UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Tesi di Laurea Magistrale in Ingegneria Informatica

Un esperimento di comparazione tra l'efficacia di tecniche di generazione di casi di test manuali, sistematiche ed ibride per applicazioni Android

Anno Accademico 2014-2015

relatore

Ch.mo Prof. Porfirio Tramontana

correlatore
Ing. Nicola Amatucci

candidato
Giuseppe Polino
matr. M63000087

[Dedica]

Indice

Introduzione

Stiamo assistendo ad una diffusione ormai esponenziale dei dispositivi da utilizzare in mobilità, in particolare di smartphone e tablet, che sta cambiando il modo di interfacciarsi con i servizi e con gli altri. I dispositivi mobili sono diventati, dunque, una realtà che nessuno che si occupi di sviluppo software può ignorare. Appoggiando i polpastrelli su icone larghe pochi millimetri di una GUI (Graphic User Interface) è possibile avere accesso a svariate funzionalità rese disponibili dalle applicazioni (più comunemente note come *App*) le cui performance sono legate anche alle attività di testing effettuate su di esse ed in particolare, come tratteremo in questa tesi, alle attività di testing automation su di una interfaccia grafica.

Le tecniche di Testing Automation consistono nell'implementazione e uso di tecnologie software per la costruzione e l'esecuzione parziale o totale di casi di test ripetibili e consistenti, con l'obiettivo di ridurre i tempi ed i costi del *software testing*. Automatizzare il test, in modo da essere eseguito senza l'intervento umano, riduce drasticamente il costo dell'esecuzione del test. È in pratica un rafforzamento delle metodologie e dei processi utilizzati nel test manuale, anche se non lo sostituisce del tutto, poiché alcune verifiche (ad esempio quelle sul layout grafico e sull'usabilità del prodotto) sembrerebbero più efficaci se eseguite manualmente.

Obiettivo del presente lavoro è concentrare l'attenzione su questo aspetto, ovvero comparare

l'uso di diverse tecniche, manuali, automatiche e ibride, in modo da poter valutare se esistono vantaggi significativi nell'esplorazione delle applicazioni e un aumento di efficacia.

Nel primo capitolo di questo lavoro verrà introdotto il sistema operativo Android descrivendone le caratteristiche, mentre nel secondo capitolo viene presentata una panoramica sul testing, in particolare sul testing di applicazioni Android e sul GUI Testing. Saranno anche introdotte le tecniche di recording ed in particolare il framework Robotium Recorder che è stato utilizzato per questo lavoro.

Il terzo ed il quarto capitolo sono dedicati a due delle tecniche utilizzate per la generazione di casi di test: Android GUI Ripper e Android Ripper Ibrido.

Android Ripper Ibrido, di cui si parlerà in dettaglio nel capitolo 4, è basato sull'Android GUI Ripper sviluppato da REvERSE (REsEarch laboRatory of Software Engineering) Group of the University of Naples "Federico II, e riunisce la componente umana e la componente automatica per la generazione e l'esecuzione dei test sulla GUI di alcune applicazioni scelte. L'attenzione verrà concentrata sulle modifiche apportate rispetto al progetto originale.

Nel quinto e ultimo capitolo di questa tesi si concentra tutta la fase di sperimentazione del lavoro svolto. Vengono analizzati in dettaglio i risultati ottenuti dal confronto tra le tecniche utilizzate nel corso degli esperimenti e si motivano questi risultati sia per mezzo di analisi qualitative, sia attraverso analisi statistiche. Vengono inoltre presentate le applicazioni su cui sono stati eseguiti i test, le configurazioni, i metodi e le procedure per raggiungere gli obiettivi prefissati, ossia comparare l'efficacia delle tecniche e validare o invalidare le ipotesi fatte nei confronti dell'esercizio di queste sulle cinque applicazioni coinvolte.

Capitolo 1: Android

In questo capitolo viene illustrato il contesto generale del sistema Android descrivendone le caratteristiche. Successivamente viene presentata una panoramica sul testing, in particolare sul testing di applicazioni Android e sul GUI Testing.

1.1 Perchè Android

Il mercato dei dispositivi mobili ha conosciuto negli ultimi anni un notevole sviluppo, arrivando a cambiare il concetto di fruibilità di molti contenuti e servizi che prima erano accessibili solo da un PC. In breve tempo si è passati all'adottare dispositivi sempre più curati nel design e nelle prestazioni. Chi oggi acquista uno di questi dispositivi intelligenti, di sicuro si imbatterà nel simpatico robottino verde, simbolo di Android, senza dubbio il sistema operativo mobile attualmente più diffuso.

Tutto Android ruota attorno al mondo open source: infatti si ha libero accesso a tutti gli strumenti utilizzati dagli sviluppatori, inoltre, chiunque può sviluppare per Android senza dover acquisire software o licenze particolari e se gli strumenti e gli opportuni documenti, libri, riviste per lo sviluppatore non mancano di certo, lo stesso si può dire degli strumenti per il testing delle applicazioni in Android. Tuttavia per la maggior parte delle applicazioni vengono rilasciate numerose releases anche a breve distanza l'una dall'altra, il che sottolinea la minore attenzione dedicata a queste fase durante lo sviluppo dell'applicazione stessa. Il testing in molte delle aziende per lo sviluppo software è una delle ultime attività a essere svolte prima della release ovvero quando le scadenze sono ormai prossime e il committente

scalpita dal desiderio di vedere il prodotto finito e funzionante. Ebbene, questo è sicuramente un deterrente per un proficuo svolgimento di questa fase.

Il testing delle applicazioni in Android è reso per alcuni aspetti semplice dall' uso della libreria JUnit di Java (che è il linguaggio adoperato per lo sviluppo di applicazioni Android). Il linguaggio Java, per Android, è conosciuto, diffuso, molto utilizzato ed inoltre gestisce automaticamente il ciclo di vita di un'applicazione e anche l'allocazione della memoria, rendendo più semplice lo sviluppo.

1.2 Il sistema Android

Google nel 2005 rileva la Android Inc. fondata nel 2003, una società che aveva tra i suoi progetti lo sviluppo di un nuovo sistema operativo mobile, e comincia a progettare la sua piattaforma. Nel 2007 annuncia Android e il rilascio delle prime SDK (Software Develoment Kit) per lo sviluppo. Il sistema è stato progettato principalmente per smartphone e tablet, con interfacce utente specializzate per televisori (Android TV), automobili (Android Auto), orologi da polso (Android Wear), occhiali (Google Glass), e altri. È per la quasi totalità Free and Open Source Software (ad esclusione per esempio dei driver non-liberi inclusi per i produttori di dispositivi) ed è distribuito sotto i termini della licenza libera Apache 2.0.

Android dispone di una vasta comunità di sviluppatori che realizzano applicazioni, scritte soprattutto in linguaggio di programmazione Java, con l'obiettivo di aumentare le funzionalità dei dispositivi. Tutti questi fattori hanno permesso ad Android di diventare il sistema operativo più utilizzato in ambito mobile, oltre a rappresentare, per le aziende produttrici, la migliore scelta in termini di bassi costi, personalizzazione e leggerezza del sistema operativo stesso. [1]

1.3 Architettura di Android

Il sistema operativo Android è basato su kernel Linux e consiste in una struttura formata da vari livelli, ognuno dei quali fornisce al livello superiore un'astrazione del sistema sottostante.

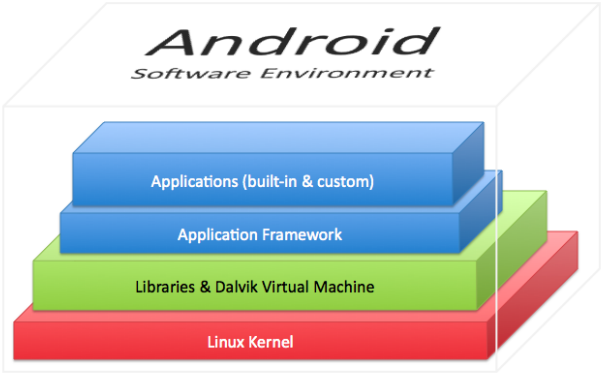


Figura 1 Android Software Environment

I livelli principali sono: Linux Kernel, Libraries, Application Framework, Applications. Come vediamo in dettaglio in Figura 2.

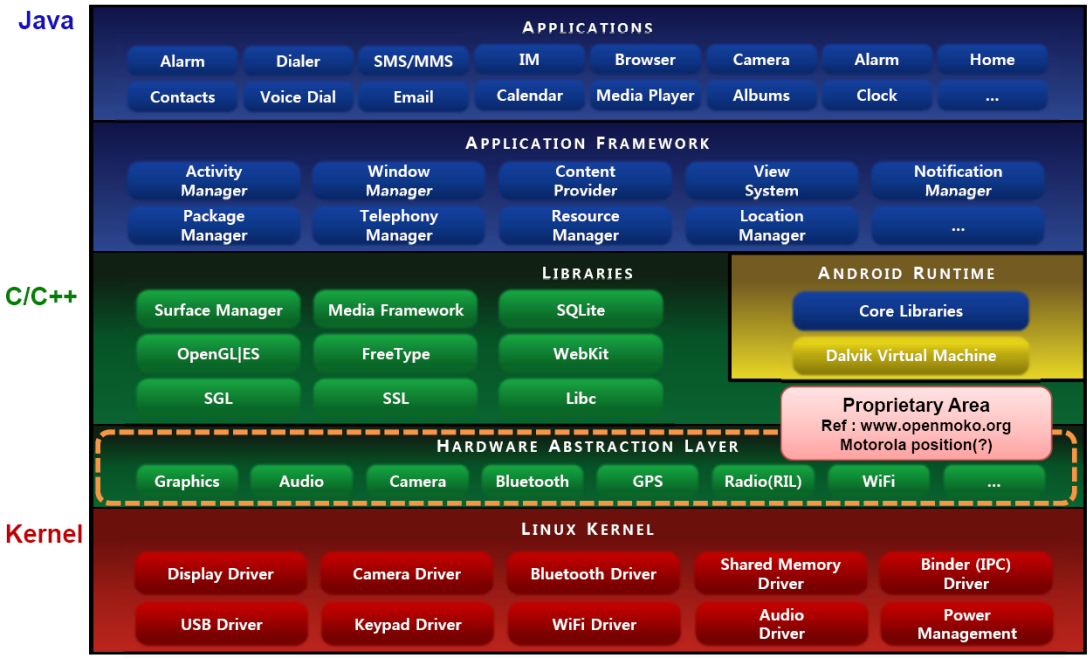


Figura 2 Architettura Android [8]

1.3.1 Linux Kernel

Alla base dello stack Android troviamo un kernel Linux. La scelta di una simile configurazione è nata dalla necessità di disporre di un vero e proprio sistema operativo che fornisse gli strumenti di basso livello per la virtualizzazione dell'hardware sottostante attraverso l'utilizzo di diversi driver. A differenza di un kernel Linux standard per Android sono stati aggiunti ulteriori moduli come:

- *Binder IPC Driver*, un driver dedicato che permette la comunicazione tra processi con un costo computazionale minore e un relativo minore consumo di batteria;
- *Low Memory Killer*, un modulo che si preoccupa di terminare i processi liberando così spazio nella memoria centrale per soddisfare le richieste di un altri processi. Mediante un sistema di ranking vengono terminati i processi con punteggio più alto; Ad esempio, un processo che controlla la UI (User Interface) di un'applicazione visibile sarà sicuramente più basso di quello relativo ad un'applicazione non visibile sullo schermo.
- *Ashmem*, sistema di memoria condiviso anonimo (Anonimous Shared Memory) che definisce interfacce che consentono ai processi di condividere zone di memoria attraverso un nome. Il vantaggio è che viene fornito al kernel uno strumento per recuperare dei blocchi di memoria non utilizzati;
- *RAM Console e Log devices*, per agevolare il debugging. Android fornisce la capacità di memorizzare i messaggi di log generati dal kernel in un buffer RAM. È disponibile inoltre un modulo separato che permette ai processi utente di leggere e scrivere messaggi di log;
- *Android Debug Bridge*, uno strumento che permette di gestire in maniera versatile un'istanza dell'emulatore o eventualmente di un dispositivo reale;
- *Power Management*, sezione progettata per permettere alla CPU di adattare il proprio funzionamento al fine di non consumare energia se nessuna applicazione o servizio ne fa richiesta. [9][10]

1.3.2 Android Runtime

Le Core Libraries insieme alla Dalvik Virtual Machine (DVM) appartengono logicamente all'Android Runtime cioè una parte dello stack Android dedicata all'esecuzione delle applicazioni. Le Core Libraries di fatto mettono a disposizione la maggior parte delle funzionalità disponibili nel linguaggio di programmazione Java: strutture dati, utility, librerie per l'accesso ai file, librerie di rete e così via.

Un programma Java per essere eseguito necessita di una Virtual Machine (JVM) in grado di interpretare il bytecode (file di estensione `.jar`) ed eseguirlo su ogni macchina. In realtà la Virtual Machine utilizzata per Android è la DVM in grado di eseguire codice contenuto all'interno di un file di estensione `.dex`, ottenuti a partire dal bytecode Java. Questa scelta è dovuta al fatto che vi è un risparmio di spazio in quanto un file `.dex` ha dimensioni nettamente inferiori rispetto ad un `.jar`. Inoltre 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, ma allo stesso tempo porta delle implicazioni dal punto di vista della sicurezza.

1.3.3 Libraries

Le librerie hanno un'importantissima caratteristica: non sono sviluppate in Java ma perlopiù in C/C++. Ebbene sì, 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.

In particolare tra di esse 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;

1.3.4 Application Framework

Tutte le librerie viste finora 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 compito dell'Activity Manager è quello di 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. [14]

1.3.5 Applications

Il livello più alto dello stack è costituito dalle applicazioni: non soltanto quelle native come per esempio il sistema di gestione dei contatti, l'applicazione per l'invio di SMS, il calendario e tante altre, ma anche quelle provenienti da altre fonti. Android non differenzia le applicazioni di terze parti da quelle già incluse nel telefono, infatti garantisce gli stessi privilegi a entrambe le categorie.

1.4 Architettura delle applicazioni Android

Le applicazioni Android sono sviluppate nel linguaggio di programmazione Java mediante l'utilizzo dell'Android Software Development Kit (SDK) oppure in C/C++ utilizzando il Native Development Kit (NDK). Con SDK si indica genericamente un insieme di strumenti per lo sviluppo e la documentazione di software. Le applicazioni consistono di uno o più componenti e di meccanismi di comunicazione in grado di ottimizzare lo sfruttamento delle risorse oltre che ad un'alta propensione alla personalizzazione e all'estensibilità della piattaforma. Tra i componenti fondamentali troviamo:

- *Activity*: rappresenta una singola schermata che contiene l'interfaccia dell'applicazione con lo scopo di gestire l'interazione tra l'utente e l'applicazione stessa. Viene realizzata attraverso la definizione di `Activity` descritte da specializzazioni dell'omonima classe del package `android.app` e risulta una delle classi fondamentali dell'architettura. Ogni applicazione consisterà di una sequenza di schermate attraverso le quali l'utente avrà la possibilità di interagire con il sistema sottostante.
- *Services*: consistono di processi con la responsabilità di eseguire applicazioni in background di lunga durata come ad esempio riprodurre file multimediali, leggere o scrivere informazioni attraverso la rete. Possiamo pensare ai service come a un insieme di componenti in grado di garantire l'esecuzione di alcuni task in background in modo indipendente da ciò che è visualizzato nel display, e quindi con ciò da cui l'utente in quel momento sta interagendo.
- *Content Provider*: gestisce un insieme condiviso di dati di un'applicazione. Visto che è possibile memorizzare i dati, ad esempio su un database SQLite, sul web, o qualsiasi altro luogo di memorizzazione permanente, una applicazione può accedere, interrogare o anche modificare i dati (se dispone di appositi permessi).
- *Broadcast Receiver*: componente che risponde a livello di sistema agli annunci trasmessi in broadcast. Un'applicazione può inviare annunci in broadcast (ad

esempio per far sapere che determinati dati sono stati scaricati dal dispositivo e che sono disponibili). Sebbene questi componenti non mostrano un'interfaccia utente, essi solitamente possono creare una notifica sulla barra di stato, che avverte l'utente in caso di eventi broadcast.

- *Intent e Intent Filter*: attraverso un intent (tradotto come "intenzioni"), 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 la stessa 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 degli intent riguarda la comunicazione tra più Activity. Possiamo distinguere tra Intent esplicito che permette il passaggio da un'Activity ad un'altra specificando esplicitamente la classe Java dell'Activity che si vuole mandare in esecuzione e Intent implicito in cui 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.[14]

1.4.1 Activity Lifecycle

Come già detto nei paragrafi precedenti una Activity rappresenta una possibile interazione dell'utente con l'applicazione e può essere associata al concetto di schermata. Lo scopo principale di uno sviluppatore deve essere naturalmente quello di realizzare applicazioni in grado di rispondere immediatamente alle azioni dell'utente. Una normale applicazione consisterà di una sequenza di schermate, alcune delle quali ad esempio permettono la visualizzazione di informazioni ed altre invece dispongono dei componenti per l'interazione. In ogni caso esistono schermate che si alternano sul display, comunicando eventualmente tra di loro e scambiandosi informazioni. A tale scopo la piattaforma organizza le attività secondo una struttura a stack, dove quella in cima è sempre quella attiva.

La visualizzazione di una nuova schermata però comporta l'avvio di una nuova Activity la quale si porterà in cima allo stack, mettendo in pausa quella precedente. E' necessario prevedere che un'Activity non in cima allo stack possa essere eliminata dal sistema per poi essere eventualmente ripristinata successivamente. Responsabilità del sistema sarà quella di fare in modo che tutte queste operazioni siano trasparenti all'utente che utilizza il dispositivo. L' Activity Manager si occupa della gestione dello stack e della gestione del ciclo di vita delle Activity. L'arresto di un'Activity a causa dell'avvio di una nuova viene notificato all'Activity stessa mediante i metodi di callback che governano il suo ciclo di vita. [2]

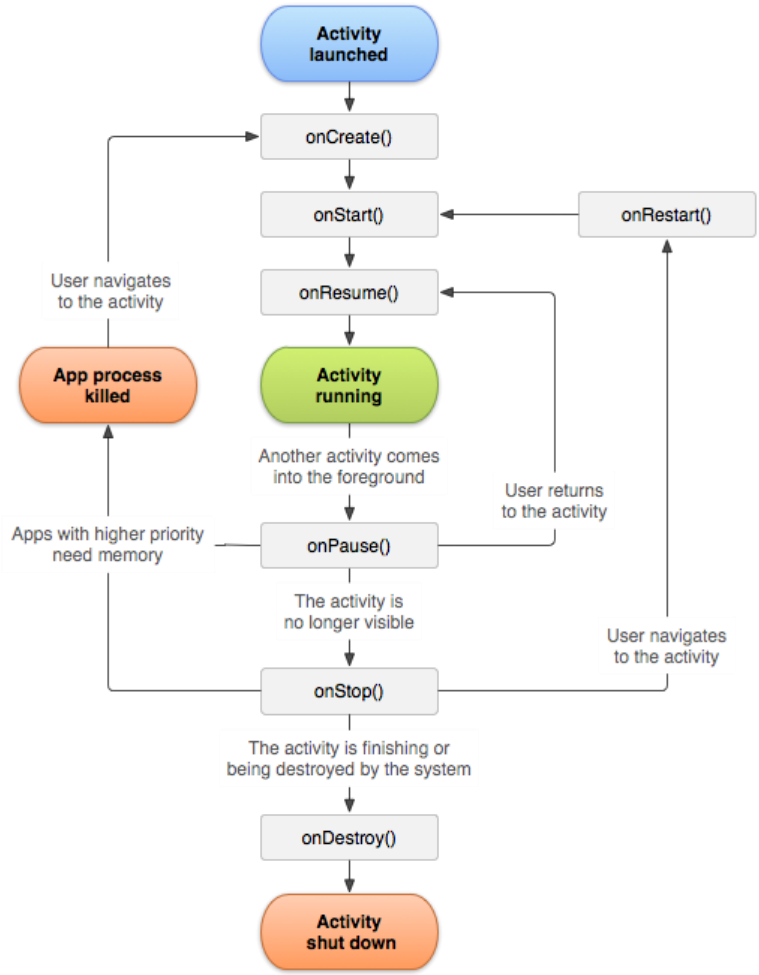


Figura 3 Activity Lifecycle

Una Activity ha essenzialmente quattro stati:

- ACTIVE: se una Activity è in primo piano sullo schermo (cioè nella parte superiore della pila), vuol dire che si trova nello stato attivo o di 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.

In generale il passaggio dell'Activity attraverso il suo ciclo di vita è dettato dai metodi che possiamo schematizzare nella seguente tabella:

Metodi	Descrizione
onCreate ()	Si tratta dell'operazione invocata in corrispondenza della creazione dell'Activity e che, nella maggioranza 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.

Tabella 1. Metodi del ciclo di vita di una Activity [2]

1.4.2 User Interface

Un'interfaccia grafica è composta da una serie di schermate a loro volta costituite da oggetti con i quali l'utente può interagire al fine di controllare l'applicazione. Ogni interazione valida genera particolari messaggi, detti eventi, che vengono raccolti da componenti in ascolto, generalmente esterni alla GUI. Nei paragrafi seguenti ci si limita a osservazioni su alcune caratteristiche del sistema Android, per quel che riguarda l'interfaccia grafica e alcuni dei componenti che la costituiscono, e che più avanti torneranno utili quando si analizzeranno le tecniche Android GUI Testing.

