orizzontali: la linea inferiore, indicata con Q1, che rappresenta il primo quartile; la linea superiore, indicata con Q3, che rappresenta il terzo quartile. La distanza tra il terzo (Q3) e il primo quartile (Q1), detta distanza interquartilica (interquartile range o IQR), è una misura della dispersione della distribuzione. Un intervallo interquartilico piccolo indica che la metà delle osservazioni ha valori molto vicini alla mediana. L'intervallo aumenta al crescere della dispersione (varianza) dei dati. I valori esterni a questi limiti sono definiti valori anomali (outliers). Nella rappresentazione grafica del box-plot, gli outliers sono segnalati individualmente, poiché costituiscono un'anomalia importante rispetto agli altri dati della distribuzione e nella statistica parametrica il loro peso sulla determinazione quantitativa dei parametri è molto grande. Se la distribuzione è normale, nel box-plot le distanze tra ciascun quartile e la mediana saranno uguali e avranno lunghezza uguale le due linee che partono dai bordi della scatola e terminano con i baffi. Anche in questo caso, il software di casa Microsoft, Excel è stato utile per "disegnare" questi diagrammi, da precisare che è possibile utilizzare i box-plot solo a partire dalla versione 2016.



(Figura 5.9: Descrizione di un Box-Plot)

5.6.3 **Box-Plot**

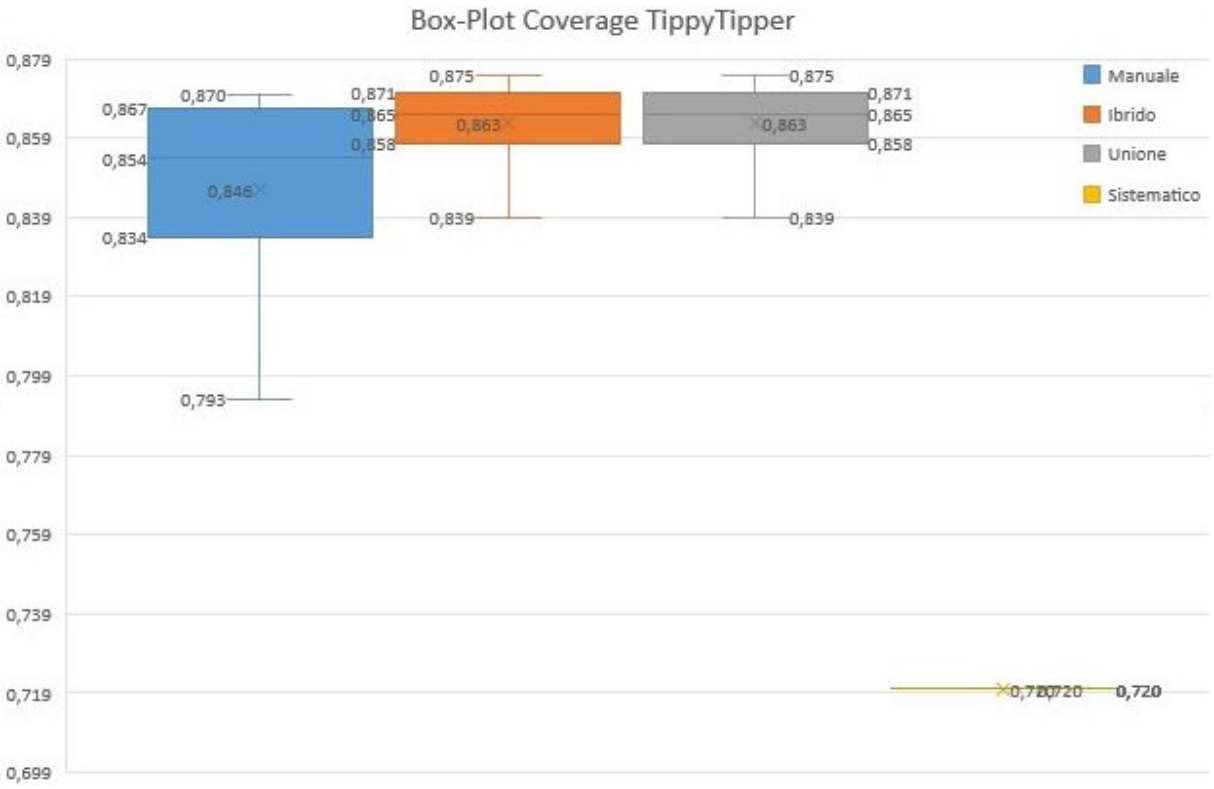
Andremo ora a riportare i grafici statistici relativi ai Box-Plot delle coperture ottenute grazie alle varie tecniche proposte. Mostreremo, tecnica per tecnica, un'unica area in cui è possibile mettere tutti i Box-Plot a confronto:



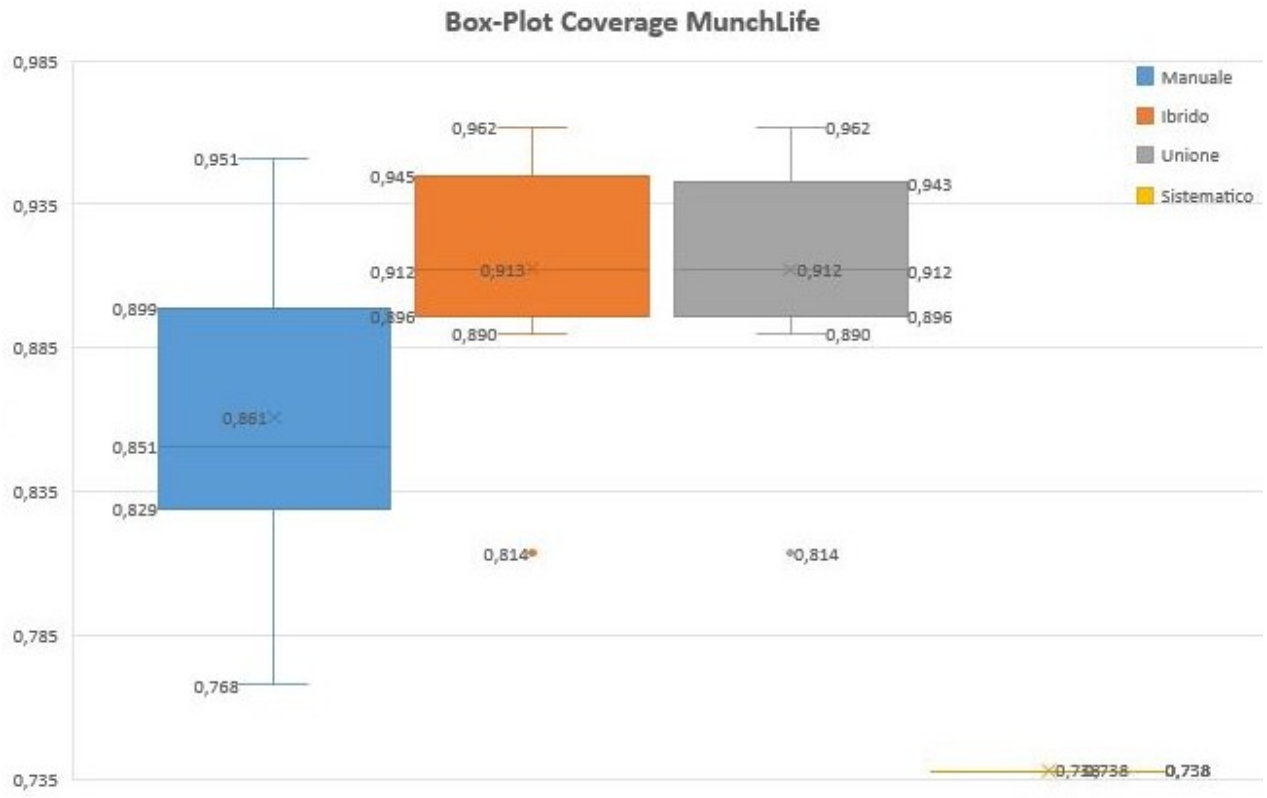
(Figura 5.10)

Precisiamo che nel Box-Plot presentato è presente una "x" la quale sta a rappresentare il valore della media della distribuzione. Basandoci su quest'ultima figura, è possibile capire graficamente come, grazie alla tecnica ibrida, sia possibile ottenere un livello di efficacia abbastanza alto rispetto alle altre tecniche, ma per rendere coerente questa affermazione c'è bisogno di eseguire questi confronti anche sulle altre App. Specifichiamo che, la linea gialla a destra, ovvero il plot relativo alla tecnica sistematica, rappresenta un punto, in quanto esiste un solo valore di copertura dato che, a prescindere dal numero di studenti che hanno effettuato le loro prove, questa tecnica, eseguita sulla stessa App più volte, restituisce sempre lo

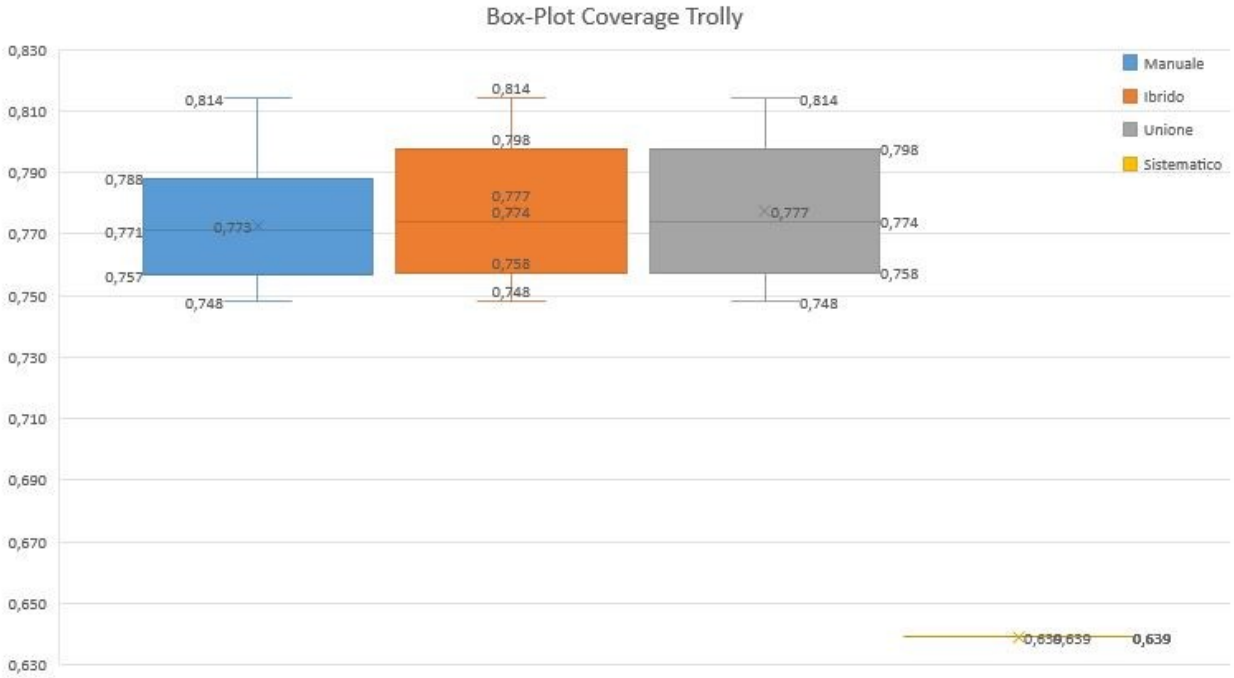
stesso valore, risulta quindi variabile solo al variare dell'AUT, pertanto, data la sua natura deterministica, non si potrebbe effettuare il Box-Plot di un punto ma, in questo caso, risulta necessario ai fini dei confronti. Ritornando al discorso, affrontato nel paragrafo 5.6.1, riguardo la scelta tra la tecnica ibrida e l'unione (sistematica)U(manuale), tramite questo Box-Plot si evince quanto già affermato in precedenza, ovvero che le due attività di testing raggiungono livelli di efficienza vicini ma, confrontando i quartili inferiori (Q1) e superiori (Q3) dei due plot, specie quelli inferiori, per questo caso in esame, con la tecnica ibrida si raggiungono percentuali di copertura maggiori su più campioni rispetto all'unione. Si procede, quindi, con la presentazione dei Box-Plot delle altre quattro AUT:



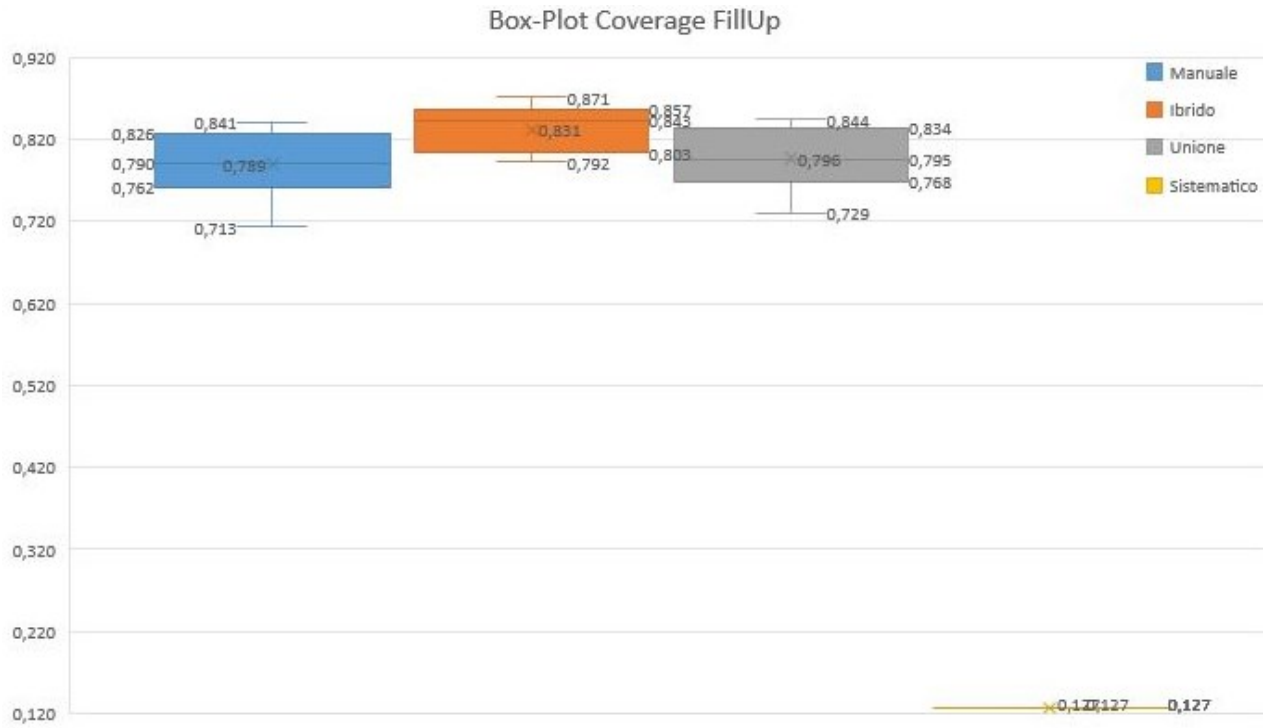
(Figura 5.11)



(Figura 5.12)



(Figura 5.13)



(Figura 5.14)

Attraverso il confronto dei Box-Plot presentati, la tecnica preponderante risulta essere ancora quella ibrida ma, grazie alla componente grafica, è molto più chiaro il grado di similitudine tra la tecnica ibrida e l'unione (sistematica)U(manuale) in quanto, grazie al confronto dei quartili inferiori e superiori, su due App ci sono differenze evidenti: osservando le figure (Figura 5.14: Box-Plot Coverage SimplyDo) e (Figura 5.34: Box-Plot Coverage FillUp) è possibile notare che, nella figura 5.14, la tecnica ibrida ha il quartile inferiore (Q1) ed il valore minimo maggiori rispetto a quelli dell'unione, mentre nella figura 5.34 abbiamo i quartili (Q1) e (Q3), i valori di massimo e minimo, media, mediana, della tecnica ibrida, maggiori rispetto a quelli ottenibili con l'unione, anche se numericamente vicini, e quindi, come già detto in precedenza, la tecnica ibrida raggiunge livelli di efficacia alti su più campioni rispetto all'unione. Considerando invece tutte e cinque le App, risulta evidente il motivo per cui è necessario affrontare un'attività di analisi qualitativa in quanto, i Box-Plot relativi alla tecnica ibrida ed all'unione, per alcune App sono uguali mentre per altre hanno valori numerici molto vicini, per cui è

necessario andare a controllare le coperture, ottenute con EMMA, a livello di codice in modo da stabilire se quelle piccole differenze possono essere considerate trascurabili o meno.

5.6.4 **Analisi qualitativa**

In questa attività di analisi è importante accedere al codice sorgente, a valle di tutte le attività di test, in modo tale da ottenere maggiori informazioni mettendo a confronto le coperture delle classi e dei metodi che compongono i test case. Grazie ad EMMA è possibile effettuare i confronti richiesti in quanto, durante la sua attività di analisi della copertura, evidenzia in rosso e verde il codice indicando, con il colore rosso, le porzioni di codice non coperte/attraversate dal test, mentre evidenzia con il colore verde le porzioni di codice coperte. Riproponendo la (Figura 2.5):

```
161     @Override
162     public void fetchLists()
163     {
164         Log.v(L.TAG, "CachingDataView.fetchLists(): Entered");
165         ViewerTask task = new ViewerTask();
166         task.taskId = ViewerTask.FETCH_LISTS;
167         doTaskAndWait(task);
168         Log.v(L.TAG, "CachingDataView.fetchLists(): Exited");
169     }
170
171     public ListDesc fetchList(int listId)
172     {
173         ListDesc rv = null;
174
175         synchronized (viewerLock)
176         {
177             for(ListDesc list : listData)
178             {
179                 if(listId == list.getId())
180                 {
181                     rv = list;
182                     break;
183                 }
```

è facile intuire la potenza di quest'attività in quanto è possibile vedere, riga per riga, le differenze di copertura esistenti tra le esecuzioni delle tecniche di testing messe a confronto. Questa attività di analisi ci consente quindi di capire se, ciò che differenzia, in minima parte, le coperture ottenute dalla tecnica ibrida e quella

ottenuta dall'unione, può essere considerato fuorviante o meno. Gli esiti ottenuti da quest'attività di analisi hanno messo in evidenza le reali differenze esistenti tra le tecniche prese in esame. Per quanto concerne le App "Trolly", "TippyTipper" e "MunchLife", è stato possibile notare come le due tecniche hanno prodotto lo stesso risultato, in termini di copertura ed efficacia, per tutti gli studenti, a meno di uno studente il quale, per l'App "MunchLife", ha eseguito una sola operazione in meno rispetto all'esecuzione dell'App con la tecnica Ibrida:

```
154         // Set gender up
155         WindowManager winManager = (WindowManager) getSystemService(WINDOW_SE
156         devDisplay = winManager.getDefaultDisplay();
157
158         if(devDisplay.getRotation() == 0) {
159             if(gender_female) {
160                 gender.setImageResource(R.drawable.female_portrait);
161             } else {
162                 gender.setImageResource(R.drawable.male_portrait);
163             }
164         } else {
165             if(gender_female) {
166                 gender.setImageResource(R.drawable.female_landscape);
167             } else {
168                 gender.setImageResource(R.drawable.male_landscape);
169             }
170         }
171         }
```

(Figura 5.15: Ramo IF non soddisfatto dal tester umano)

```
154         // Set gender up
155         WindowManager winManager = (WindowManager) getSystemService(WINDOW_SE
156         devDisplay = winManager.getDefaultDisplay();
157
158         if(devDisplay.getRotation() == 0) {
159             if(gender_female) {
160                 gender.setImageResource(R.drawable.female_portrait);
161             } else {
162                 gender.setImageResource(R.drawable.male_portrait);
163             }
164         } else {
165             if(gender_female) {
166                 gender.setImageResource(R.drawable.female_landscape);
167             } else {
168                 gender.setImageResource(R.drawable.male_landscape);
169             }
170         }
171         }
```

(Figura 5.16: Ramo IF soddisfatto dal testing ibrido)