Nelle applicazioni Android l'interfaccia utente viene costruita utilizzando le classi *View* e *ViewGroup*. La prima serve da base per la sottoclasse chiamata *widget* che offre numerosi componenti per l'interazione con l'utente. La classe *ViewGroup*, invece, serve come base per la sottoclasse chiamata *layout*, la quale offre appunto differenti tipi di layout. Un oggetto *View* è una struttura dati contenente le proprietà e i parametri di una specifica area rettangolare dello schermo. Inoltre esso gestisce anche il layout, il disegno, il focus, lo scorrimento per quella porzione di schermo nella quale risiede.

1.4.2.1 Gerarchie delle View

Su una piattaforma Android è possibile definire l'interfaccia utente di un'Activity usando una gerarchia di oggetti *View e ViewGroup* come mostrato in Figura 4: gli oggetti *View* rappresentano le foglie dell'albero, mentre gli oggetti *ViewGroup* rappresentano i rami. L'albero gerarchico può essere semplice o complesso a seconda di ciò che si vuole realizzare. Per fissare una gerarchia di *View* sullo schermo per il rendering, una Activity deve utilizzare il metodo *setContentView()* e passare un riferimento al nodo principale. Quest'ultimo richiede che i suoi figli vengano disegnati; a loro volta, ogni nodo *ViewGroup* sarà responsabile della creazione dei suoi nodi sottostanti. [14]

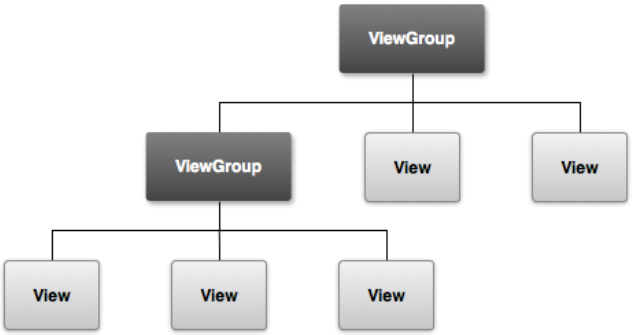


Figura 4. View e ViewGroup

1.4.2.2 Layout

Il modo più comune di definire un layout ed esprimere una gerarchia di View è quello di utilizzare i file XML. Il linguaggio XML permette infatti di realizzare una struttura leggibile per il layout. Ogni elemento XML rappresenta o un oggetto View oppure un oggetto ViewGroup. In pratica il layout permette di disporre (a seconda del tipo di layout) le View in esso contenute. Si possono anche creare layout personalizzati attraverso la definizione di attributi ed elementi custom nei file XML di layout.

1.4.2.3 Widget

I widget sono particolari specializzazioni delle View, ovvero 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, hanno responsabilità ben precise e fanno parte del package `android.widget`.

Alcuni di questi sono:

- *TextView*: permette di mostrare del testo all'utente;
- *EditTex*t: 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.

1.4.2.4 Eventi di input e gestione

Uno dei concetti principali nell'utilizzo dei componenti di un'interfaccia grafica riguarda la gestione degli eventi. Se pensiamo ad un bottone o ad una lista, per esempio, servirà un meccanismo che permetta di eseguire particolari operazioni in corrispondenza della sua selezione. Per questo si sfrutta il Delegation Model ispirato da Java. 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. Questo è ciò che si dovrebbe fare quando si implementa una View e si desidera ascoltare particolari eventi che si verificano al suo interno. Gli eventi che possono essere gestiti sono ad esempio il tocco dello schermo *onTouchEvent()*, oppure la pressione di uno dei tasti del device. Questo permette di definire il comportamento di default per ogni evento all'interno della View. [3]

1.4.2.5 Ulteriori elementi di iterazione

Non volendo dilungarsi troppo su tutti gli ulteriori elementi di iterazione, se ne citano solo alcuni di maggior interesse per le caratteristiche offerte. Tra questi i *Menù* che consentono di scegliere i settaggi di una applicazione; *i Dialog* che consistono in delle finestre di dialogo per inserimenti di brevi informazioni o nella scelta di opzioni utili all'applicazione stessa per continuare.

Capitolo 2: Android GUI Testing

In questo capitolo si descrivono le principali tecniche di GUI Testing per un'applicazione Android e alcuni strumenti utili su questa piattaforma per il testing dell'interfaccia grafica. Queste tecniche rappresentano uno degli approcci principali per testare prodotti software di questo tipo. Il capitolo introduce le tecniche di recording ed in particolare il framework che è stato utilizzato per questo lavoro di tesi.

2.1 Testing GUI Based

La GUI (*Graphical User Interface*) è il mezzo con cui l'utente interagisce con il sistema software; essa risponde agli eventi generati (come ad esempio un click) eseguendo il codice ad essi collegato. Il sistema è in uno stato stabile fino all'intervenire di un evento utente, che fa partire un codice di "event handling". A differenza di altri approcci, il GUI based prevede componenti in grado di rilevare e riconoscere i componenti della GUI, esercitare eventi su di essa, fornire degli input ai componenti (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.

2.2 Testing di applicazioni Android

Le transizioni di stato dell'interfaccia utente di un'applicazione Android, come abbiamo detto, sono guidate da eventi che possono essere originati dall'utente oppure associati a messaggi di interrupt inviati da uno dei sensori o dispositivi di comunicazione di cui è dotato il dispositivo su cui l'applicazione è in esecuzione. Android mette a disposizione apposite classi che, estendendo quelle di JUnit, automatizzano le procedure necessarie ad interfacciarsi col componente, permettendo ai tester di concentrarsi unicamente sulla definizione delle operazioni e delle condizioni che compongono i casi di test.

Poiché le attività di testing nel processo di sviluppo software sono molto importanti, Android mette a disposizione strumenti di supporto che ci permettono di effettuare in modo semplice ed automatizzato il testing a vari livelli: testing di unità, testing funzionale, testing di regressione e testing della GUI.

Esistono diversi approcci per il testing GUI-based come le tecniche Model-Based, nelle quali esiste una descrizione formale dell'applicazione sotto test ed in particolare della GUI; tecniche di Random Testing, nelle quali, in assenza di un modello, l'applicazione viene esercitata in maniera casuale, alla ricerca di eventuali eccezioni non gestite; ed esistono poi anche tecniche Model Learning, basate sull'esplorazione delle applicazioni che invece di avvenire in maniera casuale vengono effettuate in maniera metodica, seguendo una precisa strategia di navigazione; tale tecnica permette non solo di rilevare eccezioni non gestite (crash testing), ma permette anche di effettuare il reverse engineering del modello, il quale potrà essere utilizzato per ulteriori elaborazioni. Un approccio simile è utilizzato nell'Android Ripper, che sarà oggetto di discussione nei prossimi capitoli.

2.2.1 Strumenti per testing Android

Illustriamo brevemente alcuni strumenti utili per il testing e la valutazione di questi ultimi in Android.

- Robotium: è un Android test automation framework creato allo scopo di facilitare la scrittura di casi di test black-box automatizzati per le applicazioni Android. Permette di definire e comprendere in modo semplice i casi di test dedicati alle Activity

Android. Il funzionamento di Robotium è basato sull'utilizzo di un oggetto denominato "`Solo`", tramite il quale è possibile interrogare e modificare i widget della UI. [11]

- Monkey: è un programma che viene eseguito sull'emulatore o sul device con lo scopo di generare delle sequenze pseudo-casuali di eventi utente e di eventi di sistema. Utilizzato pricipalmente per effettuare lo stress-test delle applicazioni.
- MonkeyRunner: è un API che consente la scrittura di programmi in grado di controllare un dispositivo Android dall'esterno;
- Emma: è uno strumento che permette di misurare la copertura del codice in ambito Java. Ci permette di conoscere quante e quali parti del codice sorgente siano state effettivamente attraversate durante l'esecuzione di un test, permettendo di ottenere una misura quantitativa della bontà della test-suite utilizzata. Genera report e metriche, anche in formato HTML e risulta molto utile per il testing White Box.[12]

2.3 Android Testing Framework

Android è dotato di un framework che è parte integrante dell'ambiente di sviluppo ed è basato su JUnit, il cui schema di principio è descritto in Figura 5.

I casi di test JUnit sono metodi Java, organizzati in classi di test, contenute in package di test che compongono il progetto.

Il framework viene esteso da classi specifiche per i vari tipi di componenti dell'application framework in maniera tale da permettere di svolgere operazioni come la creazione di stub ed il controllo del ciclo di vita del componente.

Mentre in JUnit si utilizza direttamente un "*test runner"* per l'esecuzione dei casi di test, in Android è necessario impiegare appositi strumenti per caricare i package del progetto di test e l'applicazione sotto test.

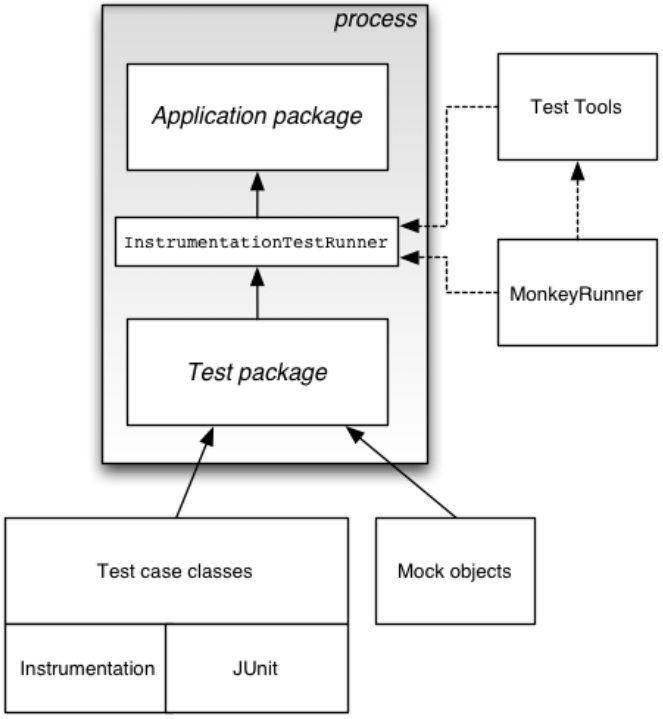


Figura 5 Android Testing Framework

Quindi, attraverso la realizzazione di una particolare implementazione di test runner che fa uso dei tool di *Instrumentation*, un'applicazione di test Android potrà interagire con la AUT (Application Under Test) gestendone il ciclo di vita dopo aver sostituito alcuni dei suoi componenti con altrettanti componenti *Mock*. Questo è possibile in quanto il test e la AUT verranno eseguiti in thread diversi ma appartenenti allo stesso processo.

L'implementazione di test runner fornita con Android SDK, che in questo caso è descritta dalla classe *InstrumentationTestRunner,* bloccherà l'AUT e avvierà quella di test che avrà la responsabilità di gestire il ciclo di vita e quindi di testarne lo stato nei diversi casi. Le API di Instrumentation permettono quindi di interagire con l'AUT inviando eventi di gestione della UI dell'applicazione reale. [4]

2.4 Testing Automation e Recording

Con il termine Testing Automation si vuole indicare l'insieme delle tecniche e delle tecnologie che consentono di automatizzare (anche parzialmente) alcune attività del processo di testing. Le difficoltà principali legate al testing della GUI sono da ritrovarsi nella sua natura dinamica. Infatti l'interfaccia grafica dovrà essere esercitata dal tester cercando di riprodurre lo stesso comportamento dell'utilizzatore finale. L'automatizzazione di un simile procedimento richiede l'utilizzo di un software che riproduca tali azioni. Uno strumento automatizzato può riprodurre tutta una serie di compiti, che possono essere ripetuti per un numero svariato di volte con diversi input e possono essere estesi per funzionalità aggiuntive anche in un secondo momento.

Alcune dovute considerazioni devono essere fatte sulla generazione dei casi di test, la preparazione ed esecuzione del test e la valutazione dell'esito dei casi di test.

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. Tecniche per la generazione automatica dei casi di test possono ridurre drasticamente i costi e i tempi legati alla fase di test design, tuttavia può però essere necessaria una fase di valutazione dell'efficacia dei casi di test e una fase di riduzione dei casi di test ridondanti. I test case possono quindi 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 che sono in grado di mantenere un log di tutte le interazioni che avvengono tra gli utenti dell'applicazione da testare e l'applicazione stessa (fase di Capture); a partire da tali dati vengono formalizzati casi di test che replichino 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 essere l'oracolo per i futuri Replay.

Il recording di casi di test è possibile per mezzo di alcuni strumenti che catturano gli eventi

generati. Nel paragrafo seguente l'attenzione verrà focalizzata su Robotium Recorder ambiente che supporta il Capture & Replay e che è oggetto di interesse per questo lavoro di tesi.

2.6 Robotium Recorder

Robotium Recorder è un framework che comprende un insieme di strumenti che consentono l'automazione di test di applicazioni Android, permettendo a sviluppatori e tester di generare casi di test in pochissimo tempo ; è inoltre integrabile sia in Eclipse che in Android Studio. E' un prodotto parzialmente di libero utilizzo (sono disponibili gratuitamente solo 5 registrazioni), che fornisce funzionalità di Capture & Replay su Android. La fase di Capture è in grado di osservare le esecuzione utente su un dispositivo reale o anche su un emulatore, mentre la fase di Replay è in grado di rieseguire le interazioni catturate e anche di generare codice JUnit + Robotium rieseguibile. Dopo l'installazione e la sua esecuzione, l'utente ha a disposizione un progetto di test su cui è possibile anche modificare i vari input registrati, coerentemente con la struttura dell'applicazione sotto test. [5] In Appendice A sono elencati i vari passi per eseguire queste procedure e le configurazioni utilizzate.

Dopo questa breve introduzione allo strumento, si procede ad approfondire lo scopo del suo utilizzo.

Robotium Recorder nella versione 2.1.25, come anticipato, produce un progetto di test relativo alla AUT (Application Under Test) che può essere rieseguito automaticamente come qualsiasi altro test, senza ulteriore supporto di Robotium Recorder. Nel package di test troveremo una classe che contiene codice Java e fa uso di un oggetto denominato "`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.

Un esempio del codice generato nella classe principale di test che estende ActivityInstrumentationTestCase2 è quello mostrato di seguito:

```
// Click on OK
```

solo.clickOnView(solo.getView(com.github.wdkapps.fillup.R.id.*buttonOK*));

Questa riga, insieme ad altre, è contenuta all'interno di un metodo chiamato `testRun()` e mostra come l'oggetto `solo` tramite metodi di libreria riesca ad intercettare il componente della AUT, simulandone la pressione. Infatti il codice mostrato servirà per replicare l'input utente associato alla pressione del tasto, che in questo caso è un bottone di OK dell'interfaccia grafica.

La classe in questione, composta dal metodo appena citato testRun() e altri metodi tra cui il setUp(), tearDown() e il costruttore della classe, costituiranno un input appositamente strutturato ed adattato per essere utilizzato nel progetto Ripper Ibrido trattato nel capitolo 4. La classe di test generata da Robotium Recorder verrà suddivisa in base a specifici eventi e i singoli metodi che faranno parte della nuova classe di test verranno richiamati per essere eseguiti all'interno del Ripper Ibrido.

Gli aspetti implementativi saranno discussi in seguito, per adesso è possibile anticipare che verrà generata una nuova classe, da uno strumento denominato `tool_jk.jar`, che non farà altro che costruire una nuova classe a runtime ridefinendo il metodo testRun() e suddividendolo in altri metodi testRunX() numerati in base agli eventi scelti in fase di progettazione.

Capitolo 3: Android Ripper

In questo capitolo si descrive Android Ripper, uno strumento per il testing automatico sviluppato da REvERSE (REsEarch laboRatory of Software Engineering) Group of the University of Naples "Federico II", che permette l'esplorazione automatica della struttura della GUI attraverso la generazione di input ed eventi e fornisce in output tutto il necessario per ricostruire i dettagli del processo e di tutti gli stati della GUI incontrati durante l'esplorazione. Inoltre supporta diverse tecniche di generazione di test basate su Model Learning e tecniche Random.

Questo progetto è la base di partenza su cui è stata sviluppata la versione Ibrida trattata nel capitolo 4.

3.1 Il processo di ripping

Il Ripper è uno strumento di automazione del testing configurabile che permette l'esplorazione automatica della struttura della GUI attraverso la generazione di input ed eventi considerando anche i possibili eventi scatenabili sul dispositivo come ad esempio rotazione, pressione dei tasti, pressione del tasto menù. Tutti gli aspetti dell'esplorazione (strategia, criterio di terminazione, possibili eventi, possibili input e così via) possono essere configurati per emulare determinati comportamenti dell'utente. Durante l'esplorazione viene tenuta traccia di tutte le eccezioni non gestite (crash) e della sequenza di eventi che le ha generate.

L'algoritmo adoperato dal Ripper per l'esplorazione dell'applicazione sotto test evolve in

diversi passi che è possibile osservare nella Figura 6.

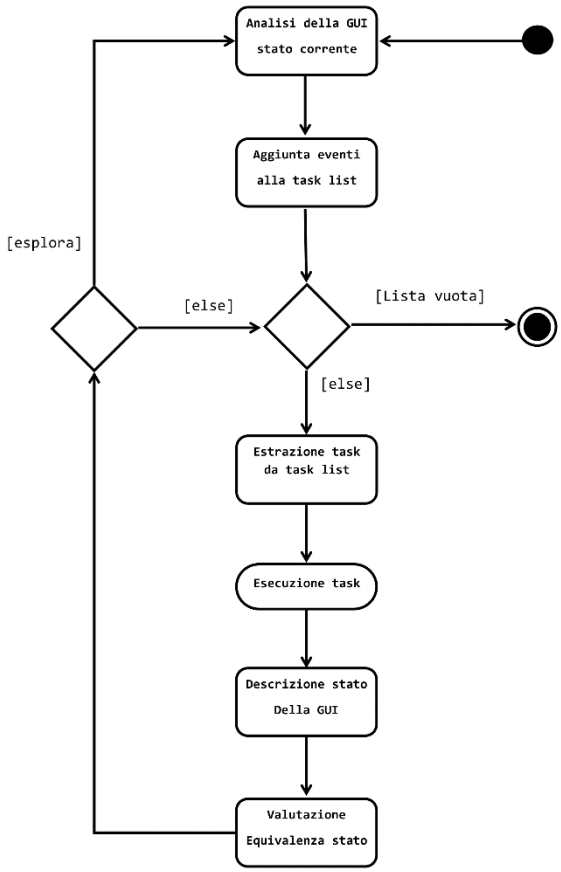


Figura 6 Activity Diagram Ripper

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. [6]

3.2 Concetti di base del modello

Prima di continuare ad addentrarsi sui discorsi riguardo la tecnica e l'architettura del Ripper è bene precisare alcuni dei concetti base per comprenderne meglio il loro significato.

- **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 del file di output.

3.3 Architettura del Ripper

Lo strumento Android GUI Ripper comprende sostanzialmente quattro componenti principali :

- La componente Driver, realizzata come progetto Java eseguibile, la componente GUI Ripper, realizzato come un progetto di test per Android. E' 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.
- La componente Ripper Test Case interagisce con la AUT per eseguire gli eventi ed estrarre le descrizioni della GUI.
- Service invece 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.

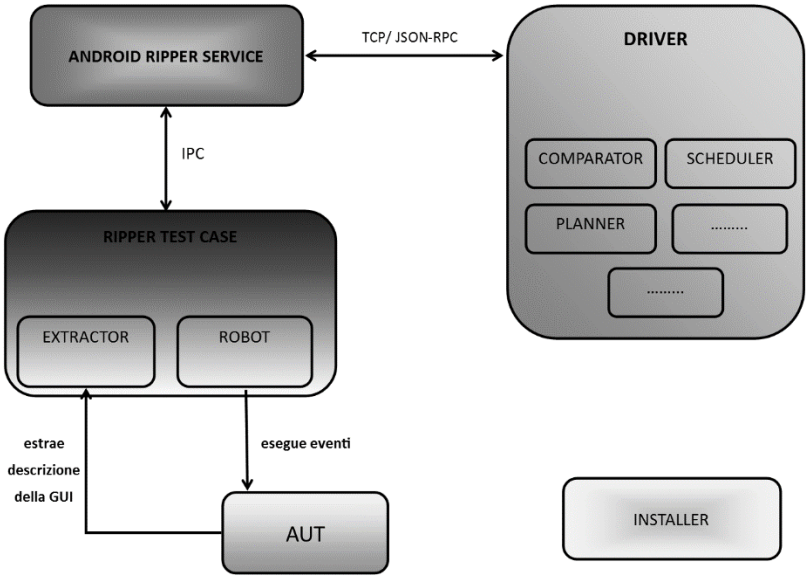


Figura 7 Diagramma concettuale Android Ripper

Analizziamo più in dettaglio le componenti del Ripper e le elenchiamo di seguito per descrivere le loro funzionalità.

- **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** (o 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.

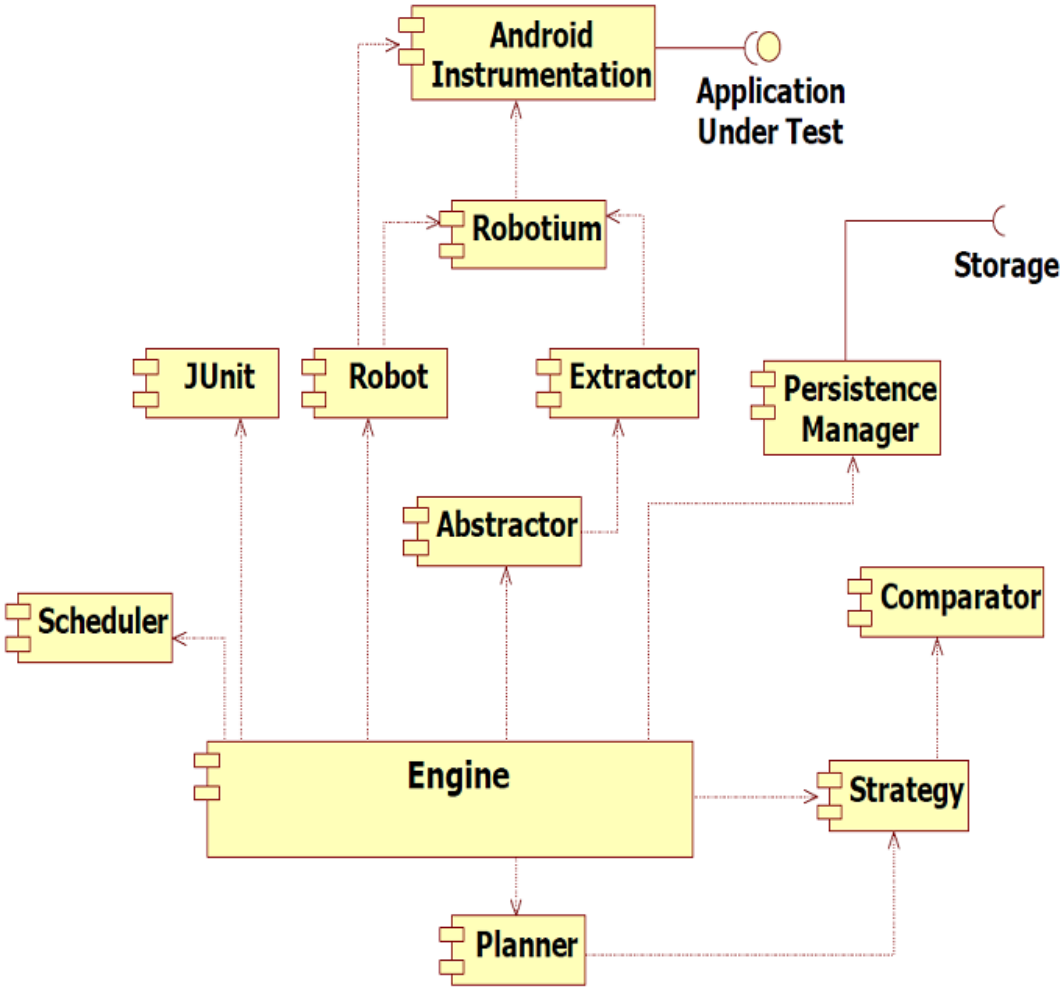


Figura 8 Architettura Android Ripper

3.4 La tecnica di ripping

Andando nel dettaglio, viene qui spiegato come funziona l'algoritmo di ripping sul quale si basa lo strumento di testing automatico. Possono essere distinte le seguenti fasi principali: una fase di *Inizializzazione* in cui le varie componenti del ripper vengono istanziate ed inizializzate; ad essa segue quella di *Setup* in cui 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*)

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. [7]

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
```

3.5 Output del Ripper

L'output del processo di Ripping è costituito dall'intera sessione (Session), ovvero un rapporto dell'esecuzione dei singoli task (trace), ed un file (activities.xml) contenente tutti gli stati della GUI incontrati durante l'esplorazione. I rapporti sono mantenuti in appositi log in formato XML e inoltre vengono generati con Emma i file di coverage relativi all'esecuzione e altri file relativi a JUnit. 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.

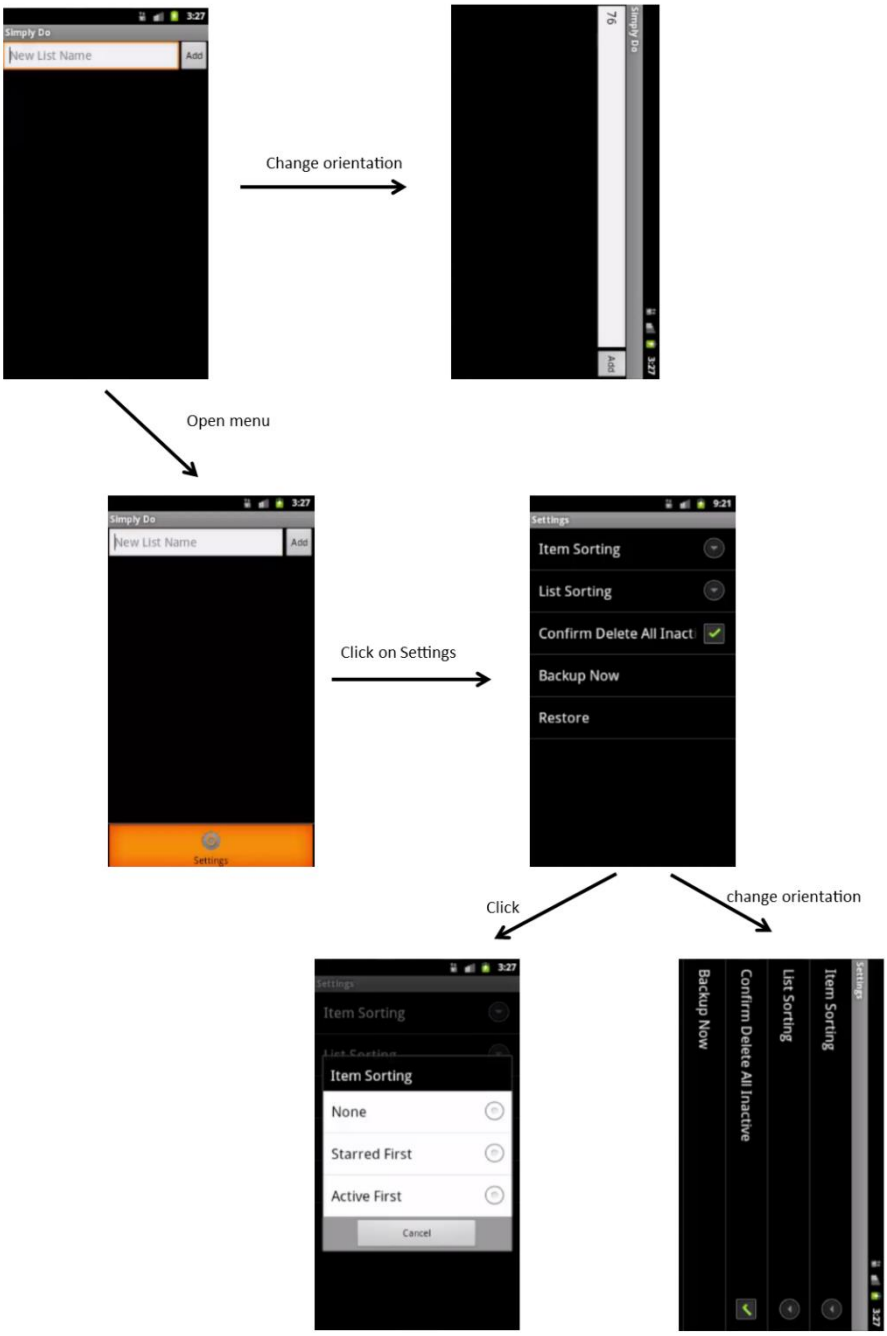


Figura 9 Esempio processo di ripping

Capitolo 4: Android Ripper Ibrido

In questo capitolo si descrive il progetto Android Ripper Ibrido, sviluppato in team per questo lavoro di tesi, e consiste in una "fork" del progetto Android Ripper discusso nel capitolo 3. Tutte le considerazioni fatte per Android Ripper, come ad esempio le operazioni di estrazione e confronto delle activity description, le operazioni di pianificazione, schedulazione ed esecuzione dei task, valgono anche per questa versione. Qui si concentra l'attenzione sulle modifiche apportate e sulla descrizione delle parti essenziali del lavoro svolto che hanno modificato le modalità e le fasi di esecuzione del progetto originale. In questo contesto verrà anche meglio descritta la fase di realizzazione del tool, creato ad hoc, per fornire gli input aggiuntivi al Ripper Ibrido, ovvero i test case generati manualmente. Per lo sviluppo del software è stato utilizzato Eclipse Luna con Android Development Toolkit versione 23.0.4.1468518.

Per le registrazioni di test utente, Robotium Recorder versione 2.1.25 e per la verifica del funzionamento del Ripper Ibrido, il sistema è stato settato utilizzando le impostazioni e i settaggi riportati nell'Appendice B "Installazione e uso Android Ripper Ibrido".

4.1 Perché un modello ibrido

La scelta di realizzare un progetto che si basi su un modello ibrido di generazione di casi di test nasce dal presupposto di cercare di aumentare l'efficacia dei test, riunendo gli automatismi dell'esplorazione sistematica con la peculiarità di test generati manualmente. Quindi si vuole provare ad utilizzare punti di partenza diversi del ripper, provenienti dai test generati manualmente, dando la possibilità al ripper di iniziare l'esplorazione da punti che

non siano quelli dell'applicazione appena avviata.

Grazie ai risultati ottenuti (gli stati visitati) con la prima fase del ripper sistematico, si possono confrontare gli eventuali stati nuovi scoperti dai test utente e ricominciare l'esplorazione sistematica da quei punti.

L'obiettivo è cercare di ottenere dalla componente manuale, dei test con grado di significatività molto alto, in modo da esplorare (magari con una certa conoscenza dell'applicativo), stati della AUT che si trovano più in profondità. Così facendo si potrà consentire al ripper sistematico di essere integrato con nuove scoperte e di poter proseguire nell'esplorazione trovandosi dei passi più avanti.

4.2 Input del processo ibrido

Nella realizzazione del progetto Ripper Ibrido si è scelto di utilizzare gli stessi file di configurazione ("ripper.properties" e "systematic.properties"). All'interno del primo file di configurazione è stata aggiunta una riga relativa al percorso del file di test utente creato attraverso l'uso dello strumento Robotium Recorder per fornirlo come input alla sessione di test del Ripper Ibrido.

Ovviamente questa diviene una precondizione da rispettare relativa all'esecuzione del Ripper Ibrido. Non aggiungendo il path del file precedentemente creato attraverso lo strumento Robotium Recorder, la sessione non può continuare.

4.3 Tool di conversione

Come già accennato, il test registrato manualmente ottenuto dallo strumento Robotium Recorder, produce un file di test che viene rielaborato e inserito in input al processo di Ripping Ibrido.

Un esempio di file prodotto è il seguente:

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

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

	public SimplyDoActivityTest_JK() {
		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: 'kkk'
		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), "kkk");
		[…]
		}
}
```

Tabella 2 Estratto di un test case prodotto da Robotium Recorder

Analizzando la struttura del file prodotto, si è cercato di modificarne la forma, per fare in modo di poterlo utilizzare nella fase iniziale dell'esecuzione del Ripper Ibrido.

Il tool di conversione, denominato tool_jk.jar posizionato nella directory "Release\AndroidRipperInstaller"), effettua le seguenti operazioni:

- verifica la presenza della classe di test specificata dall'utente nel file "ripper.properties"
- crea a partire da essa una nuova classe di test (RobotiumTest.java)
- sposta la classe ottenuta in una directory ben precisa del Ripper Ibrido

```
package it.unina.android.ripper;
import kdk.android.simplydo.SimplyDoActivity;
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;

public class RobotiumTest extends ActivityInstrumentationTestCase2<SimplyDoActivity> {
	private Solo solo;
	public RobotiumTest() {
		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();
	}

	//il numero tc = al numero tagli; da restituire al driver
	public static final int num_tc = 45;

	public void testRun1(Solo solo){
		this.solo = solo;
		// Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
		solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
		solo.sleep(1000);
	}
	public void testRun2(Solo solo){
		this.solo = solo;
		// Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
		solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
		// Enter the text: 'jjj'
		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), "jjj");
		// Click on Add
		solo.clickOnView(solo.getView(kdk.android.simplydo.R.id.AddListButton));
		solo.sleep(1000);
	}
	[…]
	public void testRun45(Solo solo){
	[…]
	}

}
```

Tabella 3. Estratto della classe RobotiumTest.java

Le rielaborazioni principali riguardano il package, modificato per essere coerente con quello del Ripper e il costruttore. I metodi "setUp()" e "tearDown" restano invariati. Il contenuto

del metodo testRun() presente in precedenza, viene suddiviso in diverse sezioni. Il numero di sezioni o "tagli" è variabile rispetto a ciascun test case. Nell'esempio proposto è pari a 45 (*public static final int num_tc = 45;*).

I tagli sono effettuati in corrispondenza di eventi che potenzialmente potrebbero scoprire nuovi stati.

L'identificazione dei tagli avviene in base a particolari stringhe riconosciute:

- "solo.clickOnView": Click su di una specifica View
- "solo.goBack": Simula la pressione del tasto back.
- "solo.waitForActivity": Attende che una Activity attesa venga visualizzata
- "solo.clickInList": Click su un elemento di una lista visualizzata

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 alcune applicazioni con una GUI particolarmente complesse è necessario un aumento di tale valore.

4.4 Funzionamento

L'Android Ripper originale, discusso nel capitolo 3, prevede una singola esecuzione, al termine della quale occorre salvare i dati di output prodotti. Il Ripper Ibrido prevede una esecuzione composta da due fasi distinte, in cui non vi è sovrascrittura dei file prodotti in output tra una fase e l'altra.

La discriminante tra le due fasi è la presenza del file "activities.xml". Nel caso questo sia presente nella cartella di output "model", viene automaticamente eseguita la seconda fase, scatenando una serie di operazioni aggiuntive rispetto a quelle eseguite nella prima fase.

FASE 1. La prima fase corrisponde esattamente all'esecuzione del Ripper originale che prevede la procedura di installazione e generazione degli output (i file di coverage, junit, log, e activities.xml). I primi tre vengono 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 è stato descritto nel capitolo 3 di questa tesi, essendo lo stesso del Ripper della versione base.

FASE 2. La seconda fase invece corrisponde all'esecuzione vera è propria del Ripper Ibrido. Questa prevede una nuova esecuzione del ripper attraverso lo stesso comando lanciato nel Driver "java -jar AndroidRipper.jar s systematic.properties".

Sebbene il comando sia il medesimo della fase 1, il comportamento cambia notevolmente. Infatti in questo caso, il Ripper controlla l'esistenza del file "activities.xml" creato nella prima fase ed effettua una serie di operazioni differenti.

Come è possibile osservare dall'activity diagram in Figura 10 ci sono alcuni cambiamenti che interessano ad esempio il criterio di terminazione della fase 2, che non è più legato alla task list vuota ma è legato alla conclusione dei test case utente.

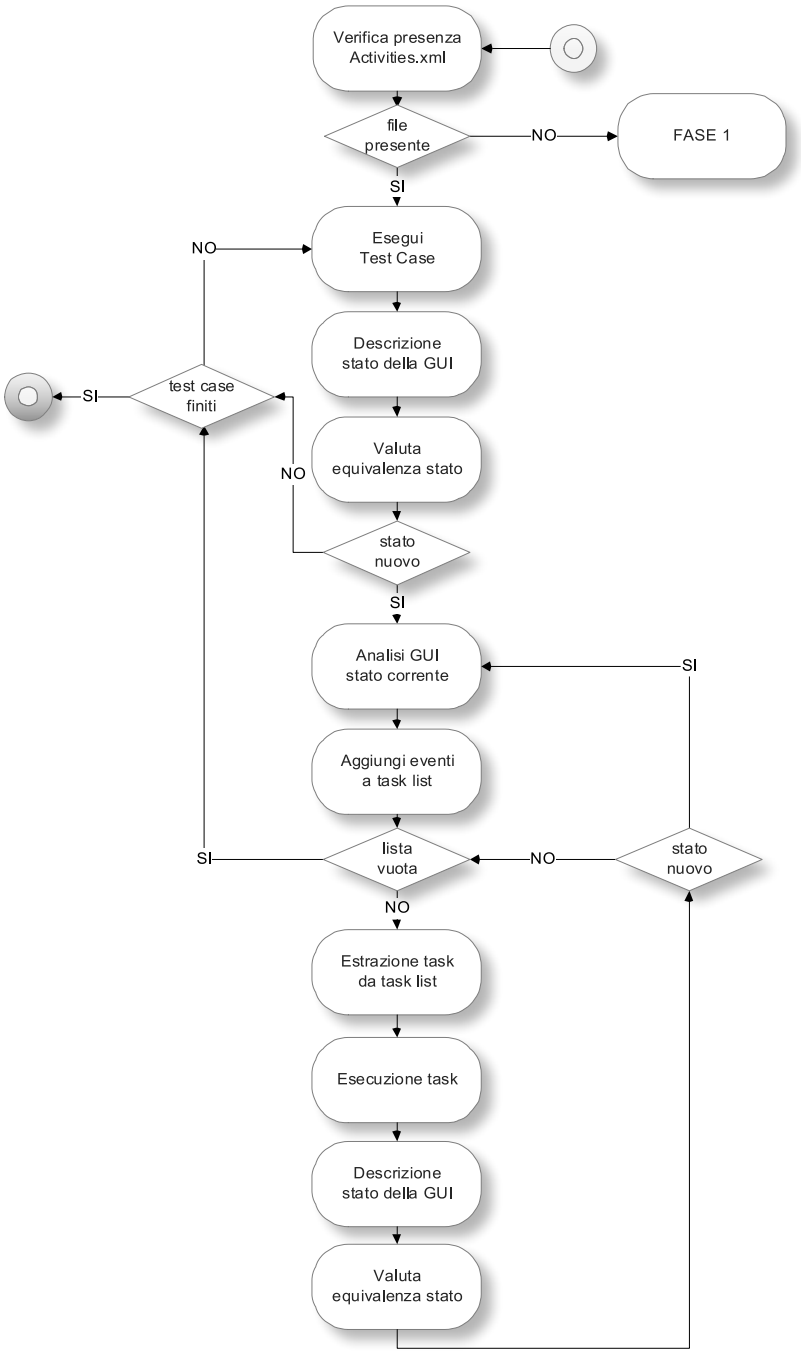


Figura 10. Activity Diagram Ripper Ibrido, 2° fase

In questa seconda fase, le componenti del ripper, schematizzate in Figura 11, scambiano una serie di messaggi. Il Driver invia un messaggio, appositamente creato, per conoscere il numero totale di test case utente che servirà per il criterio di terminazione della intera fase

2. Fin quando tale numero non è raggiunto, il driver continuerà ad inviare un altro tipo di messaggio, creato per l'esecuzione dei test case utente, al componente "Ripper Test Case", il quale, esegue le operazioni e invia l'activity description che viene incapsulata nel messaggio di risposta.

Il Driver, ottenuta l'activity description, la confronta con tutte quelle prelevate dal file "activities.xml". Nel caso sia già presente, ci troviamo nella situazione di "stato equivalente" e si passa direttamente al successivo test, altrimenti se lo stato è "non equivalente", la descrizione viene salvata nel file activities.xml e inizia nuovamente il processo di ripping a partire dallo stato nuovo scoperto. L'applicazione sarà forzata dal Driver a ripartire da questo stato, tramite un apposito messaggio, prima della pianificazione dei task e della conseguente esecuzione degli stessi. E' 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.

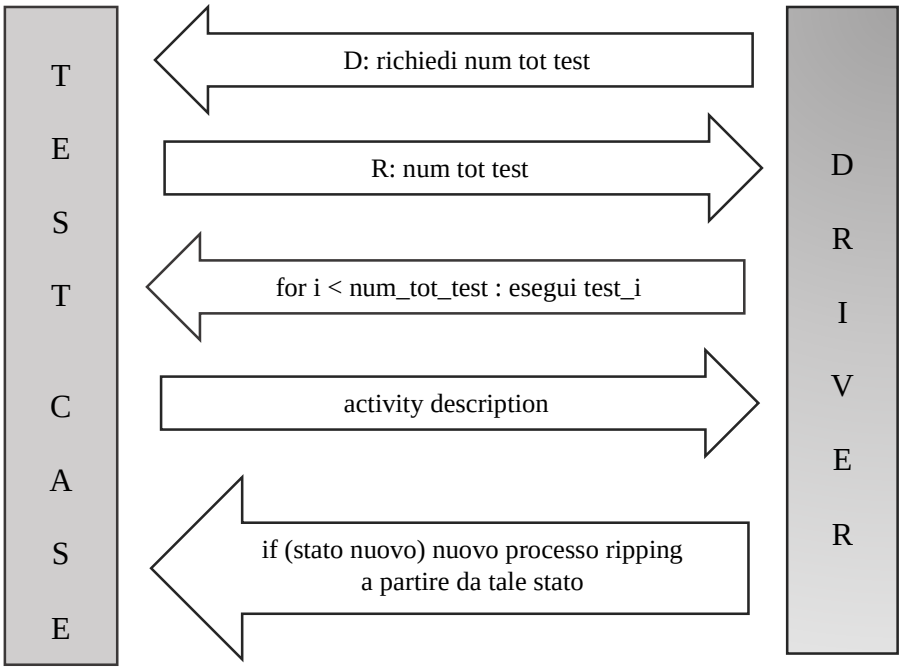


Figura 11. Scambio messaggi tra componenti del Ripper

4.5 Output

L'output prodotto nelle due fasi comprenderà i file di output della prima fase (che non saranno sovrascritti ad eccezione del file activities.xml) e gli output della seconda fase. Nel file activities.xml, durante la prima fase sono raccolti gli stati già esplorati, per effettuare poi il confronto con gli stati estratti al termine dell'esecuzione di ciascun test nella seconda fase del Ripper Ibrido.