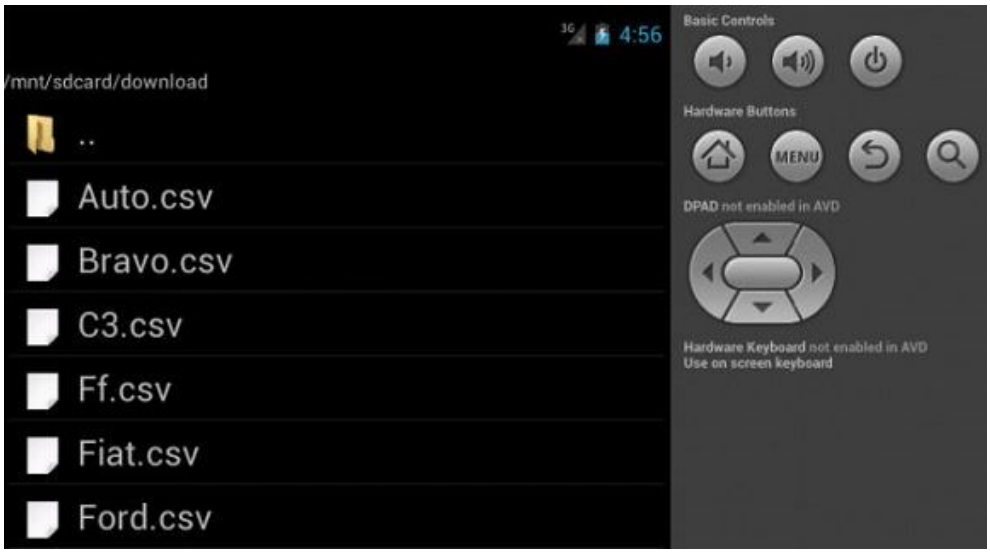
Le due figure appena presentate mostrano la differenza di copertura del codice sopracitata. In soldoni, la tecnica ibrida è riuscita a scoprire un nuovo stato in cui bisogna cambiare l'icona rappresentativa del sesso femminile se avviene una rotazione dello schermo del dispositivo nel caso in cui, precedentemente, l'utente ha già settato il parametro "sesso" con il simbolo femminile. Questo esempio ci induce a capire due cose, la prima è che grazie alla tecnica ibrida è più facile accedere a codice "nascosto", e percorribile, o comunque di difficile intuizione, mentre, la seconda, quanto sia indispensabile effettuare questo tipo di analisi in modo da comprendere la reale efficacia delle tecniche di testing e riuscire così a trarre conclusioni precise. Per quanto riguarda le altre due applicazioni, "SimplyDo" e "FillUp", le differenze tra le due tecniche hanno avuto più spessore. Grazie ad entrambe le App, a fine analisi, la tecnica ibrida si è portata in netto vantaggio, in termini di efficacia, dato che, mettendo a confronto le coperture, si sono presentate le classiche problematiche dovute alla distrazione, inesperienza, poca intuitività dei tester umani. La tecnica ibrida cerca di coprire tutti i rami della logica condizionale **IF-THEN**, così come gli **SWITCH-CASE**, questo perché cerca di soddisfare tutte le condizioni contenute in essi, condizione non assicurata se a testare l'App è un umano, infatti, con le figure 5.15 e 5.16, si può trarre la seguente conclusione:

dato che il tester umano non è stato in grado di coprire la porzione di codice interessata, l'unione ha ereditato questa perdita di copertura e, siccome l'unione è composta, oltre che dalla tecnica manuale, anche da quella sistematica, questo vuole dire che la perdita di copertura è pervenuta in egual modo in quest'ultima tecnica, per cui, a meno della tecnica sistematica, la tecnica dell'unione, rispetto a quella ibrida, è fortemente dipendente dall'esperienza del tester umano. Questa caratteristica ha una rilevanza notevole in quanto anche un singolo ramo della logica condizionale può dare accesso a tante altre operazioni, servizi, opzioni messe a disposizione dall'App.

Un altro caso interessante da esaminare è quello in cui diversi tester umani non

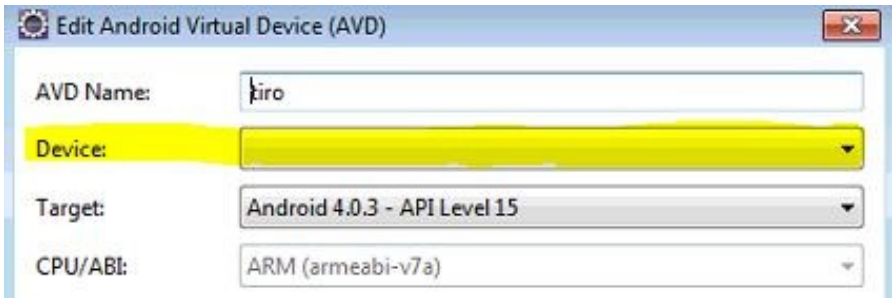
hanno eseguito, a differenza del testing Ibrido, attività di Import/Export e Backup dei dati le quali hanno portato a non aggiornare i file di Log annessi atti a ripristinare uno stato precedentemente salvato, attività, invece, eseguite dalla tecnica ibrida (i file di Log potrebbero essere utili ad un analista per estrapolare risultati adatti a studiare l'App).

Prendiamo come esempio le attività di Import/Export dell'App "FillUp":



(Figura 5.17: Selezione, dalla SDcard, del file di Backup da importare)

Per quanto riguarda l'attività di Export, una volta cliccato l'apposito bottone, il salvataggio sulla SDcard avviene in automatico, mentre, per l'Import, bisogna selezionare il file (.cvs) desiderato come mostrato in figura. Il problema per cui più tester umani non sono riusciti a coprire le porzioni di codice relative alle attività di Import/Export sta nel fatto che, sempre per distrazione, prima di avviare il dispositivo virtuale dall'AVD, per le attività di Capture and Replay, non hanno selezionato il tipo specifico di dispositivo da virtualizzare:



(Figura 5.18: Edit di configurazione di un dispositivo virtuale)

Lasciando il parametro "Device" vuoto si avvia un dispositivo virtuale di base privo della disponibilità hardware (virtuale) della SDcard, questo comporta dei problemi all'atto della richiesta delle attività di Import/Export:

(Figura 5.19)

Con la seguente figura è possibile osservare un comportamento particolare dell'App provando ad effettuare le attività di Import (a destra) ed Export (a sinistra) nel caso

in cui non è presente una SDcard, pertanto, in assenza di essa, ci viene informato che l'eventuale dispositivo di memorizzazione è non scrivibile/leggibile. Anche in questo caso, come già specificato, la distrazione e/o inesperienza dei tester umani ha limitato l'uso del codice dell'App.

```
public static File getPublicDownloadDirectory() {

        // get path to external storage directory
        File dir = Environment.getExternalStoragePublicDirectory(Context.DOWNLOAD_SERVICE);

        // make sure it exists
        dir.mkdir();

        return dir;
}
```