Tra gli altri file di output otterremo i file di coverage elaborati per fornire la copertura, i file Junit relativi alle singole sessioni e i file di logcat che vengono rinominati opportunamente per distinguerli dagli output della prima fase. Questi file sono generati solo a seguito della scoperta di un nuovo stato.

Attraverso questi file è possibile ricostruire i vari stati esplorati durante l'esecuzione ed ottenere i risultati relativi alla copertura. Questi dati saranno analizzati nel capitolo seguente e confrontati con le altre tecniche utilizzate durante questo lavoro.

Capitolo 5: Sperimentazione

In questo capitolo, dedicato alla fase di sperimentazione del lavoro svolto, vengono analizzati in dettaglio i risultati ottenuti dal confronto tra le tecniche utilizzate nel corso degli esperimenti. Si motivano questi risultati sia per mezzo di una analisi qualitativa delle tecniche di test utilizzate con l'ausilio dei dati di copertura, sia attraverso analisi statistiche. Vengono inoltre presentate le applicazioni su cui sono stati eseguiti i test, le configurazioni, i metodi e le procedure per raggiungere gli obiettivi prefissati, ossia comparare l'efficacia delle tecniche e validare o invalidare le ipotesi fatte nei confronti dell'esercizio di queste sulle cinque applicazioni coinvolte.

Nel trattare questa fase di sperimentazione si cerca di dare risposta alle research questions, inoltrando l'analisi sui dati alla Teoria delle Ipotesi, che permetterà di verificare alcuni risultati forniti dai confronti effettuati.

5.1 Obiettivi di ricerca

Come già introdotto, obiettivo del lavoro di sperimentazione è confrontare diverse tecniche di generazione di casi di test, al fine di determinare la loro efficacia rispetto alle cinque applicazioni proposte per il sistema operativo Android.

Le tecniche analizzate sono:

- Android Ripper nella sua versione sistematica (Model Learning, ML)
- I soli Test Case (soloTC) generati dal framework Robotium Recorder
- Android Ripper nella sua versione casuale o randomica (Android Ripper Random,

RDM)

- Android Ripper Ibrido (ARI) sviluppato in team durante la preparazione di questa tesi sperimentale e presentato nel capitolo 4.

In questa fase di ricerca, le *research question* a cui si intende rispondere sono:

RQ 1) L'utilizzo della tecnica Android Ripper Ibrido permette di migliorare in efficacia rispetto alla tecnica Solo Test Case?

RQ 2) L'utilizzo della tecnica Android Ripper Ibrido permette di migliorare in efficacia rispetto alla tecnica Android Ripper nella versione sistematica?

RQ 3) L'utilizzo della tecnica Solo Test Case è migliore in termini di efficacia rispetto alla tecnica Model Learning o hanno la stessa copertura?

RQ 4) L'utilizzo della tecnica Android Ripper Ibrido permette di migliorare in efficacia rispetto alla tecnica Android Ripper nella sua versione random?

Per dare risposta a queste domande si procede innanzitutto con l'analisi dei dati di copertura delle diverse tecniche, poi si passerà all'analisi statistica per validare le ipotesi e infine si analizzeranno qualitativamente le diverse tecniche.

5.2 Configurazione sperimentale

Gli esperimenti che hanno riguardato la nostra ricerca sono stati condotti nel dipartimento di Informatica e Sistemistica della facoltà di Ing. Informatica, tramite l'utilizzo di un computer messo a disposizione dal dipartimento con le seguenti caratteristiche:

Sistema Operativo:	Sul Pc inoltre sono installati:
Windows 7 Home Premium SP1 Processore: Intel Core i5-3330 3.00Ghz Memoria Ram: 4,00 Gb	JDK ver. 1.7.0_25; Android SDK Tools ver. 22.3; Android SDK Platform Tools ver. 19.0.2; Android SDK Build Tools ver. 18.1.1; Libraries for Android 2.3.3 and Android 4.0.1; Ant version 1.8.2;

Emulatore

Device (2.7in QVGA 240x320: ldpi)

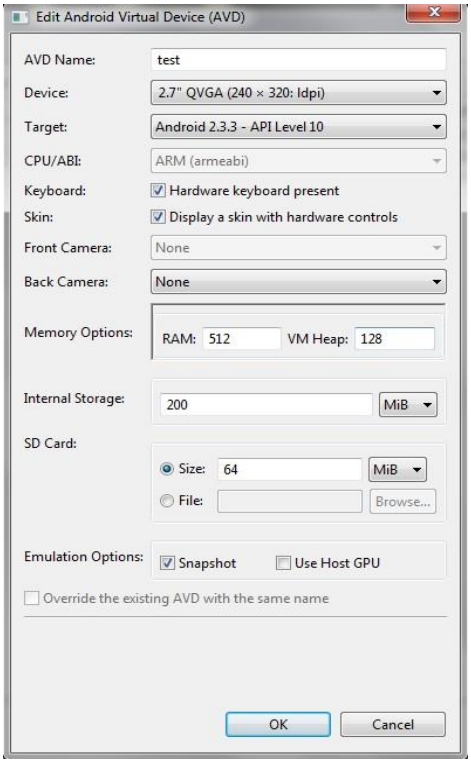
Ram 512 Mb

VM Heap :128

Internal storage: 200 Mb

Sd card: 64 Mb

Android 2.2.3 – API Level 10.



Questa configurazione ha riguardato la raccolta degli output sulle varie versioni del Ripper, mentre per quanto riguarda la generazione dei test "Soli Test Case", questi sono stati ottenuti da registrazioni di diversi utenti con il framework Robotium Recorder, già trattato nel capitolo 2, con l'uso di propri computer.

Questi progetti di test sono stati utilizzati per la raccolta dei dati di copertura relativi anche alla loro esecuzione isolata. Inoltre, come già ampiamente descritto, i test case (soloTC) così ottenuti sono parte integrante del progetto Ripper Ibrido e costituiscono, insieme al Ripper ML, una percentuale di copertura che si aggiunge a tale tecnica.

Le varie percentuali di copertura derivanti dalle diverse tecniche sono state ottenute per mezzo dello strumento Emma, che ha permesso di ricavare il numero di locazioni di codice coperto per le 5 applicazioni sotto test (AUT). Le coperture delle varie versioni del Ripper sono state ottenute automaticamente dall'applicativo, quelle relative ai TC ricavate eseguendo singolarmente i test sulle applicazioni.

5.2.1 Oggetti Sperimentali : le Applicazioni

Le applicazioni sottoposte ai test delle diverse tecniche sono cinque e sono disponibili su internet o su Google Play. Qui si elencano per darne una breve descrizione:

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 della propria automobile.

	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

Nella tabella sono riassunte le classi, i metodi, i blocchi, le activities e il numero di righe di codice che compongono le cinque applicazioni.

5.2.2 Le Tecniche

In questo paragrafo si evidenziano gli aspetti che riguardano le varie tecniche confrontate, prima di addentrarsi nell'analisi dei dati che riguarda la loro efficacia.

Model Learning (ML)

Android Ripper consiste nella versione originale sistematica del Ripper (Model Learning, ML) che è stato discusso nel capitolo 3 e che procede con un'esplorazione che avviene in ampiezza in maniera sistematica. Le tecniche ML sono in grado di creare dinamicamente un modello della AUT e di generare casi di test sulla base di questo modello. Esse si basano su tecniche di reverse engineering che esplorano in modo dinamico l'applicazione e, contestualmente, ne deducono il modello che descrive il comportamento.

I Soli Test Case (soloTC)

I Soli Test Case generati dal framework Robotium Recorder sono frutto di registrazioni ottenute da diversi utenti, cinque per l'esattezza, con discreta conoscenza delle AUT. I progetti di test considerati comprendono una classe principale, all'interno del package di test, che è possibile rieseguire anche senza il supporto del framework con le quali sono state create.

Android Ripper Random (RDM)

Le tecniche random sono in grado di generare casi di test composti da sequenze di input generate in modo casuale. Queste tecniche sono spesso utilizzate in test black-box e cercano di simulare test composti di sequenze di eventi utente e di sistema scelti a caso.

Android Ripper Ibrido (ARI)

Android Ripper Ibrido è stato sviluppato in team durante la preparazione di questa tesi sperimentale e racchiude il risultato di due fasi di esecuzione: la prima fase che rappresenta il modello originale ML del ripper ed una seconda fase, consecutiva alla prima, in cui i test case utente (la componente manuale "umana") sono forniti in input per cercare di ottenere una maggiore efficacia di questo modello ibrido.

5.3 Variabili e metriche

In questo paragrafo viene illustrato cosa è stato fatto variare negli esperimenti condotti e cosa invece è rimasto costante, attraverso l'individuazione sia di variabili indipendenti, che

dipendenti. Le *independent variables*, cioè le variabili che sono variate o manipolate durante la sperimentazione, sono ovviamente rappresentate dalle tecniche, mentre come *dependent variable*, (quelle che subiscono gli effetti dei cambiamenti agiti sulle variabili indipendenti), è stata considerata la copertura (o anche coverage).

Per quanto riguarda la misurazione dell'efficacia abbiamo adottato come metrica di copertura il numero di classi coperte, dei metodi e delle locazioni (LOC) coperte. Ognuno di questi parametri è in grado di esprimere la capacità delle suite di test nell'eseguire il codice sorgente dell'applicazione sotto test.

In questo studio si discuterà l'efficacia misurata principalmente in termini di copertura di LOC (line %).

L'efficacia delle test suite sperimentali ottenute adottando una specifica tecnica (T), è misurata come:

$$\varepsilon(T) = \frac{\text{LOC Coverage raggiunte da T}}{Totale\ LOC\ AUT} * 100$$

5.4 Metodologia di analisi

L'analisi sui dati sperimentali in questa fase di ricerca è stata condotta analizzando i risultati raccolti in forma tabellare per la valutazione delle percentuali di copertura delle differenti tecniche per poi passare ad effettuare il confronto tra queste mediante analisi statistica con assunzioni di natura insiemistica e analitica. Infine, scendendo più nel dettaglio, si è passati ad una analisi qualitativa che permettesse di estrapolare i dettagli sulle coperture delle tecniche adottate.

Nei paragrafi che seguono si farà riferimento e si esamineranno alcune analisi statistiche, in modo da ottenete misure sintetiche delle caratteristiche più importanti sulle distribuzioni dei

dati e l'interpretazione dei risultati ottenuti con i test applicati, allo scopo di discutere:

a) Le ipotesi che si intendono verificare, in relazione alle research question;
b) Le caratteristiche statistiche sulle varie distribuzioni dei dati applicate alle differenze delle tecniche.

Per discutere di questo, si farà uso della teoria di verifica delle ipotesi.

5.4.1 Teoria di verifica delle ipotesi

La verifica di ipotesi è una metodologia per affrontare i problemi di scelta tra due (o più) ipotesi ed ha inizio con la formulazione del sistema di ipotesi sottoposto a verifica.

Il sistema consiste in due ipotesi contrapposte, l'ipotesi nulla (H0) che è, in genere, l'ipotesi sottoposta a verifica che si vorrebbe rifiutare e l'ipotesi alternativa (H1) specificata come ipotesi opposta a quella nulla e che rappresenta la conclusione raggiunta quando l'ipotesi nulla è rifiutata.

E' importante chiarire sin da ora che l'ipotesi nulla H0 deve essere rifiutata solamente se esiste l'evidenza che la contraddice. Essa non è necessariamente vera quando i dati campionari (eventualmente pochi) non sono tali da contraddirla.

Lo strumento utilizzato per affrontare problemi di verifica d'ipotesi viene chiamato test statistico. Quest'ultimo rappresenta il mezzo utile per verificare quanto i dati a disposizione siano o meno a favore delle nostre ipotesi. Si può definire test statistico una procedura che, sulla base di dati campionari consente di decidere se è ragionevole respingere l'ipotesi nulla H0 (ed accettare implicitamente l'ipotesi alternativa H1) oppure se non esistono elementi sufficienti per respingerla. La teoria della verifica di ipotesi fornisce una regola su cui basare il processo decisionale e si basa sul calcolo della probabilità di ottenere un dato risultato campionario nel caso in cui l'ipotesi nulla sia vera. Questo risultato viene ricavato determinando prima la distribuzione campionaria della statistica di interesse (statistica test), quindi calcolando il valore assunto per il particolare campione considerato. [13]

Secondo uno schema valido per molti test statistici, il procedimento logico che seguiremo comprende diverse fasi:

1) Stabilire l'ipotesi nulla (H0) e l'eventuale ipotesi alternativa (H1).

2) Scegliere il test più appropriato per saggiare l'ipotesi nulla H0, secondo le finalità della ricerca e le caratteristiche statistiche dei dati.
3) Specificare il livello di significatività.
4) Trovare la distribuzione di campionamento del test statistico nell'ipotesi nulla H0, di norma fornita da tabelle.
5) Stabilire la zona di rifiuto (che negli esercizi di norma sarà prefissata al 5% indicato con la simbologia $\alpha = 0.05$)
6) Calcolare il valore del test statistico sulla base dei dati sperimentali (con approccio p-value questo verrà calcolato automaticamente dagli strumenti che introdurremo a breve).
7) Sulla base del confronto tra il p-value e il livello di significatività, trarre le conclusioni:
    - se il p-value risulta superiore al valore di alpha (α) prefissato, concludere che non è possibile rifiutare l'ipotesi nulla H0
    - se il p-value risulta inferiore al valore di alpha (α) prefissato, è possibile rifiutare l'ipotesi H0 e quindi implicitamente accettare l'ipotesi alternativa H1

Per il livello di significatività, la prassi didattica consiglia di decidere che un esperimento è significativo solo quando la probabilità stimata con il test è inferiore al valore critico convenzionale prefissato, di norma scelto tra $\alpha = 0.05$, $\alpha = 0.01$, $\alpha = 0.001$.
Per effettuare le valutazioni delle nostre analisi risultano molto importanti due parametri: il p-value e l'effect size.

P-value

Negli ultimi anni, anche grazie all'ampia diffusione di pacchetti statistici e fogli elettronici, si è affermato un altro approccio alla verifica di ipotesi: l'approccio del p-value. Il p-value rappresenta la probabilità di osservare un valore della statistica test uguale o più estremo del

valore che si calcola a partire dal campione, quando l'ipotesi H0 è vera.

Il p-value è anche chiamato livello di significatività osservato e il vantaggio offerto consiste nel fatto che non serve andare a consultare le tavole della Normale, della t-Student, ecc. per decidere. Tutto quello che si deve fare è confrontare il valore del p-value con il livello di significatività α che abbiamo fissato in precedenza.

Effect Size

Un altro parametro che è stato valutato nell'analisi statistica è la *grandezza dell'effetto,* dall'inglese *Effect Size* (*ES*), che riguarda la valutazione della significatività dei risultati ottenuti nella ricerca, ovvero la grandezza reale dell'effetto riscontrato nei dati della sperimentazione.

Gli indici di dimensione dell'effetto sono una misura quantitativa che fornisce indicazioni utili sulla reale importanza di un effetto sperimentale. In pratica servono per determinare quanto un effetto è significativo, importante, e possono aiutare in ampia misura il ricercatore a decidere se un risultato è significativo e importante a livello pratico.

Una stima approssimativa delle dimensioni dell'effetto viene fatta attraverso delle scale. Nella nostra analisi verrà denominato "*effect r"* assumendo che:

effect r = .8 large effect

effect r = .5 medium effect

effect r = .2 small effect

5.4.1.1 Impostazioni delle ipotesi

In genere, sono possibili due diverse impostazioni dell'ipotesi alternativa H1. E' possibile verificare:

1. se esiste una differenza nelle frequenze relative tra i due gruppi, senza predeterminare quale dei due debba essere il maggiore (o il minore): si tratta di un test bilaterale o a due code (H1: $\theta_1 \neq \theta_2$)
2. se un gruppo ha una frequenza relativa significativamente maggiore (oppure minore) e in questo caso si parla di un test unilaterale o a una coda con la possibilità di confrontare H0: $\theta_1 < \theta_2$ contro H_1: $\theta_1 > \theta_2$ oppure H0: $\theta_1 > \theta_2$ contro H1: $\theta_1 < \theta_2$

In ognuno di questi due ultimi casi ad una coda, viene a priori rifiutata come non accettabile od illogica, la possibilità alternativa a quella proposta.

La distinzione tra test a due code e test a una coda non è solamente una questione di logica ma ha effetti pratici importanti: da essa dipende la distribuzione delle probabilità ed il valore critico per rifiutare l'ipotesi nulla, come chiarisce il grafico seguente.

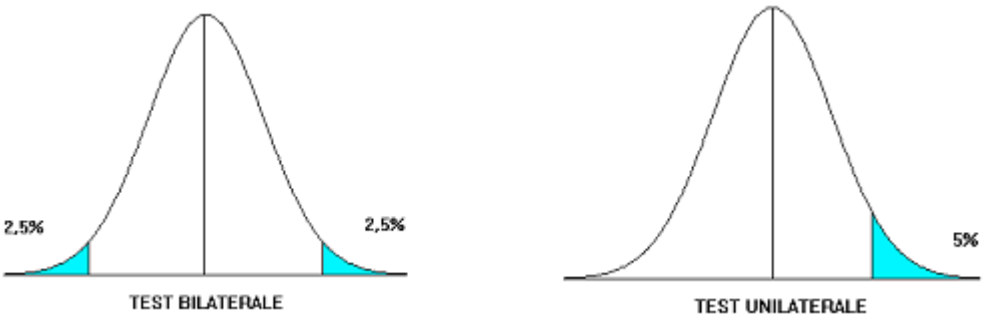


Figura 12. Test Bilaterale e Test Unilaterale

Scegliendo la probabilità del 5%, in un test a due code si hanno due zone di rifiuto collocate ai due estremi, ognuna con un'area di 2,5%. In un test a una coda, si ha una sola zona di rifiuto, con un'area del 5 %. Esistono maggiori probabilità di rifiutare l'ipotesi nulla quando si effettua un test ad una coda, che quando si effettua un test a due code. Anche nella rappresentazione grafica, risulta evidente in modo visivo che, alla stessa probabilità totale, in un test unilaterale il valore critico è minore di quello in un test bilaterale.

Quindi l'insieme di valori ottenibili con il test forma la distribuzione campionaria della statistica test e può essere divisa in due zone:

- la zona di rifiuto dell'ipotesi nulla, detta anche regione critica, che corrisponde ai valori collocati agli estremi della distribuzione, quelli che hanno una probabilità piccola di verificarsi, quando l'ipotesi nulla H0 è vera;
- la zona di accettazione di H0, che comprende i restanti valori;

Nella figura che segue viene rappresentato questo concetto.

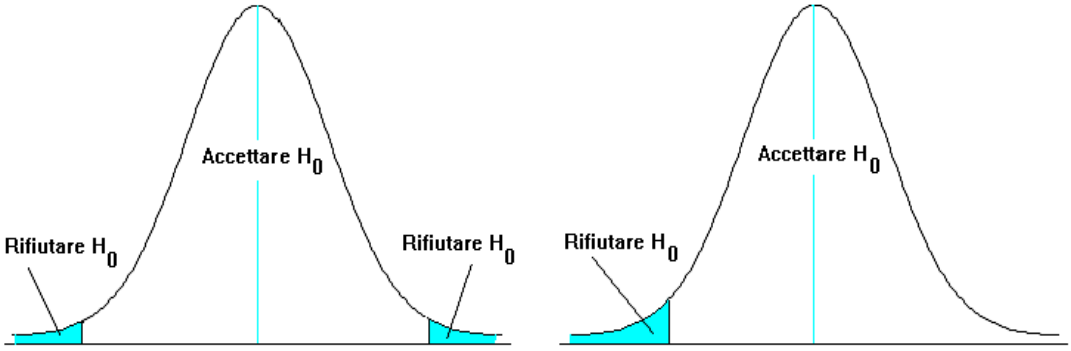


Figura 13. Zone accettazione e di rifiuto

Per una corretta comprensione dei concetti utilizzati, è importante evidenziare che, accettando queste conclusioni, è possibile commettere un errore.

Esistono due tipi di errore a cui andiamo in contro:

Errore di I tipo: Consiste nel rifiutare l'ipotesi nulla H0, quando in realtà essa è vera.

Errore di II tipo: Consiste nel non rifiutare (o accettare) l'ipotesi nulla H0, quando in realtà essa è falsa.

		Realtà	
		H0 è vera	**H0 è falsa**
Conclusione del Test	**Si accetta H0**	**Corretta**	**Errore II tipo**
	Si rifiuta H0	**Errore I tipo**	**Corretta**

Tabella 4. Tipi di Errore

In statistica, non è possibile eliminare questi due tipi di errore; è possibile solamente ridurre la loro frequenza al minimo e conoscere con precisione la probabilità con la quale avvengono. Solo conoscendo la probabilità di sbagliare, è possibile scegliere in modo corretto. [13]

5.4.1.2 Condizioni di normalità e tipologia di test

Le distribuzioni che consideriamo spesso sono distribuzioni statistiche note, come la Normale (Gaussiana), ma potrebbe non esserlo.