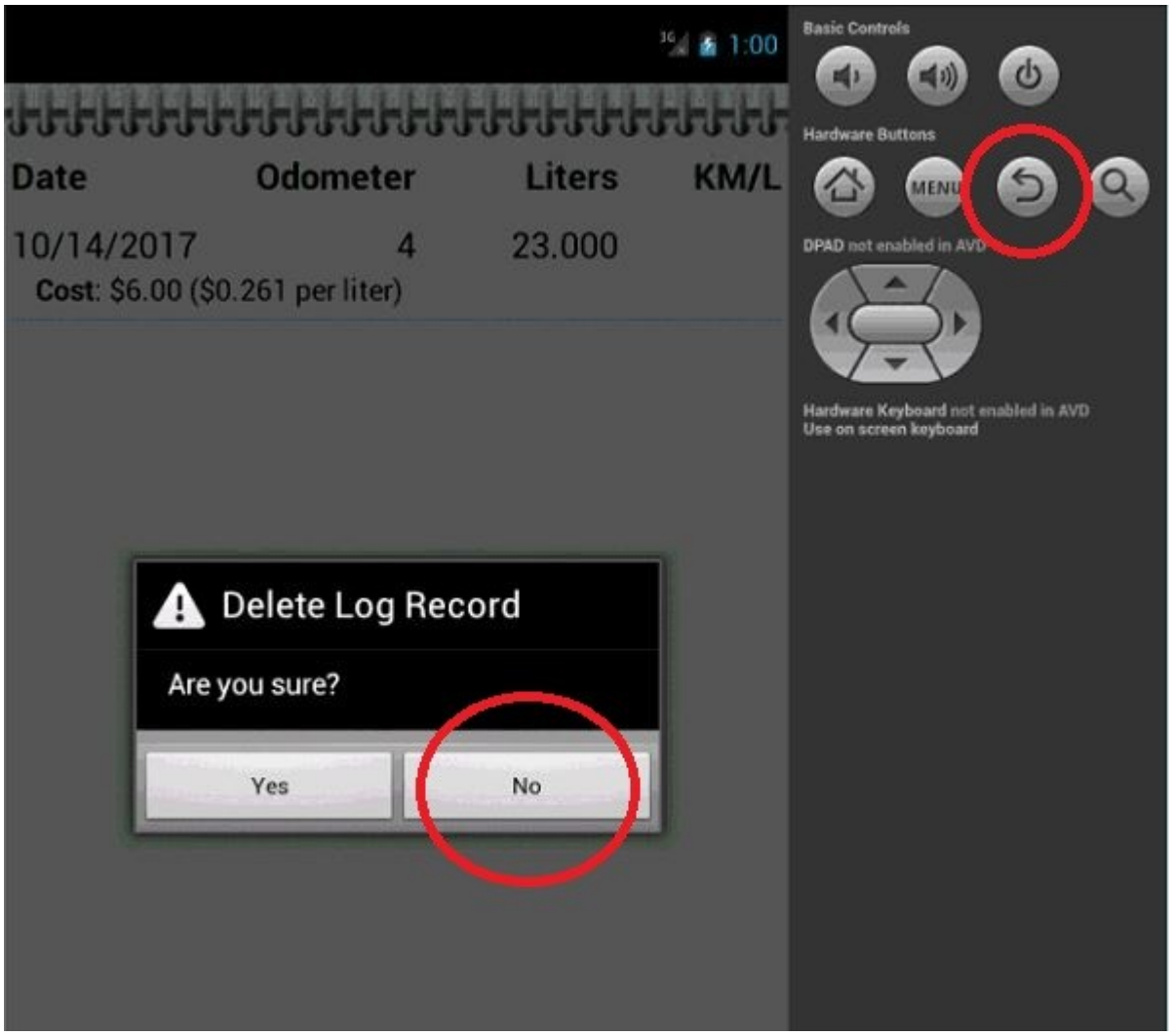
(Figura 5.20: Esempio di insuccesso di copertura con la tecnica dell'unione)

```
public static File getPublicDownloadDirectory() {

        // get path to external storage directory
        File dir = Environment.getExternalStoragePublicDirectory(Context.DOWNLOAD_SERVICE);

        // make sure it exists
        dir.mkdir();

        return dir;
}
```

(Figura 5.21: Esempio di successo di copertura con la tecnica ibrida)

Questo esempio di copertura è strettamente legato all'esempio precedente in quanto le attività di Import/Export comportano la creazione, nel caso in cui non esistesse, della directory ".../download" (metodo **mkdir()** utilizzato in quanto se la cartella già esiste allora non accade nulla e l'esecuzione procede) utilizzata per memorizzare o recuperare i file di backup. Risulta evidente che, nei casi in cui è stato utilizzato il dispositivo virtuale di base, non avendo a disposizione l'accesso alla SDcard, in quanto assente, la porzione di codice mostrata in figura non viene mai attraversata. Ancora una volta è fondamentale sottolineare come una semplice distrazione del tester umano possa compromettere l'intera attività di testing tenendo, ad esempio, centinaia di righe di codice nascoste.

Un altro caso ancora, che prova la superiorità della tecnica ibrida, è quello in cui si è riscontrata la presenza di alcuni listener nell'App "FillUp", quelli più comuni, non soddisfatti dalla maggior parte degli studenti ma soddisfatti in ogni caso dalla tecnica ibrida, sono quei listener che restano attivi, all'apertura di alcuni "dialog", in attesa della pressione del bottone "CANCEL/NO" o del tasto "BACK" sulla tastiera virtualizzata (vedere Figura 5.22). Questa situazione, come quella precedente, evidenzia il fatto che con la tecnica ibrida è più facile coprire il codice sotto test e quindi accedere a più porzioni di codice (questo perché il listener, una volta soddisfatto, potrebbe attivare qualche tipo di evento) e quindi ottenere un livello di efficacia maggiore.



(Figura 5.22)

```
// require user selection
Dialog dialog = builder.create();
dialog.setCanceledOnTouchOutside(false);

dialog.setOnCancelListener(new DialogInterface.OnCancelListener() {
                @Override
                public void onCancel(DialogInterface dialog) {
        listener.onStorageSelectionDialogResponse(id,Result.RESULT_CANCEL,null);
                }
        });
```

(Figura 5.23: esempio di listener non soddisfatto dal tester umano)

```
// require user selection
Dialog dialog = builder.create();
dialog.setCanceledOnTouchOutside(false);

dialog.setOnCancelListener(new DialogInterface.OnCancelListener() {
                @Override
                public void onCancel(DialogInterface dialog) {
         listener.onUnitsDialogResponse(id,Result.RESULT_CANCEL,null);
                }
         });
```

(Figura 5.24: esempio di listener soddisfatto dal testing ibrido)

5.7 Risultati complessivi delle analisi

Dalle analisi che hanno riguardato questo capitolo di sperimentazione si sono potuti riscontrare degli aspetti molto interessanti in merito alle diverse tecniche di generazione dei casi di test. Giunti a questo punto, dopo aver esaminato i risultati ottenuti dai diversi confronti, è possibile rispondere alle Research Question:

RQ 1) Il Ripper Ibrido consente di migliorare il livello di efficacia rispetto a quello ottenuto dall'utilizzo della sola Tecnica Manuale?

RS 1) Dai risultati ottenuti sia grazie all'analisi quantitativa che quella qualitativa si può affermare che la tecnica Ibrida ha permesso di migliorare l'efficacia dei test eseguiti sulle applicazioni rispetto alla sola tecnica Manuale. La risposta a questa domanda è giustificata dal fatto che la tecnica Ibrida, oltre ad includere la componente dei test eseguiti manualmente, riesce ad aumentare i risultati di copertura perché coadiuvata dalla componente sistematica. Quest'ultima, anche se in

piccola parte, risulta importante per raggiungere porzioni di codice in cui i soli test case, per diversi motivi, non riescono a spingersi. Questi motivi, evidenziati durante l'analisi, comprendono condizioni di scelta tra rami IF-ELSE (non sempre), o condizioni che riguardano la scelta di codice contenente SWITCH-CASE (non sempre), o ancora a mancate coperture relative alla pressione di pulsanti presenti nelle schermate e/o non ben visibili dagli utenti durante l'esplorazione dell'applicazione.

RQ 2) Il Ripper Ibrido consente di migliorare il livello di efficacia rispetto a quello ottenuto dall'utilizzo della sola tecnica sistematica (ML)?

RS 2) Come per la precedente domanda, dalle valutazioni effettuate si evince che la tecnica Ibrida consente di ottenere un aumento molto significativo di efficacia rispetto alla tecnica ML. Il motivo più evidente è da ricercarsi nel fatto che la tecnica Ibrida includa la componente manuale che produce molti eventi più significativi sull'applicazione. Inoltre l'adozione della tecnica ibrida permette di avere oltre che una copertura più elevata, derivante dai test case utente, anche una più completa esplorazione di tipologie di codice non raggiunte nei test manuali perché si avvale di un'altra componente sistematica scatenata dopo la scoperta di nuovi stati.

RQ 3) Il Testing Manuale risulta più efficace rispetto alla tecnica ML?

RS 3) Osservando i dati, la risposta a questa domanda sembra scontata se si guarda alle sole percentuali di copertura; infatti, la tecnica Manuale migliora notevolmente l'efficacia dei test rispetto alla tecnica ML; tuttavia si è anche osservato che molte tipologie di codice come alcune condizioni IF-ELSE, non vengono raggiunte dalla tecnica Manuale. In più occasioni la componente manuale ha coperto una sola condizione, mentre la componente sistematica, esplorando in ampiezza, ha esaminato gli altri possibili rami delle condizioni.

Questi risultati "giocano" a favore della tecnica ibrida, la quale include la componente manuale in grado di scatenare eventi molto significativi sull'applicazione e la componente sistematica in grado di esplorare porzioni di codice non considerate nei test manuali.

RQ 4) L'unione tra la tecnica Manuale e quella Sistematica risulta essere più efficace delle due tecniche da cui è composta se considerate singolarmente?

RS 4) I risultati ottenuti ed organizzati in forma tabellare mostrano chiaramente come l'unione delle due tecniche mette insieme i loro pregi ottenendo, attraverso il Merge delle coperture, un livello di efficacia quasi sempre vicino a quello ottenuto con la tecnica Ibrida, alcune volte uguale e non molte volte lontano.

RQ 5) L'unione tra la tecnica Sistematica e quella Manuale risulta essere più efficace rispetto alla tecnica ibrida?

RS 5) Per rispondere a questa domanda bisogna avvalersi dei risultati ottenuti con l'analisi qualitativa in quanto, se ci soffermassimo ai soli risultati ottenuti dall'analisi quantitativa e quindi i Box-Plot, si potrebbe affermare che nella maggior parte dei casi i livelli di efficacia sono molto vicini quindi una tecnica vale l'altra (non considerando i tempi di esecuzione lunghi della tecnica Ibrida), mentre, approfondendo i dettagli con l'analisi qualitativa, si è visto che la tecnica Ibrida attraversa tutte quelle porzioni di codice di difficile intuizione, copre sempre tutti i rami della logica condizionale soddisfacendo tutte le condizioni e sfrutta al massimo tutti i servizi e le features messe a disposizione dall'App. Sulla base dei risultati ottenuti dalle analisi effettuate si può affermare che la tecnica Ibrida risulta più efficace rispetto all'unione della tecnica Manuale con quella Sistematica.

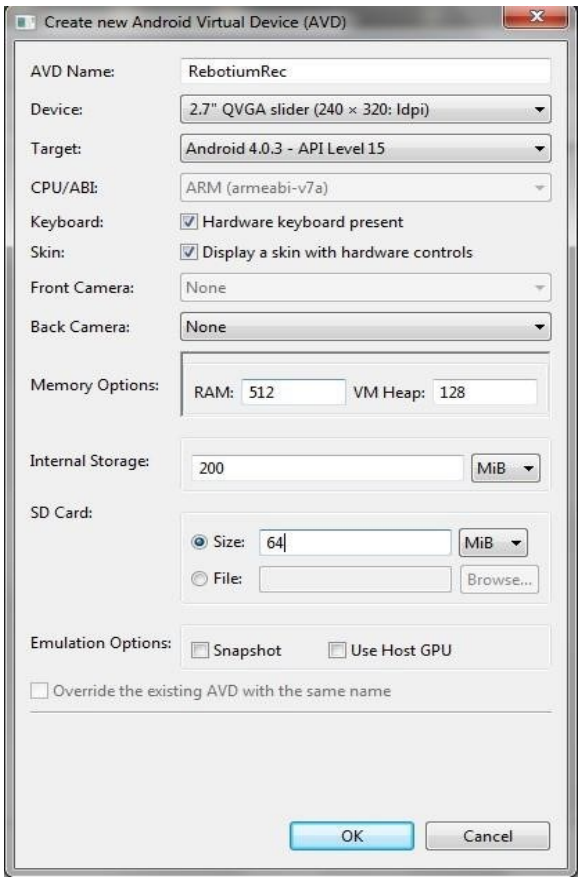
Conclusioni e Sviluppi Futuri

Attraverso il confronto tra l'efficacia delle tecniche di generazione di casi di test, l'adozione della tecnica Ibrida ha portato un significativo vantaggio nell'esplorazione delle applicazioni. La componente sistematica unita alla componente manuale ha consentito la copertura di porzioni di codice difficilmente raggiungibili da una sola delle due. Il progetto Android Ripper Ibrido, oltre a rivelarsi molto efficace, si è dimostrato una valida scelta progettuale avendo riunito i vantaggi dell'esplorazione manuale composta da input di elevata significatività e derivanti dall'intelligenza della componente umana, ed i test generati in maniera sistematica con l'esplorazione in ampiezza delle applicazioni.