A livello teorico alcuni test sono più adatti di altri in certe condizioni per il loro comportamento asintotico. Possiamo suddividere i test in parametrici e non parametrici.

Per quanto riguarda i test parametrici, questi sono utilizzati quando la distribuzione risulta normale, condizione che verificheremo attraverso i test di normalità di Shapiro-Wilk.

Il test di Shapiro-Wilk è un test per la verifica di ipotesi statistiche ed è considerato uno dei test più potenti per la verifica della normalità, soprattutto per piccoli campioni.

Nei confronti delle distribuzioni normali i dati saranno trattati con test parametrici: noi utilizzeremo il metodo di t-Student (T-Test) che presuppone, tra le altre cose, la normalità delle distribuzioni.

Utilizzeremo tecniche non parametriche quando gli assunti teorici relativi alle condizioni di validità della distribuzione normale non sono dimostrati con sicurezza.

In condizioni di incertezza sull'esistenza delle condizioni richieste da un test parametrico, come spesso succede quando si dispone di pochi dati, una soluzione sempre più diffusa suggerisce una duplice strategia:

- utilizzare un test appropriato di statistica parametrica,
- convalidare tali risultati mediante l'applicazione di un test non parametrico equivalente.

Se le probabilità stimate con i due differenti metodi risultano simili, sono confermate la robustezza del test parametrico e la sua sostanziale validità anche in quel caso.

Il test non parametrico quindi può servire per confermare i risultati ottenuti con quello parametrico e come misura preventiva contro eventuali obiezioni sulla normalità dei dati.

La statistica non parametrica si dimostra particolarmente utile nell'apprendimento dei processi logici, in riferimento alla formulazione delle ipotesi, alla stima delle probabilità mediante il test e all'inferenza sui parametri a confronto.

Nel caso di test non parametrici utilizzeremo il test di Wilcoxon che non richiede la normalità e la simmetria dei dati.

L'impostazione classica del test per ranghi di Wilcoxon, detto più semplicemente anche test

T di Wilcoxon, permette di verificare se la tendenza centrale di una distribuzione si discosta in modo significativo da un qualsiasi valore prefissato di confronto. Come termine di confronto utilizza la mediana. I motivi della scelta della mediana sono diversi: è meno influenzata dai valori anomali; se la distribuzione fosse normale, media e mediana coinciderebbero, quindi le inferenze sulla mediana possono essere estese alla media.

Nei casi in cui la distribuzione dei dati è spesso lontana dalla normalità, il test T di Wilcoxon è preferibile al test parametrico e assicura condizioni di validità più generali, senza perdere in potenza-efficienza (a volte aumentandola).

5.4.2 Grafici

Per evidenziare le caratteristiche di una tabella o di un semplice elenco di dati, ci sono stati utili anche rappresentazioni grafiche o semigrafiche. Quelli che vengono più avanti mostrati e che serviranno per evidenziare l'andamento delle nostre distribuzioni sono chiamati diagrammi di Box-and-Whisker (scatola-e-baffi), o anche detti più rapidamente box-plot. Consistono in un metodo grafico molto diffuso e reso di uso corrente da molti programmi, in cui la quantità di informazioni che viene fornita è molto elevata.

Queste informazioni vengono riassunte in cinque valori contenute nella distribuzione: la mediana, il primo e il terzo quartile, il valore minimo e quello 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.

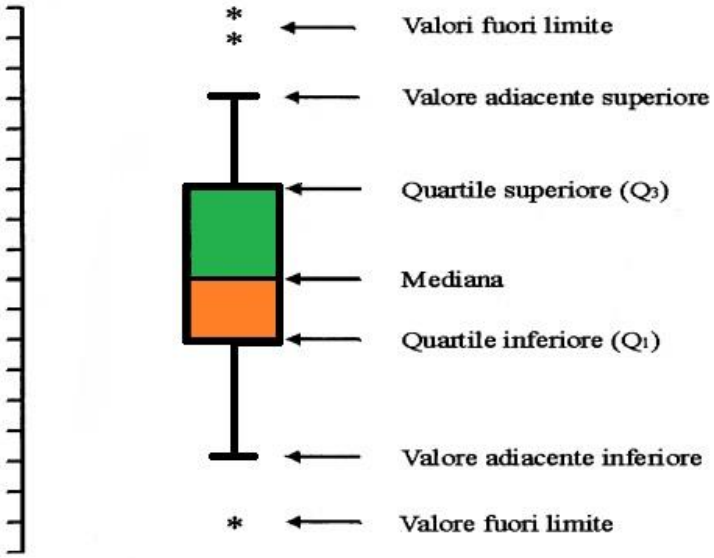


Figura 14. Esempio di BoxPlot

Volendo descrivere alcune caratteristiche di un box plot, si osserva la 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 una 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.

5.4.3 Strumenti per i test statistici

Per la raccolta dei dati di analisi, per la loro sintesi in forma tabellare e per utilizzare i test statistici su di essi sono stati utilizzati i seguenti strumenti:

Excel 2013: programma prodotto da Microsoft, dedicato alla produzione ed alla gestione dei fogli elettronici. È parte della suite di software di produttività personale Microsoft Office.

Real-Statistics: è uno strumento per fare analisi statistiche utilizzando Excel. Estende le funzionalità statistiche integrate di Excel e strumenti di analisi dei dati in modo da poter facilmente eseguire una vasta gamma di analisi statistiche. [15]

R ver. 3.2.0: R-Project for Statistical Computing è una suite integrata di servizi software per la manipolazione dei dati, per il calcolo statistico e offre una grande varietà di tecniche per la visualizzazione grafica. Si tratta di un progetto della R Foundation disponibile come software libero, sotto i termini della Free Software Foundation 's GNU General Public License sotto forma di codice sorgente. Disponibile su una vasta gamma di piattaforme UNIX e sistemi similari (incluso FreeBSD e Linux), Windows e MacOS. [16]

5.5 Analisi Statistica

I dati di copertura raccolti vengono presentati ora sotto forma tabellare e sono relativi alle AUT utilizzate per la sperimentazione. La Tabella 5 riportata di seguito mostra le percentuali di copertura ottenute (line, %), dalle diverse tecniche e dai diversi utenti.

Si ricorda che le percentuali della tecnica ML risultano le stesse per ogni utente perché la tecnica sistematica risulta la stessa, mentre i soli Test Case mostrano una copertura diversa per i diversi utenti. La Tecnica ARI che racchiude la tecnica ML e quella solo TC rappresenta la copertura completa ottenuta per ciascun utente. La tecnica Random invece riporta la copertura del test casuale che risulta la stessa poiché deriva da una sperimentazione con 12 sessioni che si è arrestata solo quando da tutte le sessioni si è ottenuta, a parità di numero di eventi, la stessa identica copertura. Per questo motivo è possibile considerare il suo valore come una costante.

Per avere una visione più generale dei dati di copertura, qui si presentano i dati in maniera raggruppata in una singola tabella che servirà per le analisi statistiche.

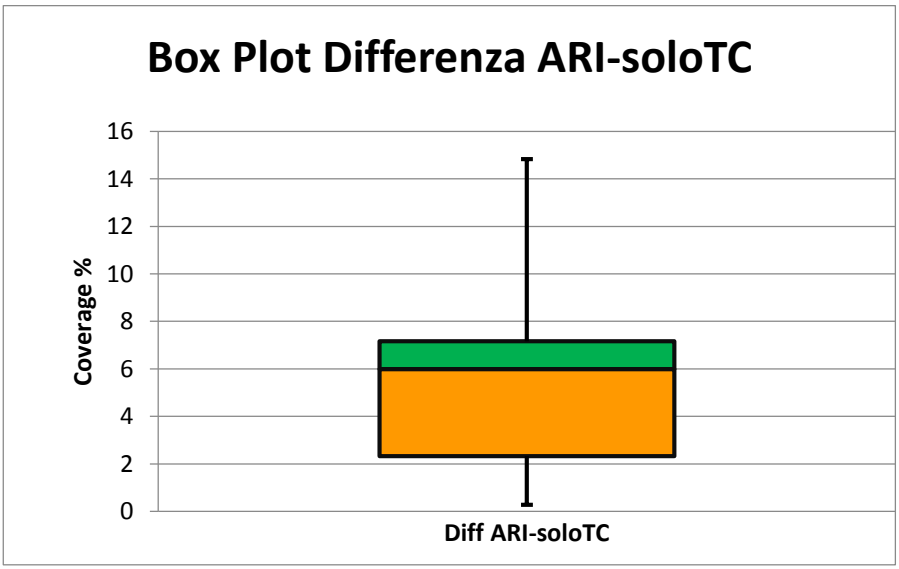
		COVERAGE % - TECNICHE			
APP	Tecniche / Utenti	Model Learning	Solo TC	ARI	Random
SimplyDo	**Dev**	32,36	65,30	72,31	85,10
	Ale	32,36	79,33	81,48	85,10
	Fed	32,36	76,53	82,52	85,10
	Nic	32,36	80,64	83,20	85,10
	PTra	32,36	47,78	55,38	85,10
TippyTipper	**Dev**	57,42	79,74	86,11	87,46
	Ale	57,42	82,80	87,98	87,46
	Fed	57,42	84,52	88,38	87,46
	Nic	57,42	81,87	86,75	87,46
	PTra	57,42	81,87	86,75	87,46
Trolly	**Dev**	34,12	77,86	78,13	80,88
	Ale	34,12	75,05	75,60	80,88
	Fed	34,12	75,85	76,13	80,88
	Nic	34,12	68,54	74,81	80,88
	PTra	34,12	70,19	70,74	80,88
MunchLife	**Dev**	80,22	75,76	85,11	86,90
	Ale	80,22	79,62	89,02	86,90
	Fed	80,22	78,53	88,48	86,90
	Nic	80,22	75,27	90,11	86,90
	PTra	80,22	75,27	90,11	86,90
FillUp	**Dev**	12,93	59,84	67,00	77,56
	Ale	12,93	34,82	36,68	77,56
	Fed	12,93	69,86	72,19	77,56
	Nic	12,93	62,65	68,80	77,56
	PTra	12,93	62,49	68,97	77,56

Tabella 5. Coverage % delle Tecniche

Le considerazioni che seguiranno fanno parte dell'analisi statistica effettuata sui dati raccolti che vengono modellati come variabili aleatorie per considerare le distribuzioni ottenute dalle differenze delle coperture sulle varie applicazioni in base alle diverse tecniche utilizzate. Si sottoporranno queste distribuzioni ad una serie di test statistici e attraverso la teoria di verifica di ipotesi, si cercherà di dare risposta alle research question proposte ad inizio capitolo.

5.5.1 Differenza tra tecniche ARI e Solo TC

Ad una prima stima dei dati, anche solo visivamente, sembrerebbe che la tecnica ARI sia, anche se di poco, migliore della tecnica solo TC. Questo si spiega con il fatto che i soli test case sono inclusi nella tecnica ibrida e che quest'ultima racchiude anche una minima parte sistematica. Per dare prova di questo, però, si procede con l'analisi della differenza aritmetica tra le due tecniche, sottoponendola ai test statistici.

% Diff ARI-soloTC	Box plot e Normalità
7,01 2,15 5,99 2,56 7,60 6,37 5,18 3,85 4,87 4,87 0,27 0,55 0,27 6,26 0,55 9,35 9,40 9,95 14,84 14,84 7,16 1,86 2,33 6,15 6,47	

Shapiro-Wilk Test

	Differenza ARI-soloTC
W	0,9274
p-value	0,075703
alpha	0,05
normal	yes

> shapiro.test(diff ARI-soloTC)

Shapiro-Wilk normality test

data: diff ARI-soloTC
W = 0.92743, p-value = 0.07581

Considerando il grafico della distribuzione e i valori ottenuti attraverso il test di normalità di Shapiro-Wilk con i due strumenti, RealStatistics e R, la distribuzione esaminata risulta normale.

Sulla base di questo risultato si può pensare di effettuare il test parametrico T-Test (t-Student) sulla differenza aritmetica delle due tecniche.

Come descritto nel paragrafo 5.4.1, si sceglie una ipotesi nulla (H0) ed una ipotesi alternativa (H1) del tipo: H0: La distribuzione data dalla differenza tra ARI e soloTC è uguale a zero. H1: ipotesi opposta.

$$H0: ARI - soloTC = 0;\ \ H1: ARI - soloTC \neq 0$$

La tecnica soloTC, come già detto, è inclusa in ARI, quindi ciò che si vuole provare è solo che tra le due vi è differenza in termini di copertura. Si passa quindi a verificare le ipotesi attraverso il test statistico.

T Test: One Sample with RealStatistics

SUMMARY			**Alpha**	**0,05**			
Count	*Mean*	*Std Dev*	*Std Err*	*t*	*df*	*Cohen d*	***Effect r***
25	5,63	4,013071	0,802614	7,012817	24	1,402563	**0,81978**

T TEST			Hyp Mean	0	
	p-value	*t-crit*	*lower*	*upper*	*sig*
One Tail	1,49E-07	1,710882			yes
Two Tail	**2,99E-07**	2,063899	3,972072	7,285101	yes

T Test: One Sample with R

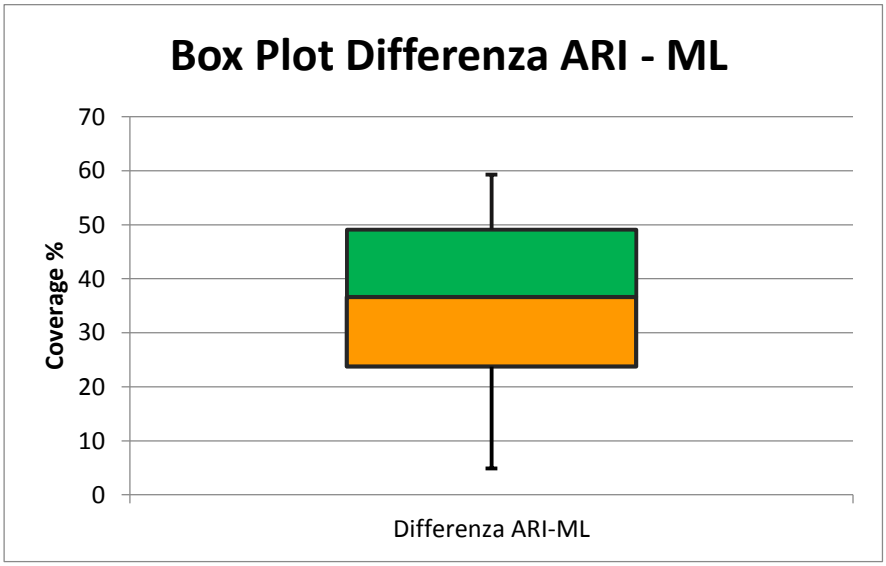
```
> t.test(diff ARI-soloTC)

data:  diff ARI-soloTC
t = 7.014, df = 24, p-value = 2.981e-07
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 3.973634 7.287166
sample estimates:
mean of x
        56.304
```

Si assume un test bilaterale (a due code). Il valore di p-value risulta piccolo (2,99E-07 < 0.05) rispetto ad alpha. Dal risultato di questo test si evince che è possibile rifiutare H0 e accettare l'ipotesi alternativa H1, che le due tecniche, ARI ed TC, non sono uguali e quindi

la loro differenza è diversa da zero. Il valore dell'*effect r* (0,81978) risulta *large*, il che indica un livello di effetto del test grande.

5.5.2 Differenza tra tecniche ARI e ML

Come secondo paragone si considera la differenza tra le tecniche ARI e ML. Come nel primo caso anche la tecnica ML è inclusa nella prima. Quindi si limiteranno le ipotesi alla verifica della sola differenza in termini aritmetici.

% Diff ARI-ML	Box plot e Normalità
39,95 49,12 50,16 50,84 23,02 28,69 30,56 30,96 29,33 29,33 44,01 41,48 42,01 40,69 36,62 4,89 8,80 8,26 9,89 9,89 54,07 23,75 59,26 55,87 56,04	

Shapiro-Wilk Test

	Differenza ARI-ML
W	0,936275
p-value	0,121393
alpha	0,05
normal	yes

```
> shapiro.test(diff ARI-ML)

Shapiro-Wilk normality test

data:  diff ARI-ML
W = 0.93626, p-value = 0.1213
```

Come nel caso precedente, stando al grafico della distribuzione ed ai valori ottenuti attraverso il test di normalità di Shapiro-Wilk con i due strumenti, RealStatistics e R, la

distribuzione esaminata risulta normale. Sulla base di questo risultato è possibile effettuare il test parametrico T-Test (t-Student) sulla differenza aritmetica delle due tecniche. Si sceglie una ipotesi nulla (H0) ed una ipotesi alternativa (H1).

H0: La distribuzione data dalla differenza tra ARI e ML è uguale a zero. H1: ipotesi opposta

$$H0: ARI - ML = 0; \quad H1: ARI - ML \neq 0;$$

T Test: One Sample with RealStatistics

SUMMARY			**Alpha**	**0,05**			
Count	*Mean*	*Std Dev*	*Std Err*	*t*	*df*	*Cohen d*	***Effect r***
25	34,30	16,719895	3,3439791	10,257408	24	2,0514817	**0,90236**

T TEST			Hyp Mean	0	
	p-value	*t-crit*	*lower*	*upper*	*sig*
One Tail	1,49E-10	1,710882			yes
Two Tail	**2,99E-10**	2,063898	27,398927	41,202195	yes

T Test: One Sample with R

```
> t.test(diff ARI-ML)

data:  diff ARI-ML
t = 10.257, df = 24, p-value = 2.988e-10
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 27.3979 41.2013
sample estimates:
mean of x
        342.996
```

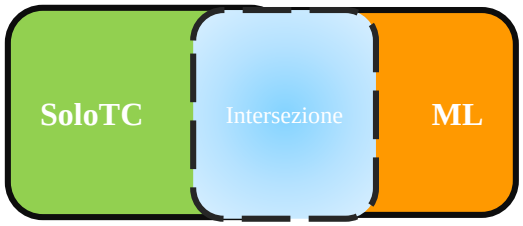
Anche in questo caso si sta assumendo un test bilatero (a due code). Il valore di p-value risulta piccolo (2,99E-10 < 0.05) rispetto ad alpha. Dal risultato di questo test si evince che è possibile rifiutare H0 e accettare l'ipotesi alternativa H1, la cui condizione afferma che le due tecniche, ARI ed TC, non sono uguali e quindi la loro differenza sia diversa da zero. Il valore dell'*effect r* (0,90236) risulta *large*, il che indica un livello di effetto del test ancora più grande del precedente.

5.5.3 Differenza tra Unione – Intersezione delle tecniche SoloTC e ML

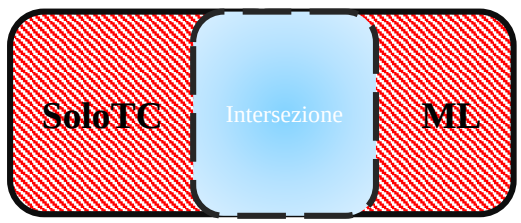
Prima di concentrarsi sulle differenze riguardanti queste due tecniche, è importante fare un'osservazione su come sono stati raccolti i dati relativi ad esse. Per verificare le ipotesi è stata presa in considerazione, non solo in questo caso, la differenza in termini di insiemistici riguardante le coperture delle due tecniche. Utilizzando uno strumento a nostra disposizione è stato possibile ricavare tramite i risultati di coverage, l'unione e l'intersezione delle coperture delle due tecniche rispetto alle AUT.

Questo ha permesso di analizzare più a fondo le diversità delle tecniche.

Nel confronto tra queste due tecniche, si fa riferimento per prima cosa alla loro differenza rispetto agli insiemi di copertura, considerando come distribuzione la differenza tra l'unione delle due tecniche sottratta all'intersezione. Successivamente le due tecniche verranno analizzate sotto il profilo della differenza aritmetica, ma questa volta con ipotesi diverse allo scopo di verificare quale delle due risulti maggiore o minore.



La tecnica Solo Test Case e la tecnica Model Learning dal punto di vista insiemistico.

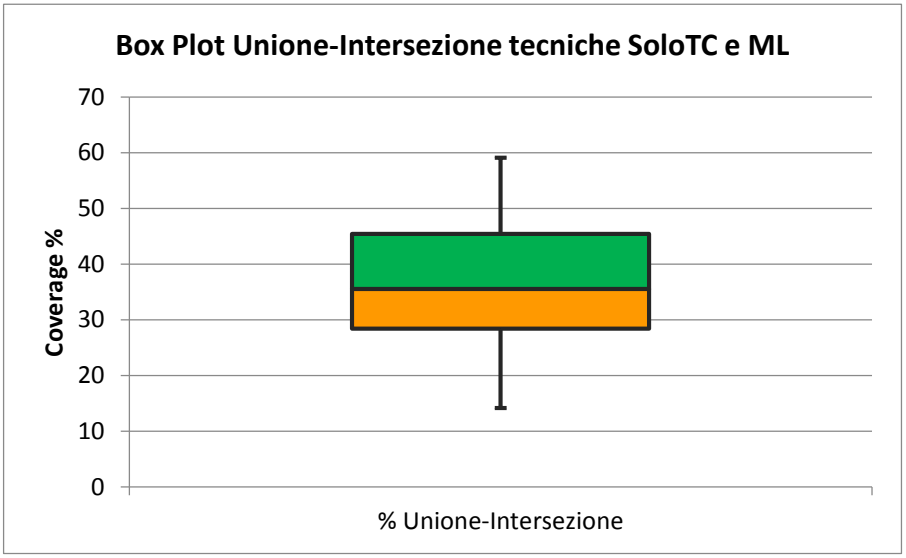


Area in rosso: Differenza Unione-Intersezione in termini insiemistici

Intersezione

Differenza Unione-Intersezione

Tabella 6 Rappresentazione Insiemi di copertura

% Diff Unione-Intersezione	Box plot e Normalità
34,04 47,78 45,43 49,41 28,42 34,13 35,54 34,73 33,23 33,23 44,23 41,48 42,31 44,51 36,54 14,13 17,93 17,93 24,46 24,46 49,91 24,48 59,10 52,43 52,27	

Shapiro-Wilk Test	
	Differenza Unione-Intersezione
W	0,936275
p-value	0,121393
alpha	0,05
normal	yes

Dal grafico della distribuzione e dai valori ottenuti attraverso il test di normalità di Shapiro-Wilk, la distribuzione esaminata risulta normale. Si passa ad effettuare il test parametrico T-Test (t-Student) sulla differenza di copertura delle due tecniche, risultato dell'Unione meno l'intersezione.

Si sceglie una ipotesi nulla (H0) ed una ipotesi alternativa (H1).

H0: La distribuzione data dalla differenza tra Unione e Intersezione è uguale a zero.

H1: ipotesi opposta

$$H0: \text{Unione} - \text{Intersezione} = 0; \quad H1: \text{Unione} - \text{Intersezione} \neq 0$$

T Test: One Sample with RealStatistics							
SUMMARY			**Alpha**	**0,05**			
Count	*Mean*	*Std Dev*	*Std Err*	*t*	*df*	*Cohen d*	***Effect r***
25	36,88	12,0282	2,405639138	15,3325875	24	3,066517	**0,952558**

T TEST			Hyp Mean	0	
	p-value	*t-crit*	*lower*	*upper*	*sig*
One Tail	3,36E-14	1,710882			yes
Two Tail	**6,72E-14**	2,063898	31,91967732	41,8496676	yes

T Test: One Sample with R

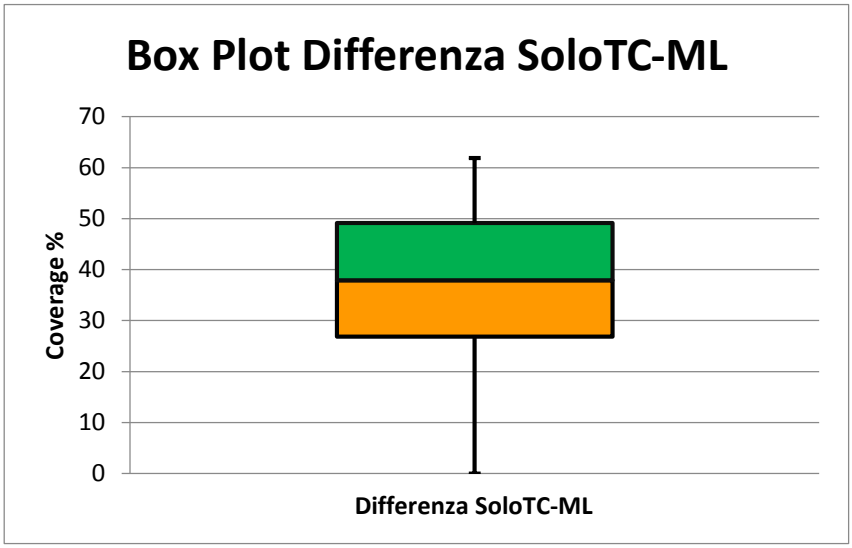
```
> t.test(diff Unione-Intersezione)

data:  diff Unione-Intersezione
t = 15.332, df = 24, p-value = 6.732e-14
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 31.91932 41.84948
sample estimates:
mean of x
        368.844
```

Si tratta di un test bilaterale. Il valore di p-value risulta piccolo (6,72E-14 < 0.05) rispetto ad alpha. Dal risultato di questo test si evince che si può rifiutare H0 e accettare l'ipotesi alternativa H1, cioè che l'unione sottratta all'intersezione delle due tecniche, soloTc ed ML, non sono uguali e quindi la loro differenza è diversa da zero.

Il valore dell'*effect r* (0,95255) risulta *large*, il che indica un livello di effetto del test grande. Come ci si aspettava, il risultato del test, prova che l'area rappresentata in Tabella 6 risulta molto diversa da zero.

5.5.4 Differenza tecniche SoloTC e ML

Adesso si procede con l'esame della distribuzione associata alla differenza aritmetica tra le coperture delle due tecniche.

% Diff SoloTC-ML	Box plot e Normalità
32,94 46,97 44,18 48,28 15,42 22,32 25,39 27,11 24,45 24,45 43,74 40,93 41,73 34,42 36,07 -4,46 -0,60 -1,68 -4,95 -4,95 46,91 21,89 56,93 49,72 49,56	

Shapiro-Wilk Test	
	Differenza SoloTC-ML
W	0,902536
p-value	0,020842
alpha	0,05
normal	no

Dal grafico della distribuzione e dai valori ottenuti attraverso il test di normalità di Shapiro-Wilk, questa volta la distribuzione esaminata non risulta normale. In questo caso si adopera un test non parametrico di Wilcoxon sulla differenza aritmetica di copertura delle due tecniche. Si sceglie come ipotesi nulla (H0) ed una ipotesi alternativa (H1).

H0: La distribuzione data dalla differenza tra soloTC e ML è minore di zero.

H1: ipotesi opposta

H0: soloTC - ML < 0; H1: soloTC - ML > 0

Wilcoxon Signed-Rank Test for a Single Sample con Real-Statistics

sample median	32,94
pop median	0
count	25
# unequal	25
T+	15
T-	310
T	15

	one tail	two tail
alpha	**0,05**	
mean	162,5	
std dev	37,1651718 7	
z-score	-3,96876948 5	
effect r	**0,79375389**	
T-crit	100,868732 3	89,157601 7
p-value	**3,61224E-05**	**7,2245E-05**
sig	yes	yes

T-crit	100	89
sig	yes	yes

Wilcoxon Signed-Rank Test for a Single Sample in R

```
> wilcox.test(diff soloTC-ML, exact=FALSE)

 Wilcoxon signed rank test with continuity correction

data:  diff soloTC-ML
V = 310, p-value = 7.632e-05
alternative hypothesis: true location is not equal to 0
```

Wilcoxon Signed-Rank Test (with continuity correction**) for a Single Sample alternative=GREATER**

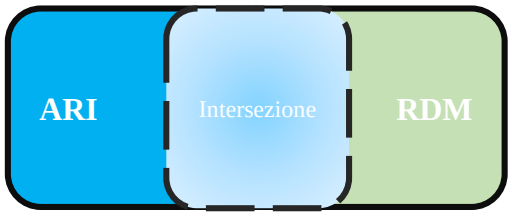
```
> wilcox.test(diff soloTC-ML, exact=FALSE, alternative="greater")

data:  diff soloTC-ML
V = 310, p-value = 3.816e-05
alternative hypothesis: true location is greater than 0
```

Questa volta si tratta di un test unilatero. Il p-value 3.816e-05 risulta piccolo rispetto ad alpha, quindi è possibile rifiutare l'ipotesi nulla H0: soloTC-ML<0 e accettare l'ipotesi alternativa H1:">0", quindi in pratica si può rifiutare che la differenza sia minore di zero e alternativamente si può dedurre che soloTC>ML.

5.5.5 Differenza Unione-Intersezione tecniche ARI e RDM

Come nel caso visto in precedenza, si esamineranno le distribuzioni delle differenze delle due tecniche prima dal punto di vista dell'Unione-Intersezione, poi dal punto di vista aritmetico.

Innanzitutto si considera la differenza tra l'Unione e l'Intersezione delle tecniche ARI e RDM.



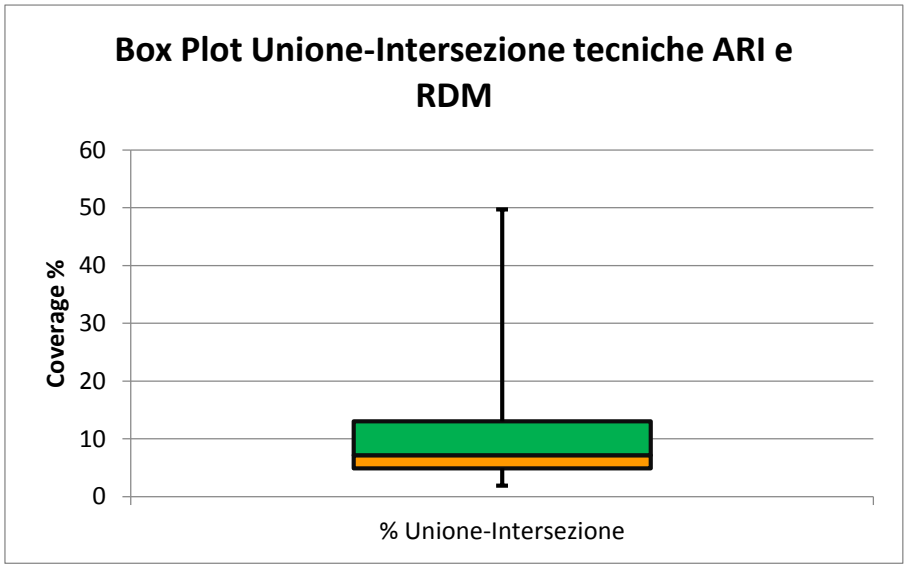
La tecnica Android Ripper Ibrido e la tecnica Random dal punto di vista insiemistico.

Intersezione

Area in rosso: Differenza Unione-Intersezione in termini insiemistici

Differenza Unione-Intersezione

Tabella 7 Rappresentazione Insiemi di copertura

% Diff Unione-Intersezione	Box plot e Normalità
12,80 3,59 2,58 1,87 29,82 5,91 4,10 3,50 5,31 5,31 3,85 6,32 5,77 7,14 11,26 4,89 8,70 11,41 13,04 13,04 16,31 49,70 15,52 14,84 14,66	

Shapiro-Wilk Test

	Differenza % Unione-Intersezione ARI-RDM
W	0,710018
p-value	9,96983E-06
alpha	0,05
normal	no

> shapiro.test(diff *Unione-Intersezione ARI-RDM*)

Shapiro-Wilk normality test

data: diff *Unione-Intersezione ARI-RDM*
W = 0.71002, **p-value = 9.97e-06**

Dal grafico della distribuzione e dai valori ottenuti attraverso il test di normalità di Shapiro-Wilk, la distribuzione esaminata non risulta normale. In questo caso si adopera il test non parametrico di Wilcoxon sulla differenza tra Unione ed Intersezione delle due tecniche.

Si sceglie una ipotesi nulla (H0) ed una ipotesi alternativa (H1).

H0: La distribuzione data dalla differenza tra Unione e Intersezione è uguale a zero.

H1: ipotesi opposta

H0: Unione - Intersezione = 0; H1: Unione - Intersezione ≠ 0

Wilcoxon Signed-Rank Test for a Single Sample con Real-Statistics

sample median	7,14
pop median	0
count	25
# unequal	25
T+	0
T-	325
T	0

	one tail	two tail
alpha	**0,05**	
mean	162,5	
std dev	37,16517187	
z-score	-4,372373161	
effect r	**0,874474632**	
T-crit	100,8687323	89,157601 7
p-value	**6,14516E-06**	**1,23E-05**
sig	yes	yes

T-crit	100	89
sig	yes	yes

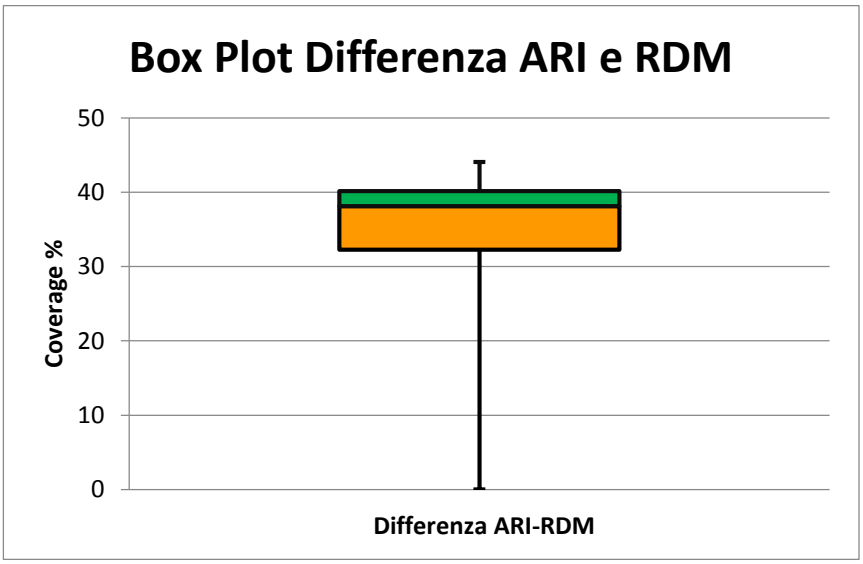
Wilcoxon Signed-Rank Test for a Single Sample in R

```
> wilcox.test(diff % Unione-Intersezione ARI-RDM, exact=FALSE, alternative="two.sided")

 Wilcoxon signed rank test with continuity correction

data:  diff % Unione-Intersezione ARI-RDM
V = 325, p-value = 1.305e-05
alternative hypothesis: true location is not equal to 0
```

Il test è bilaterale. Il valore di p-value risulta piccolo (1,23E-05 < 0.05) rispetto ad alpha. Questo equivale a dire che è possibile rifiutare H0 e considerare che ARI ed RDM non siano uguali. Come nel caso discusso precedentemente sugli insiemi di copertura, anche in questo test ci conferma che l'area in rosso in Tabella 7 non è nulla.

5.5.6 Differenza tecniche ARI e RDM

Continuando ad esaminare le due tecniche ARI ed RDM, ci si concentra sulla distribuzione della differenza ottenuta aritmeticamente.

% Diff ARI-RDM	Box plot e Normalità
-12,79 -3,62 -2,58 -1,90 -29,72 -1,35 0,52 0,92 -0,71 -0,71 -2,75 -5,27 -4,75 -6,07 -10,14 -1,79 2,12 1,58 3,21 3,21 -10,56 -40,88 -5,37 -8,76 -8,59	

Shapiro-Wilk Test	
	Diff. % ARI-RDM
W	0,72788
p-value	1,77E-05
alpha	0,05
normal	no

> shapiro.test(diff *ARI-RDM*)

Shapiro-Wilk normality test

data: diff *ARI-RDM*
W = 0.72788, **p-value = 1.77e-05**

Per i valori ottenuti attraverso il test di normalità di Shapiro-Wilk, la distribuzione esaminata non risulta normale. In questo caso si adopera un test non parametrico di Wilcoxon sulla differenza tra le due tecniche. Si sceglie l'ipotesi nulla (H0) ed una ipotesi alternativa (H1)

H0: La distribuzione data dalla differenza tra ARI e RDM è minore di zero.

H1: ipotesi opposta

$$H0: ARI - RDM < 0;\ \ H1: ARI - RDM > 0$$

Wilcoxon Signed-Rank Test for a Single Sample con Real-Statistics

sample	
median	-2,75
pop	
median	0
count	25
# unequal	25
T+	45
T-	310
T	45

	one tail	two tail
alpha	**0,05**	
mean	162,5	
std dev	37,165171	
z-score	-3,1615621	
effect r	**0,632312**	
T-crit	100,8687	89,1576017
p-value	**0,000784627**	**0,00156925**
sig	yes	yes

T-crit	100	89
sig	yes	yes

Wilcoxon Signed-Rank Test for a Single Sample in R

```
> wilcox.test(diff ARI - RDM,
exact=FALSE,
alternative="two.sided")

 Wilcoxon signed rank test with
 continuity correction

data:  diff ARI - RDM
V = 45, p-value = 0.001642
alternative hypothesis: true location is
not equal to 0
```

Wilcoxon Signed-Rank Test for a Single Sample alternative=GREATER

```
> wilcox.test(diff ARI - RDM, exact=FALSE, alternative="greater")

      Wilcoxon signed rank test with continuity correction

data:  diff ARI - RDM
V = 45, p-value = 0.9993
alternative hypothesis: true location is greater than 0
```

Siamo nel caso di test unilatero (ad una coda). Il valore di p-value risulta uguale a (0.9993>0.05) maggiore di alpha. Questo non ci consente di rifiutare l'ipotesi nulla ARI-RDM<0. Con effect r di (0.6323), effetto "medio" si deduce che RDM, anche se di poco, sia maggiore di ARI.

TABELLA RIASSUNTIVA ANALISI STATISTICA CONFRONTO TRA TECNICHE
Ax : coppie su cui la differenza viene valutata aritmeticamente
Ix : coppie in cui si considera la differenza tra Unione ed Intersezione

Coppie di Tecniche	Ipotesi		Shapiro		Test				Possibilità di rifiutare HP nulla
	HP nulla	HP alternativa	Norm.	p-value	tipo	p-value	Effect r	Size	
A1: coppia (ARI, SoloTC)	$H0^1$: ARI-TC= 0	H1: ARI-TC ≠0	yes	0,075703	t-Test	2,99E-07	0,81978	large	yes
A2: coppia (ARI, ML)	$H0^2$: ARI-ML= 0	H1: ARI-ML ≠0	yes	0,121393	t-Test	2,99E-10	0,902365	large	yes
I1: Unione-Intersezione (Un-Int) di SoloTC e ML	$H0^3$: UN-INT = 0	H1: UN-INT ≠0	yes	0,670079	t-Test	6,73E-14	0,952558	large	yes
A3: coppia (SoloTC, ML)	$H0^4$: TC-ML<0	H1: TC-ML >0	no	0,020842	Wilcoxon	3,82E-02	0,793754	large	yes
I2: Unione-Intersezione (Un-Int) di ARI e RDM	$H0^6$: UN-INT=0	H1: UN-INT ≠0	no	9,969E-06	Wilcoxon	1,23E-05	0,874475	large	yes
A5: coppia (ARI, RDM)	$H0^7$: ARI-RDM<0	H1: ARI-RDM>0	no	1,770E-05	Wilcox	0.9993	0,632312	medium	no

5.6 Analisi qualitativa

In questo paragrafo ci si concentra sugli aspetti legati alle differenze in termini qualitativi riscontrate nelle diverse tecniche di generazione di casi di test sulle cinque applicazioni oggetto di studio.

L'analisi prevede di ricercare e ipotizzare una categorizzazione sulle tipologie di righe di codice coperte, dandone delle motivazioni. Queste saranno di carattere generale e cercheranno di spiegare quali tipologie di righe di codice vengono coperte da una tecnica invece che da un'altra, o eventualmente da nessuna.

Allo scopo di identificare le differenze in base alle varie tipologie che è possibile riscontrare, ad esempio tra tecniche sistematiche e tecniche manuali o randomiche, si considera l'unione delle coperture ottenute sia nella tecnica soloTC, che nella tecnica ARI. Queste ultime si avvalgono di casi di test che risultano variabili in dipendenza del fatto che vengono ottenute da utenti diversi, e quindi da una componente umana diversa.

Risulta dunque lecito pensare di unificare i test così ottenuti per poter avere una apprezzabile comparazione delle diverse componenti in gioco.

Nella tabella che segue, vengono schematizzate le caratteristiche delle tecniche utilizzate.

ML	Tecnica Model Learning. Esplorazione in ampiezza sistematica
UTC	Unione delle coperture ottenute dai test delle 5 componenti manuali (SoloTC)
UARI	Unione delle tecniche Android Ripper Ibrido per le 5 componenti manuali, più componente sistematica
RDM	Tecnica Random. Esplorazione randomica dell'applicazione

Tabella 8 Tecniche per analisi qualitativa

5.6.1 Confronto ML e UTC

In questo confronto vengono mostrate le differenze qualitative ottenute tra la componente sistematica del Ripper e i test case ottenuti dalle diverse registrazioni utente. Questo

paragone evidenzierà una prima differenza tra due approcci differenti applicate alle AUT. Dai dati riportati nelle tabelle saranno esclusi i dettagli sulle singole classi dei package per motivi legati alla grande quantità di dati che non possono essere elencati in questa stesura. Queste ovviamente saranno considerate per i risultati di comparazione.

Di seguito l'elenco in forma tabellare per le coperture delle due tecniche.