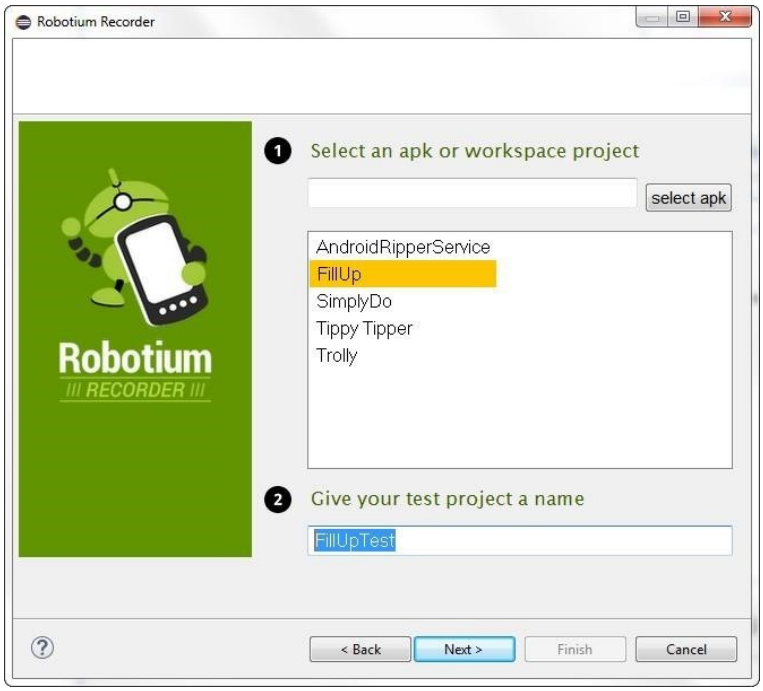
Riproponendo la caratteristica legata all'efficacia dell'unione, ovvero che spesso $\varepsilon_U \backsimeq \varepsilon_I$, si può dedurre che nei casi in cui non si vuole prediligere un valore alto di efficacia, ma si vuole prediligere, ad esempio, un tempo breve, sulla base dei risultati ottenuti concernenti il tempo speso per ogni esecuzione del ripper ibrido, si può scegliere di ricorrere all'unione in quanto, l'esecuzione della riga di comando che permette di realizzarla, risulta essere quasi istantanea. In futuro può essere presa in considerazione l'idea di testare il Ripper Ibrido su applicazioni con activity più complesse in modo da cercare di capire se e dove ci sono limiti che tentano di ostacolare l'alto livello di efficacia ottenibile con questa tecnica di testing.

Appendice A – Utilizzo Robotium Recorder

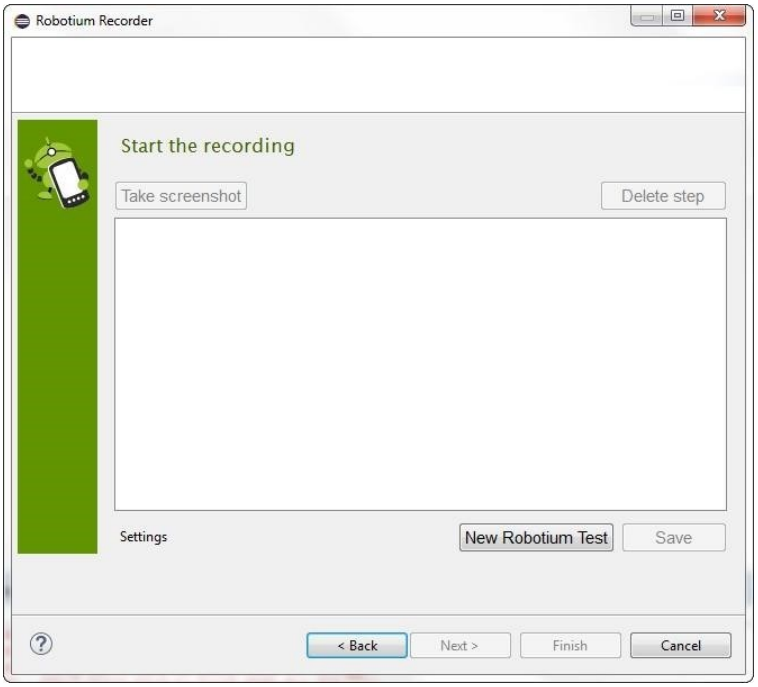
In questa sezione vedremo come eseguire una registrazione di un test con Robotium Recorder ver. 2.1.25 su Eclipse.
Creare un AVD (requisito: API Level 15 o superiore) e avviarlo.



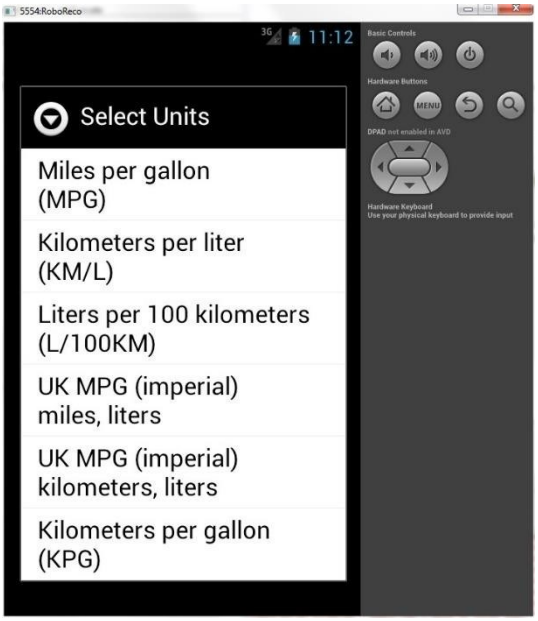
Da Eclipse: File > New > Other. Nella finestra selezionare "New Robotium Test" dalla sezione “Android - Robotium Recorder” e click su "Next".
Selezionare l’applicazione su cui effettuare la registrazione (es. FillUp, il cui progetto è presente in Eclipse) e dare un nome al progetto di test (es. FillUpTest).