SimplyDo

OVERALL STATS SUMMARY - SimplyDo

total packages	total executable files	total classes	total methods	total executable lines
1	15	46	246	1281

Copertura tecnica ML

OVERALL COVERAGE SUMMARY – SimplyDo

name	class, %	method, %	block, %	line, %
all classes	91% (42/46)	43% (105/246)	35% (1915/5523)	**32% (414,5/1281)**

Copertura tecnica UTC

OVERALL COVERAGE SUMMARY - SimplyDo

name	class, %	method, %	block, %	line, %
all classes	96% (44/46)	87% (213/246)	84% (4645/5523)	**82% (1054,7/1281)**

Tippy Tipper

OVERALL STATS SUMMARY - TippyTipper

total packages	total executable files	total classes	total methods	total executable lines
5	13	42	225	999

Copertura tecnica ML

OVERALL COVERAGE SUMMARY – TippyTipper

name	class, %	method, %	block, %	line, %
all classes	90% (38/42)	65% (146/225)	56% (2386/4253)	**57% (573,6/999)**

Copertura tecnica UTC

OVERALL COVERAGE SUMMARY – TippyTipper

name	class, %	method, %	block, %	line, %
all classes	98% (41/42)	85% (191/225)	88% (3727/4253)	**87% (865,7/999)**

Trolly

OVERALL STATS SUMMARY - Trolly

total packages	total executable files	total classes	total methods	total executable lines
3	5	19	64	364

Copertura tecnica ML

OVERALL COVERAGE SUMMARY - Trolly

name	class, %	method, %	block, %	line, %
all classes	68% (13/19)	47% (30/64)	31% (582/1888)	**34% (124,2/364)**

Copertura tecnica UTC

OVERALL COVERAGE SUMMARY - Trolly

name	class, %	method, %	block, %	line, %
all classes	89% (17/19)	81% (52/64)	80% (1505/1888)	**80% (290,4/364)**

MunchLife

OVERALL STATS SUMMARY - MunchLife

total packages	total executable files	total classes	total methods	total executable lines
1	2	10	28	184

Copertura tecnica ML

OVERALL COVERAGE SUMMARY – MunchLife

name	class, %	method, %	block, %	line, %
all classes	90% (9/10)	93% (26/28)	76% (642/841)	**80% (147,6/184)**

Copertura tecnica UTC

OVERALL COVERAGE SUMMARY – MunchLife

name	class, %	method, %	block, %	line, %
all classes	100% (10/10)	96% (27/28)	85% (713/841)	**86% (157,5/184)**

FillUp

OVERALL STATS SUMMARY - FillUp

total packages	total executable files	total classes	total methods	total executable lines
1	57	105	669	3807

Copertura tecnica ML

OVERALL COVERAGE SUMMARY - FillUp

name	class, %	method, %	block, %	line, %
all classes	22% (23/105)	16% (110/669)	11% (2050/18096)	**13% (492,2/3807)**

Copertura tecnica UTC

OVERALL COVERAGE SUMMARY - FillUp

name	class, %	method, %	block, %	line, %
all classes	86% (90/105)	77% (515/669)	75% (13639/18096)	**75% (2867,6/3807)**

Alla luce di questi risultati ottenuti e mostrati nelle tabelle precedenti è possibile fare diverse considerazioni in merito alla differenza sostanziale delle due tecniche. Innanzitutto si nota l'incremento significativo di copertura ottenuta con la tecnica UTC per tutte le AUT. Ad esempio si passa dal valore di copertura (line%) della tecnica ML per SimplyDo che è del 32% (414,5/1281) al valore di copertura della tecnica UTC che è del 82% (1054,7/1281). Questo risultato scaturisce dal semplice fatto che la configurazione del Ripper sistematico non produce degli eventi significativi sull'applicazione come quelli scatenati dalla tecnica manuale.

Scendendo nei dettagli possiamo osservare, per esempio, che nella classe principale "SimplyDoActivity.java", con la tecnica ML non viene coperto il metodo "addItem()" associato al click sul Bottone "AddItemButton", che consente l'inserimento di un elemento in lista. Di conseguenza non vengono coperti i metodi che effettuano altre operazioni sugli elementi in lista.

Codice relativo alla tecnica ML - SimplyDo

```
142         Button addItem = (Button)findViewById(R.id.AddItemButton);
143         addItem.setOnClickListener(new View.OnClickListener() {
144             @Override
145             public void onClick(View v)
146             {
147                 addItem();
148             }
149         });
150
```

Codice relativo alla tecnica UTC - SimplyDo

```
142         Button addItem = (Button)findViewById(R.id.AddItemButton);
143         addItem.setOnClickListener(new View.OnClickListener() {
144             @Override
145             public void onClick(View v)
146             {
147                 addItem();
148             }
149         });
150
```

Questo comportamento ovviamente si ripercuote sull'utilità di altre classi, facendo in modo che la copertura relativa associata a queste sia compromessa.

Continuando il confronto, fa eccezione la classe "Settings.java", che viene coperta per il 79% (42/53) da entrambe le tecniche. Questo risultato è dettato dal fatto che la componente sistematica ha esplorato la maggior parte della lista dei settaggi e così è successo per la componente umana.

Come è facile notare dai dati tabellari, la tecnica manuale, complessivamente, ha maggiore copertura, ma la tecnica ML riesce a coprire anche parti di codice che UTC non copre. Basti vedere l'esempio riportato successivamente per comprendere quali tipologie di codice

possono essere coperte dalla tecnica sistematica e che UTC non è stata in grado di coprire.

Codice relativo alla tecnica ML - SimplyDo

```
290 if (savedInstanceState==null)
291    {
292     Log.d(L.TAG, "SimplyDoActivity.onCreate()");
293    }
294  else
295     {
296      Log.d(L.TAG, "SimplyDoActivity.onCreate() with a supplied state");
297
298      Integer listId = (Integer)savedInstanceState.getSerializable("currentListId")
;
299       if(listId != null)
300         {
301          ListDesc listDesc = dataViewer.fetchList(listId);
302          if(listDesc != null)
303            {
304              listSelected(listDesc, false);
305             }
306           else
307              {
308   Log.w(L.TAG, "SimplyDoActivity.onCreate(): savedInstanceState had bad list ID");
309              }
310          }
311    }
```

Codice relativo alla tecnica UTC - SimplyDo

```
290  if (savedInstanceState==null)
291     {
292       Log.d(L.TAG, "SimplyDoActivity.onCreate()");
293     }
294  else
295      {
296       Log.d(L.TAG, "SimplyDoActivity.onCreate() with a supplied state");
297
298     Integer listId = (Integer)savedInstanceState.getSerializable("currentListId");
299       if(listId != null)
300         {
301          ListDesc listDesc = dataViewer.fetchList(listId);
302          if(listDesc != null)
303             {
304              listSelected(listDesc, false);
305              }
306           else
307             {
308   Log.w(L.TAG, "SimplyDoActivity.onCreate(): savedInstanceState had bad list ID");
309             }
310          }
311     }
```

Come è possibile notare dalla porzione di codice presentata, alcune condizioni IF-ELSE, non vengono raggiunte dalla tecnica UTC. La componente manuale ha coperto una sola condizione, mentre la componente sistematica, esplorando in ampiezza, ha esaminato le due possibili soluzioni. Questa particolare situazione può rivelarsi molto importante quando si andrà a considerare le due tecniche che ora si stanno confrontando, riunite in una sola tecnica UARI.

Passando a considerare le restanti applicazioni, in generale è possibile notare come per esempio in TippyTipper, la tecnica ML non esplori correttamente la classe "Total.java".

In questa activity, utilizzata per il calcolo del conto a ristorante, non viene effettuata la pressione sulla GUI dei tasti *R.id.btn_SplitBill, R.id.btn_round_down, R.id.btn_round_up, R.id.btn_TipAmount1, R.id.btn_TipAmount2, R.id.btn_TipAmount3*, cosa che può essere dovuta a particolari configurazioni del ripper. Con la tecnica UTC tutto questo non avviene; l'activity Total viene esplorata, con conseguente pressione dei tasti inclusi in essa.

Ancora, in questa applicazione possiamo rilevare che ML non interagisce con la SeekBar, mentre UTC, pur interagendo con quest'ultima, non ha effettuato tutte le possibili modifiche.

Nell'applicazione Trolly, una considerazione importante può essere fatta sulla particolare configurazione del ripper per la tecnica ML, nella quale la successione di eventi per inserire un oggetto in lista non si verifica. Per l'inserimento di un elemento nella lista dell'applicazione è necessario immettere il nome nell'apposita EditText e cliccare sul bottone predisposto per l'aggiunta dell'elemento. Non essendoci queste condizioni, nessun oggetto viene inserito nella lista e questo si ripercuote su altri metodi dell'applicazione come ad esempio il metodo "bindView()" che contiene particolari azioni da svolgere su questi elementi: OFF_LIST, ON_LIST, IN_TROLLEY.

Queste condizioni, nel caso UTC sono presenti. Il bottone "Add" (R.id.btn_add) viene cliccato e i restanti metodi vengono esplorati aumentando la copertura relativa alla classe di appartenenza. Nel codice che segue si può osservare una parte della copertura delle due tecniche a confronto.

Codice relativo alla tecnica ML - Trolly

```
246 mTextBox = (AutoCompleteTextView)findViewById(R.id.textbox);
247 btnAdd = (Button)findViewById(R.id.btn_add);
248
249 mTextBox.setOnClickListener(new Button.OnClickListener(){
250         @Override
251          public void onClick(View view) {
252 //If the text box is clicked while full-list adding, stop adding
253                 if (adding) {
254                   adding = false;
255                   updateList();
256                 }
257         }
258 });
```

Codice relativo alla tecnica UTC - Trolly

```
246 mTextBox = (AutoCompleteTextView)findViewById(R.id.textbox);
247 btnAdd = (Button)findViewById(R.id.btn_add);
248
249 mTextBox.setOnClickListener(new Button.OnClickListener(){
250         @Override
251          public void onClick(View view) {
252 //If the text box is clicked while full-list adding, stop adding
253                 if (adding) {
254                   adding = false;
255                   updateList();
256                 }
257         }
258 });
```

Nel caso dell'applicazione Munchife, la differenza di copertura risulta minima, avendo come risultante 80% (147,6/184) per ML e 86% (157,5/184) per UTC. Questo può essere giustificato da una particolare condizione che deve verificarsi durante il test. Questa condizione si verifica allorquando un giocatore raggiunge un livello prefissato e vince la partita. Per vincere una partita è necessario che si esegua una sequenza di pressioni sul pulsante "Up to Level" nella schermata principale dell'applicazione. Nelle impostazioni di MunchLife questo valore è impostato di default a 10, quindi alla decima pressione consecutiva, il giocatore vince una partita e una finestra di dialog viene visualizzata sullo schermo per informare dell'evento. Nella tecnica ML questa condizione non si verifica e il dialog non viene mai visualizzato. Per quanto riguarda UTC, invece, questa particolare

condizione viene eseguita dagli utenti con il conseguente aumento di copertura del codice relativo a questi eventi.

Codice relativo alla tecnica ML - MunchLife

```
255 case DIALOG_GAMEWIN:
256 AlertDialog.Builder gamewinbuilder = new AlertDialog.Builder(this);
257             gamewinbuilder.setMessage(R.string.win);
258             DialogInterface.OnClickListener gamewinClickListener;
259         gamewinClickListener = new DialogInterface.OnClickListener()
260                 {
261                  public void onClick(DialogInterface dialog, int item)
262                         {
263                            dialog.dismiss();
264                         }
265                     };
266    gamewinbuilder.setNeutralButton(R.string.ok, gamewinClickListener);
267              return gamewinbuilder.create();
```

Codice relativo alla tecnica UTC - MunchLife

```
255 case DIALOG_GAMEWIN:
256 AlertDialog.Builder gamewinbuilder = new AlertDialog.Builder(this);
257             gamewinbuilder.setMessage(R.string.win);
258             DialogInterface.OnClickListener gamewinClickListener;
259         gamewinClickListener = new DialogInterface.OnClickListener()
260                 {
261                  public void onClick(DialogInterface dialog, int item)
262                       {
263                          dialog.dismiss();
264                       }
265                 };
266    gamewinbuilder.setNeutralButton(R.string.ok, gamewinClickListener);
267              return gamewinbuilder.create();
```

Nell'applicazione FillUp, la differenza della copertura raggiunta dalle due tecniche è notevole: 13% (492,2/3807) per ML e il 75% (2867,6/3807) per UTC.

Come per i casi descritti in precedenza con SimplyDo e Trolly, nell'esplorazione del ripper in ML manca la pressione di alcuni pulsanti della GUI che consentono il passaggio ad altre activity. Mentre la "StartupActivity" viene esplorata, nella activity "MainActivity.java" la tecnica ML non interagisce con i tasti AddFuel, Log, Plot, Statistics.

La motivazione può essere ricercata nel dialog di inserimento veicolo.

Quando il ripper ML incontra il dialog di inserimento veicolo (VehicleDialog.java), resta "bloccato" in questo punto. Da questo punto in poi non risultano coperte le restanti classi.

Codice relativo alla tecnica ML - FillUp

```
202  public void onClick(View v) {
204       switch (v.getId()) {
206          case R.id.buttonVehicleAdd:
207               showDialog(DIALOG_ADD_VEHICLE_ID);  break;
210          case R.id.buttonVehicleEdit:
211              if (getSelectedVehicle() != null) {
212               showDialog(DIALOG_EDIT_VEHICLE_ID);  break; }
216           case R.id.buttonVehicleDelete:
217              if (getSelectedVehicle() != null) {
218               showDialog(DIALOG_DELETE_VEHICLE_ID); break; }
222           case R.id.buttonGetGas:
223               getGas(v);
224                 break;
226           case R.id.buttonViewLog:
227               viewLog(v);
228                  break;
230           case R.id.buttonPlotData:
231                  plotData(v);
232                  break;
234           case R.id.buttonViewStatistics:
235                   viewStatistics(v);
236                   break;
238           default:
239                  Utilities.toast(this,"Invalid view id.");
240             }
242    }
```

Codice relativo alla tecnica UTC - FillUp

```
202   public void onClick(View v) {
204         switch (v.getId()) {
206           case R.id.buttonVehicleAdd:
207                  showDialog(DIALOG_ADD_VEHICLE_ID); break;
210           case R.id.buttonVehicleEdit:
211              if (getSelectedVehicle() != null) {
212                     showDialog(DIALOG_EDIT_VEHICLE_ID); break; }
216           case R.id.buttonVehicleDelete:
217              if (getSelectedVehicle() != null) {
218                     showDialog(DIALOG_DELETE_VEHICLE_ID); break; }
222           case R.id.buttonGetGas:
223                getGas(v);
224                break;
226           case R.id.buttonViewLog:
227                viewLog(v);
228                break;
230           case R.id.buttonPlotData:
231                plotData(v);
232                break;
234           case R.id.buttonViewStatistics:
235                viewStatistics(v);
236                break;
238           default:
239                Utilities.toast(this,"Invalid view id.");
240                 }
242         }
```

5.6.2 Confronto UARI e ML

In questo confronto verranno mostrate le differenze qualitative ottenute tra la componente sistematica e l'unione delle tecniche Android Ripper Ibrido. Questo confronto, come il prossimo che verrà esaminato, sarà utile per notare le differenze sostanziali nell'uso di tecniche combinate quali UARI, con le sole sistematiche come ML.

I dati di copertura relativi all'applicazione con tecnica ML non vengono riportati perché già presenti in tabelle viste in precedenza. Vengono qui riportati i soli dati di copertura ottenuti con l'unione delle tecniche ARI con i diversi contributi dei test manuali.

Copertura tecnica UARI - SimplyDo

OVERALL COVERAGE SUMMARY - SimplyDo

name	class, %	method, %	block, %	line, %
all classes	96% (44/46)	87% (214/246)	86% (4723/5523)	**84% (1078,1/1281)**

Copertura tecnica UARI - TippyTipper

OVERALL COVERAGE SUMMARY - TippyTipper

name	class, %	method, %	block, %	line, %
all classes	98% (41/42)	88% (199/225)	90% (3830/4253)	**89% (887,2/999)**

Copertura tecnica UARI – Trolly

OVERALL COVERAGE SUMMARY – Trolly

name	class, %	method, %	block, %	line, %
all classes	89% (17/19)	84% (54/64)	80% (1507/1888)	**80% (292,4/364)**

Copertura tecnica UARI – MunchLife

OVERALL COVERAGE SUMMARY – MunchLife

name	class, %	method, %	block, %	line, %
all classes	100% (10/10)	100% (28/28)	91% (767/841)	**92% (168,8/184)**

Copertura tecnica UARI – FillUp

OVERALL COVERAGE SUMMARY – FillUp

name	class, %	method, %	block, %	line, %
all classes	90% (95/105)	83% (557/669)	81% (14601/18096)	**81% (3093,9/3807)**

Come è lecito aspettarsi, le tecniche UARI hanno percentuali molto maggiori di quelle ML. Questo perché la tecnica UARI oltre ad includere la tecnica ML, comprende la componente manuale dei test case effettuati dai diversi utenti e inoltre una componente sistematica innescata dopo la scoperta di nuovi stati.

Paragonando alcuni dati di copertura si passa ad esempio dal 32% (414,5/1281) di ML per SimplyDo a 84% (1078,1/1281) per UARI. L'adozione della tecnica ibrida permette oltre che ad avere una copertura più elevata, derivante dai test case utente (per motivazioni viste in ML-UTC), anche ad una più completa esplorazione di tipologie di codice non raggiunte nei test manuali grazie alla componente sistematica scatenata dopo la scoperta di nuovi stati.

5.6.3 Confronto UARI e UTC

I dati di copertura delle due tecniche non vengono riportati perché presenti nelle precedenti tabelle. Anche in questo confronto si mostreranno le differenze qualitative ottenute tra la l'unione delle tecniche Android Ripper Ibrido e le loro componenti manuali. Innanzitutto si notano alcune differenze sui dati di copertura come ad esempio in MunchLife con 86% (157,5/184) data da UTC contro il 92% (168,8/184) di UARI oppure riferendoci a FillUp si passa da 75% (2867,6/3807) di UTC a 81% (3093,9/3807) di UARI.

Addentrandosi nell'analisi, si riscontrano delle importanti differenze dovute ad alcune condizioni a cui avevamo accennato in precedenza. Per esempio, nell'applicazione SimplyDo alcune tipologie di codice come i rami IF-ELSE, nella tecnica UTC risultavano coperti in parte perché la componente manuale aveva esplorato un solo ramo della condizione. Nel caso di UARI oltre al ramo esplorato dai test case utente, viene esplorato, e quindi coperto, anche il ramo di codice mancante grazie all'apporto della componente sistematica, che con la sua esplorazione in ampiezza favorisce questa condizione.

Un altro particolare esempio può essere osservato in Tippy Tipper, nell'activity Tippy Tipper del package "net.mandaria.tippytipper.activities", in cui la componente umana (manuale) in UTC non copre un particolare evento di LongClick su dei pulsanti.

La componente automatica presente in UARI, invece, riesce ad effettuare tutte le possibili

interazioni su tutti i widget trovati: onLongClick, click, ecc.

Codice relativo alla tecnica UTC - TippyTipper

```
132 btn_delete.setOnLongClickListener(new OnLongClickListener()
133                 {
134                     public boolean onLongClick(View v)
135                       {
136                           clearBillAmount();
137                           return true;
138                       }
139                 });
141 // Added Clear Button -- Maintained the Long-Click Delete -- SPDJR
142         View btn_clear = findViewById(R.id.btn_clear);
143         btn_clear.setOnClickListener(new OnClickListener()
144                 {
145                  public void onClick(View v)
146                    {
147                       clearBillAmount();
148                       FlurryAgent.onEvent("Clear Button");
149                    }
150             });
152         btn_clear.setOnLongClickListener(new OnLongClickListener()
153                 {
154                  public boolean onLongClick(View v)
155                     {
156                       clearBillAmount();
157                       return true;
158                     }
159              });
```

Codice relativo alla tecnica UARI - TippyTipper

```
132 btn_delete.setOnLongClickListener(new OnLongClickListener()
133             {
134                 public boolean onLongClick(View v)
135                     {
136                       clearBillAmount();
137                       return true;
138                     }
139             });
141 // Added Clear Button -- Maintained the Long-Click Delete -- SPDJR
142         View btn_clear = findViewById(R.id.btn_clear);
143         btn_clear.setOnClickListener(new OnClickListener()
144                 {
145                  public void onClick(View v)
146                    {
147                       clearBillAmount();
148                       FlurryAgent.onEvent("Clear Button");
149                     }
150             });
152         btn_clear.setOnLongClickListener(new OnLongClickListener()
153                 {
154                  public boolean onLongClick(View v)
155                    {
156                       clearBillAmount();
157                       return true;
158                    }
159             });
```

Per quanto riguarda l'applicazione MunchLife non risultano particolari tipologie di codice che differiscono nelle due tecniche, tranne per il caso del lancio di un dado in cui vi è una componente randomica che influisce sui diversi risultati del lancio.

Codice relativo alla tecnica UTC – FillUp: DataEntryModeDialog.java

```
94  public void onCancel(DialogInterface dialog) {
95     listener.onDataEntryModeDialogResponse(id,Result.RESULT_CANCEL);
96                              }
```

Codice relativo alla tecnica UARI – FillUp: DataEntryModeDialog.java

```
94  public void onCancel(DialogInterface dialog) {
95     listener.onDataEntryModeDialogResponse(id,Result.RESULT_CANCEL);
96                              }
```

Codice relativo alla tecnica UTC – FillUp: DateTimeActivity.java

```
80  * Initialize the Activity's standard options menu.
84      public boolean onCreateOptionsMenu(Menu menu) {
85       //getMenuInflater().inflate(R.menu.activity_gas_record, menu);
86          return true;
87      }
109 * Get the current data values from the widgets after user edit,validate them,and return results
114     protected boolean getData() {
115             int month = datePicker.getMonth();
116             int day = datePicker.getDayOfMonth();
117             int year = datePicker.getYear() - 1900;
118             int hour = timePicker.getCurrentHour();
119             int minute = timePicker.getCurrentMinute();
120             Date date = new Date(year,month,day,hour,minute);
121             milliseconds = date.getTime();
122             return true;
123     }
```

Codice relativo alla tecnica UARI – FillUp: DateTimeActivity.java

```
80      * Initialize the Activity's standard options menu.
84      public boolean onCreateOptionsMenu(Menu menu) {
85          //getMenuInflater().inflate(R.menu.activity_gas_record, menu);
86          return true;
87      }
109 *Get the current data values from the widgets after user edit,validate them, and return results
114     protected boolean getData() {
115             int month = datePicker.getMonth();
116             int day = datePicker.getDayOfMonth();
117             int year = datePicker.getYear() - 1900;
118             int hour = timePicker.getCurrentHour();
119             int minute = timePicker.getCurrentMinute();
120             Date date = new Date(year,month,day,hour,minute);
121             milliseconds = date.getTime();
122             return true;
123     }
```

Nelle tabelle appena mostrate si osserva il comportamento sull'applicazione FillUp delle due tecniche. Alcune considerazioni posso essere fatte su degli eventi omessi dai test case manuali che invece risultano esplorate dalla componente sistematica, come ad esempio nella classe "DataEntryModeDialog.java" la pressione di alcuni pulsanti.

Un altro esempio simile si ha nella classe "DateTimeActivity.java" nella quale la tecnica UTC non ha cliccato su elemento menù e non ha coperto il metodo "getData()", mentre lo ha fatto la parte sistematica di UARI.

Continuando le nostre considerazioni per FillUp, si può rilevare che le differenze più sostanziali si notano sulla copertura di particolari classi dell'applicazione, in cui i test case effettuati dagli utenti tralasciano, in alcuni casi anche del tutto, alcuni aventi che invece nella componente sistematica vengono realizzati.

Ad esempio considerando le classi "FileSelectionActivity.java" in cui si riscontra 0% di copertura per UTC e ben il 79% per UARI, oppure FileSelectionListAdapter.java nella quale si ottiene 0% per UTC e 100% di copertura per UARI, si deduce che le coperture dei test case non hanno interessato queste righe di codice, mentre la componente sistematica nella tecnica UARI ha esplorato gran parte delle classi.

Continuando nell'analisi delle classi "GasLogListActivity.java" coperta per il 42% in UTC e per il 68% in UARI, si nota che la sostanziale differenza è dovuta a vari "case" e ad alcuni metodi riferiti alla schermata dell'activity Log, in cui la componente sistematica della tecnica UARI produce molti più eventi che non sono stati eseguiti dagli utenti su particolari azioni relative a file di log (import, export di file).

In UARI, come in UTC, non vengono effettuate operazioni con file CSV.

Nella classe "StatisticsActivity.java" per le stesse ragioni nella tecnica UTC è stata mancata la pressione del bottone di share dal menu in Log con alcuni eventi di "report file not create" e di "not shareReport"; o ancora in "TankNeverFilledDialog.java" il mancato click su "button cancel".

Alla luce di queste differenze è possibile affermare che l'adozione della tecnica ibrida permette di avere una copertura, anche se di poco, più elevata rispetto ai soli test case utente.

Questo risultato è frutto della componente sistematica che porta ad avere una più completa esplorazione di diverse tipologie di codice non raggiunte nei test manuali. Come discusso in questi ultimi paragrafi, queste possono essere classificate come condizioni di scelta tra rami IF-ELSE, o condizioni che riguardano la scelta di codice contenente switch-case, 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.

5.6.4 Confronto UARI e RDM

I dati di copertura della tecnica UARI non vengono riportati perché presenti nelle precedenti tabelle, ma si riportano quelle relative alla tecnica RDM.

Copertura tecnica RDM - SimplyDo

OVERALL COVERAGE SUMMARY - SimplyDo

name	class, %	method, %	block, %	line, %
all classes	96% (44/46)	87% (214/246)	86% (4755/5523)	**85% (1090,1/1281)**

Copertura tecnica RDM - TippyTipper

OVERALL COVERAGE SUMMARY - TippyTipper

name	class, %	method, %	block, %	line, %
all classes	98% (41/42)	86% (193/225)	88% (3756/4253)	**87% (873,7/999)**

Copertura tecnica RDM – Trolly

OVERALL COVERAGE SUMMARY – Trolly

name	class, %	method, %	block, %	line, %
all classes	89% (17/19)	86% (55/64)	80% (1518/1888)	**81% (294,4/364)**

Copertura tecnica RDM – MunchLife

OVERALL COVERAGE SUMMARY – MunchLife

name	class, %	method, %	block, %	line, %
all classes	90% (9/10)	93% (26/28)	87% (733/841)	**87% (159,9/184)**

Copertura tecnica RDM – FillUp

OVERALL COVERAGE SUMMARY – FillUp

name	class, %	method, %	block, %	line, %
all classes	81% (85/105)	78% (525/669)	75% (13569/18096)	**78% (2952,8/3807)**

La comparazione qualitativa effettuata su queste due tipologie di tecniche non ha riscontrato particolari differenze in termini di percentuali di copertura. I risultati di coverage non si discostano di molto. In Trolly ad esempio otteniamo 80% (292,4/364) per UARI e 81% (294,4/364) per RDM. In FillUp 81% (3093,9/3807) per UARI e 78% (2952,8/3807) per RDM.

Un risultato interessante scaturisce dalle coperture relative all'applicazione MunchLife, in cui osserviamo le percentuali di coverage di UARI: 92% (168,8/184) e di RDM: 87% (159,9/184).

Questa differenza, come nel caso di comparazione tra UARI e ML, è motivata dalla condizione di vittoria nel gioco MunchLife a seguito delle ripetute pressioni del pulsante "Up to Level". Anche in questo caso, nella tecnica RDM, non si verifica questa particolare condizione e il dialog associato all'evento non viene visualizzato.

Questa serie di eventi nelle tecniche UARI, che includono i test generati dagli utenti, essendo stata generata, ha fatto in modo di aumentare del 5% la copertura relativa all'applicazione. Nel codice riportato di seguito si può osservare la differenza di una parte del codice delle due tecniche.

Codice relativo alla tecnica RDM – MunchLife : MunchLifeActivity.java

```
255 case DIALOG_GAMEWIN:
256 AlertDialog.Builder gamewinbuilder = new AlertDialog.Builder(this);
257             gamewinbuilder.setMessage(R.string.win);
258             DialogInterface.OnClickListener gamewinClickListener;
259           gamewinClickListener = new DialogInterface.OnClickListener()
260                 {
261                  public void onClick(DialogInterface dialog, int item)
262                          {
263                             dialog.dismiss();
264                          }
265                      };
266    gamewinbuilder.setNeutralButton(R.string.ok, gamewinClickListener);
267               return gamewinbuilder.create();
```

Codice relativo alla tecnica UARI – MunchLife: MunchLifeActivity.java

```
255 case DIALOG_GAMEWIN:
256 AlertDialog.Builder gamewinbuilder = new AlertDialog.Builder(this);
257             gamewinbuilder.setMessage(R.string.win);
258             DialogInterface.OnClickListener gamewinClickListener;
259           gamewinClickListener = new DialogInterface.OnClickListener()
260                 {
261                  public void onClick(DialogInterface dialog, int item)
262                          {
263                             dialog.dismiss();
264                          }
265                      };
266    gamewinbuilder.setNeutralButton(R.string.ok, gamewinClickListener);
267               return gamewinbuilder.create();
```

Nelle altre applicazioni si riscontrano alcune differenze di copertura, che derivano da mancate condizioni che la tecnica RDM non è stata in grado di verificare. In particolare si rileva che in FillUp che nella classe "GasGauge.java" si ottiene per UARI 65% di copertura e per RDM il 25%. Queste condizioni fanno presupporre che nei test RDM non venga visualizzato il "gas gauge" (indicatore benzina) associato agli eventi di calcolo del carburante.

Altre piccole differenze si notano su condizioni di "fullTank": alcune parti del codice nelle altre classi non vengono coperte da RDM (vedi GasRecordList.java, MileageCalculation.java, MileageEstimateDialog.java, StatisticsSummaryTable.java).

Altro caso simile in "PlotDateRange.java" in cui alcune condizioni di "case", riguardanti i pulsanti in Plot, non sono coperte da RDM.

5.6 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 di casi di test. Giunti a questo punto, dopo aver esaminato i risultati ottenuti dai diversi confronti, le risposte alle research questions introdotte ad inizio capitolo possono essere argomentate come segue.

RQ 1) L'utilizzo della tecnica Android Ripper Ibrido (ARI) permette di migliorare in efficacia rispetto alla tecnica Solo Test Case?

Dalle valutazioni sulle distribuzioni analizzate e dall'analisi qualitativa si può affermare che la tecnica ARI ha permesso di migliorare l'efficacia dei test eseguiti sulle applicazioni rispetto ai soli test case. La risposta a questa domanda è giustificata dal fatto che la tecnica ARI, oltre ad includere la componente dei test eseguiti manualmente, riesce ad aumentare i risultati di coverage 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, o condizioni che riguardano la scelta di codice contenente switch-case, 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) L'utilizzo della tecnica Android Ripper Ibrido (ARI) permette di migliorare in efficacia rispetto alla tecnica Android Ripper nella versione sistematica (ML)?

Come per la precedente domanda di ricerca, dalle valutazioni effettuate si evince che la tecnica ARI 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 ARI 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) L'utilizzo della tecnica Solo Test Case è migliore in termini di efficacia rispetto alla tecnica Model Learning o hanno la stessa copertura?

Come osservabile dai dati, la risposta a questa domanda sembra scontata se si guarda alle sole percentuali di copertura; infatti, la tecnica soloTC 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 soloTC. 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'utilizzo della tecnica Android Ripper Ibrido (ARI) permette di migliorare in efficacia rispetto alla tecnica Android Ripper nella sua versione random (RDM)?

Riguardo alla comparazione sull'efficacia di queste due tecniche non è possibile trarre delle conclusioni precise e scegliere quale delle due sia migliore dell'altra. Questo perché oltre ad equipararsi grossomodo sui loro risultati di coverage, non presentano significative differenze sulle tipologie di codice esplorato.

Una considerazione però molto interessante ci viene fornita dall'esercizio di queste tecniche su determinate applicazioni (nel nostro caso MunchLife), in cui una particolare serie di eventi scatenati sulla GUI porta al verificarsi di determinate situazioni e quindi di sezioni di codice difficilmente raggiungibili, almeno per quanto osservato, nelle tecniche RDM.

Come abbiamo evidenziato nel paragrafo 5.6.4 Confronto UARI e RDM, la sequenza di eventi (nel nostro caso di "click" ripetuti sul pulsante "Up to Level") genera una casistica che viene riscontrata solo nella tecnica ARI e non in quella RDM.

Conclusioni

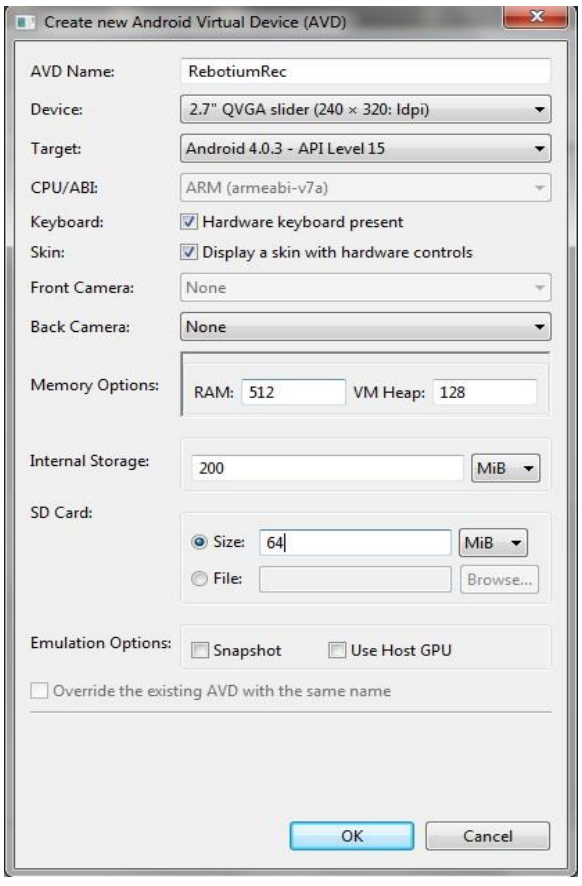
Come si è avuto modo di verificare con il confronto tra l'efficacia delle tecniche di generazione di casi di test, l'adozione della tecnica ibrida ha portato un significativo vantaggio nell'esplorazione delle applicazioni. La componente automatica unita alla componente manuale ha consentito la copertura di porzioni di codice difficilmente raggiungibili da una sola delle due. Il progetto Android Ripper Ibrido, oltre a rilevarsi 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, e i test generati in maniera sistematica con l'esplorazione in ampiezza delle applicazioni.

Inoltre le analisi a cui sono state sottoposte le varie tecniche, hanno dato modo di esprimere e categorizzare tipologie e differenze di codice esplorato che potranno servire in futuro per caratterizzare altre versioni del ripper e cercare di apportare miglioramenti.

Appendice A – Utilizzo Robotium Recorder

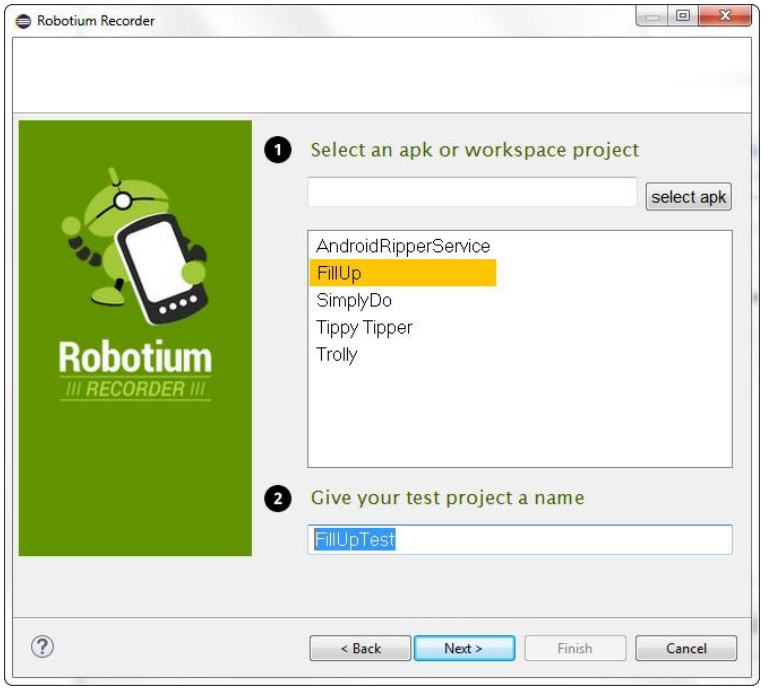
In questa sezione vedremo come eseguire una registrazione di un test con Robotium Recorder ver. 2.1.25 su Eclipse.

Creare un AVD (requisito: API Level 15 o superiore) e avviarlo.

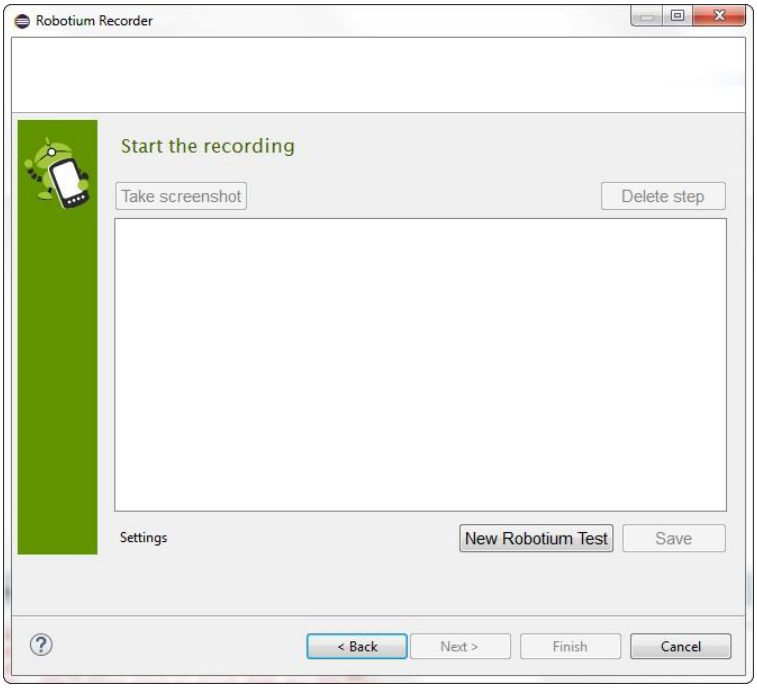


Da Eclipse: File > New > Other. Nella finestra selezionare "New Robotium Test" dalla sezione "Android - Robotium Recorder" e click su "Next".

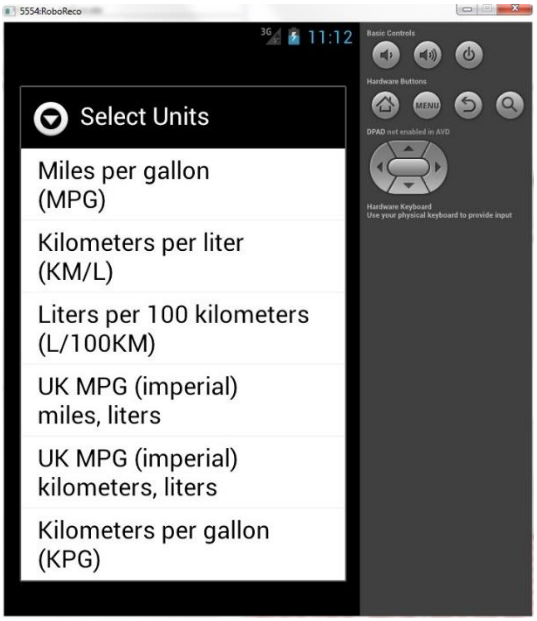
Selezionare l'applicazione su cui effettuare la registrazione (es. FillUp, il cui progetto è presente in Eclipse) e dare un nome al progetto di test (es FillUpTest).



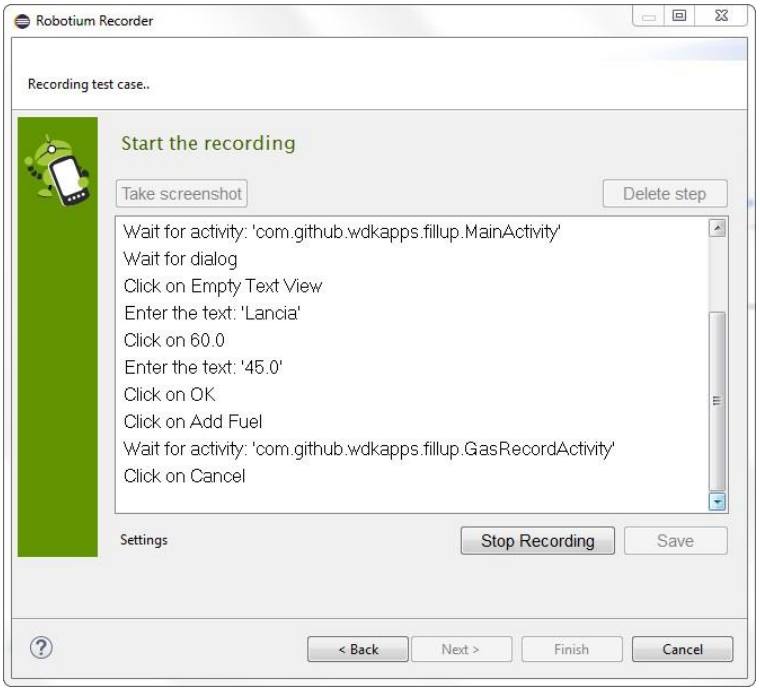
click su "Next":



click su "New Robotium Test". Nell'AVD verrà eseguita l'applicazione (FillUp); esplorare l'applicazione:



Nella finetra di Robotium Recorder verranno registrati tutti gli eventi scatenati nella GUI dell'app:



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

E' 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.

E' fortemente raccomandato, al termine della registrazione, rieseguire il test Android JUnit prodotto per verificare il corretto comportamento.

E' 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	E' 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 E' possibile scaricare l'installer per Windows, Linux e Mac OS X o il pacchetto compresso pronto all'uso per Windows, Linux e Max OS X.
Android Libraries	Versione 2.3.3 o più recenti
Ant	Versione 1.8.2 o più recente. Ant può essere scaricato qui
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. Ant può essere scaricato qui

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	E' 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] Android wiki: http://it.wikipedia.org/wiki/Android

[2] Android Developer: http://developer.android.com/

[3] Lezioni Android Pelliccia: http://www.informatica.uniroma2.it/

[4] Android Developer - Testing: http://developer.android.com/tools/testing/testing_android.html

[5] Robotium Recorder - http://robotium.com/products/robotium-recorder

[6] Luca Nastro. Sperimentazione di tecniche parallele di generazione automatica di casi di test per applicazioni Android

[7] Giuseppe Di Maio. Realizzazione di tecniche parallele di generazione automatica di casi di test per applicazioni Android

[8] http://wiki.gurubee.net/pages/viewpage.action?pageId=26743123

[9] Android Kernel Features : http://elinux.org/Android_Kernel_Features

[10] Android-Linux-Kernel-Additions http://www.lindusembedded.com/blog/2010/12/07/android-linux-kernel-additions/

[11] Robotium : https://code.google.com/p/robotium/

[12] Emma : http://emma.sourceforge.net/index.html

[13] Lamberto Soliani. Fondamenti di Statistica Applicata: http://www.dsa.unipr.it/soliani/soliani.html

[14] Carli M. . Android guida per lo sviluppatore. Apogeo, 2010.

[15] Charles Zaiontz - http://www.real-statistics.com/

[16] The R Project for Statistical Computing - http://www.r-project.org/