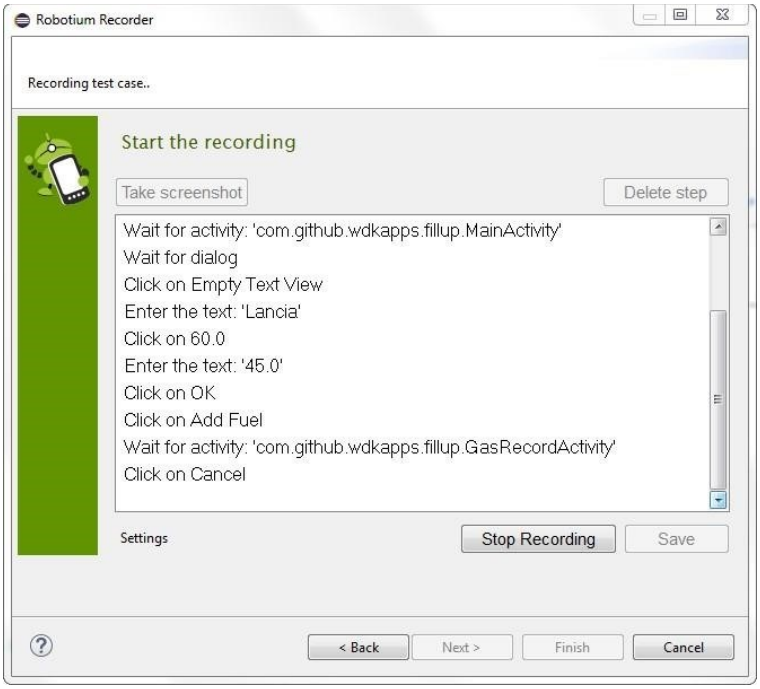
Click su “Next”:



Click su “New Robotium Test”. Nell’AVD verrà eseguita l’applicazione (FillUp); esplorare l’applicazione:



Nella finestra di Robotium Recorder verranno registrati tutti gli eventi scatenati nella GUI dell'applicazione:



Al termine, click su Stop Recording e salvare (clic su "Save") il test case.

È possibile salvare gli screenshot (Take screenshot) e cancellare i singoli eventi registrati (Delete step).
Salvare assegnando un nome al test case (es: TestCase_FillUp).

In Eclipse è stato creato un nuovo progetto "FillUpTest". All'interno del package

"com.github.wdkapps.fillup.test" è presente la classe "TestCase_FillUp.java" relativa alla registrazione.
È fortemente raccomandato, al termine della registrazione, rieseguire il test Android JUnit prodotto per verificare il corretto comportamento.
È necessario a tal proposito cancellare dall'AVD i dati temporanei dell'App o disinstallarla dal dispositivo.

Installazione & Uso	http://robotium.com/pages/installation http://robotium.com/pages/user-guide
Note	Importante: Se è prevista successivamente una esecuzione del Ripper Ibrido, la registrazione con Robotium Recorder e l'esecuzione del Ripper devono essere effettuate usando un AVD avente la stessa dimensione dello schermo (nell'esempio 2.7" QVGA (240 x 320: ldpi))

Appendice B – Installazione e uso Android Ripper Ibrido

B.1 Prerequisiti

Requisiti di sistema	
Sistema operativo	Windows XP o più recente, Linux, Mac OS X
JDK	Versione 1.7.25 o più recente
Android Tools	È necessario avere esattamente queste versioni: • Android SDK Tools ver. 22.3; • Android SDK Platform Tools ver. 19.0.2; • Android SDK Build Tools ver. 18.1.1 È possibile scaricare l'installer per Windows, Linux e Mac OS X o il pacchetto compresso pronto all'uso per Windows, Linux e Mac OS X.
Android Libraries	Versione 2.3.3 o più recenti
Ant	Versione 1.8.2 o più recente.

Note	Se l'applicazione dichiara nel Manifest.xml ha una versione di "targetSdkVersion" e "minSdkVersion" maggiori della versione della libreria del Ripper (10), è necessario installare le librerie richieste attraverso l'SDK Manager

Configurazione di sistema (variabili ambiente)	
ANDROID_HOME	settata al path dell'Android SDK (e.g. android-sdk)
JAVA_HOME	settata al path di Java SDK (e.g. "C:\Java\jdk1.7.0_25")
PATH	deve contenere: "bin" path di Ant "platform-tools" path di Android SDK "tool" path di Android SDK Per OS Windows aggiungere "aapt.exe" al path
Android Libraries	Versione 2.3.3 o più recenti
Ant	Versione 1.8.2 o più recente.

Creazione AVD	
Target	Android 2.3.3 - API Level 10 (o più recente)
Device Ram Size	512 MB o maggiore
Max VM Heap	128 MB
Internal Storage	200 MB o più

SD Card Size	64 MB o più
Snapshot	Attivato
Note	E' fortemente raccomandato creare un nuovo AVD per ogni esecuzione del Ripper

Registrazione utente con Robotium Recorder	
Versione	2.1.25 (o più recente)
Installazione & Uso	http://robotium.com/pages/installation http://robotium.com/pages/user-guide
Note	È fortemente raccomandato, al termine della registrazione, cancellare dall'AVD i dati dell'applicazione e rieseguire il test Android JUnit prodotto per verificare il corretto comportamento. E' importante che l'esecuzione del Ripper e la registrazione con Robotium Recorder siano effettuate usando un AVD avente la stessa dimensione dello schermo (stesso device emulato)

B.2 Android Ripper Installer

AZIONI	
Estrarre l'archivio Android Ripper Installer	
Editare il file "ripper.properties"	**AUT_PATH**: la directory radice dell'applicazione da testare (AUT). In Windows usare gli slash ("/"). Tale directory deve contenere il file "Manifest"
	TEST_RR_FILE: file registrato con Robotium Recorder comprensivo di path. In Windows usare gli slash "/" (e.g. C:/ActivityTest/src/com/ex/test/ActivityTest.java)
	AVD_NAME: nome dell'AVD
	AVD_PORT: porta dell'AVD (default: 5554)
Eseguire Android Ripper Installer	java -jar AndroidRipperInstaller.jar
Chiudere emulatore al termine	

B.3 Android Ripper Driver

B.3.1 Fase 1

AZIONI	
Estrarre l'archivio Android Ripper Driver	
Editare il file "ripper.properties"	**APP_PACKAGE**: il nome del package principale dell'AUT (e.g. com.example). Il nome del package può essere trovato nel file "Manifest" come valore dell'attributo "package" del tag "manifest"
	APP_MAIN_ACTIVITY: il nome della classe dell'activity principale comprensiva del package (e.g. com.example.ui.MainActivity) Il nome dell'activity può essere trovato nel file Manifest.xml come valore dell'attributo android:name di uno dei tag "activity"
	AVD_NAME: nome dell'AVD
	AVD_PORT: porta dell'AVD (default: 5554)

Eseguire Android Ripper Installer	java -jar AndroidRipper.jar s systematic.properties

B.3.2 Fase 2

AZIONI	
Eseguire nuovamente Android Ripper Driver al termine della fase1	java -jar AndroidRipper.jar s systematic.properties

Per una nuova esecuzione, eliminare i file di output prodotti nella precedente.

Bibliografia

[1] Autori, Titolo dell'articolo, Nome della rivista, volume, pagine, Data

[2] Autori, Titolo del Libro, Editore, Anno, pagine.

[3]Autori, Titolo dell'articolo, Nome della conferenza di cui è agli atti, editore, anno, pagine.

[4] Nome del sito web, indirizzo http, ultima data in cui è stato acceduto