POLITECNICO DI TORINO

Laurea Magistrale in Software Engineering

Tesi di Laurea Magistrale

Sviluppo di uno strumento per l'analisi della fragilità delle suite di test

Supervisori

Dott. Luca ARDITO

Dott. Riccardo COPPOLA

Candidato

Francesco RIBA

Aprile 2022

Sommario

Il system testing tramite l'interfaccia utente grafica (GUI) è una preziosa forma di verifica e convalida per le applicazioni moderne, in particolare nei settori ad alta intensità grafica come le applicazioni mobili. Tuttavia, la pratica è spesso trascurata dagli sviluppatori principalmente a causa della sua natura costosa e dell'assenza di un feedback immediato sulla qualità della suite di test. Questa tesi propone lo sviluppo di un software che può essere utilizzato per ottenere feedback sulla qualità dei test attraverso la rappresentazione grafica di alcune metriche, consentendo così una rapida comprensione della qualità di questi ultimi. L'estrema digitalizzazione che si sta verificando in questo periodo storico porta ad un aumento significativo dello sviluppo di applicazioni mobili e non. Il lavoro di questa tesi è quello di fornire una misura per il test, che consenta di migliorarne le funzionalità e di conseguenza la qualità del codice delle applicazioni.

Il lavoro di tesi è iniziato con lo studio e la selezione di alcune metriche relative sia al codice che compone l'applicazione in questione, sia alle suite di test. Tra tutte le metriche descritte in letteratura, è stata selezionata una serie generalizzabile e di facile comprensione, che è risultata più adeguata per una presentazione in uno strumento grafico. Sulla base di queste decisioni, sono state sviluppate le seguenti applicazioni: la prima basata su script python e la seconda in Java. L'applicazione Python riceve come input un repository di codice open source, esegue i pull di tutte le versioni e calcola le metriche. Una volta calcolate le metriche, le inserisce in un file CSV. Tale file verrà successivamente esaminato dall'applicazione Java che manipola la GUI con cui l'utente interagisce, consentendo la generazione di più grafici e il calcolo di varie funzioni statistiche. Alcuni dati relativi all'intero progetto sono ottenuti con l'ausilio del software SonarQube.

Uno dei principali problemi legati a questa tesi è stato quello di ottenere repository con codice di qualità: la maggior parte dei repository analizzati, infatti, presentava una piccola percentuale di linee di codice di test rispetto a quella delle linee di codice che compone l'applicazione. Questa notevole differenza implica un importante problema di fondo: spesso gli sviluppatori concentrano molte più energie sullo sviluppo del codice tralasciando così quella dei test case. In conclusione, è stato sviluppato uno strumento di lavoro completo ed è stato valutato su molti

progetti open source testati. Lo strumento è stato in grado di recuperare tutte le metriche desiderate e i grafici generati sono facilmente comprensibili. Lo strumento attuale può essere applicato a qualsiasi progetto open source per fornire preziose informazioni agli sviluppatori nell'analisi dello stato delle loro suite di test.

Un futuro upgrade sullo strumento potrebbe includere l'integrazione continua direttamente nella piattaforma GitHub Actions e l'implementazione della capacità di riconoscere linguaggi diversi da java e kotlin.

Indice

Elenco delle Immagini

Capitolo 1

Introduzione

1.1 V-Model

Nello sviluppo del software, il V-Model[1] rappresenta un processo di sviluppo che può essere considerato un'estensione del modello a cascata[2]. Invece di scendere in modo lineare, le fasi del processo vengono piegate verso l'alto dopo la fase di sviluppo, per formare la tipica forma a V. Il V-Model mostra le relazioni tra ciascuna fase del ciclo di vita dello sviluppo e la fase di test associata, in immagine 1.1 é illustrato un grafico esplicativo. Gli assi orizzontale e verticale rappresentano rispettivamente il tempo o la completezza del progetto (da sinistra a destra) e il livello di astrazione.

[1] `https://en.wikipedia.org/wiki/V-Model_(software_development)`

[2] Il modello a cascata è la scomposizione delle attività del progetto in fasi sequenziali lineari, in cui ogni fase dipende dai risultati della precedente e corrisponde ad una specializzazione dei compiti. L'approccio è tipico di alcune aree della progettazione ingegneristica. Nello sviluppo del software, tende a essere tra gli approcci meno iterativi e flessibili, poiché il progresso scorre in gran parte in una direzione ("verso il basso" come una cascata) attraverso le fasi di ideazione, avvio, analisi, progettazione, costruzione, test, distribuzione e manutenzione

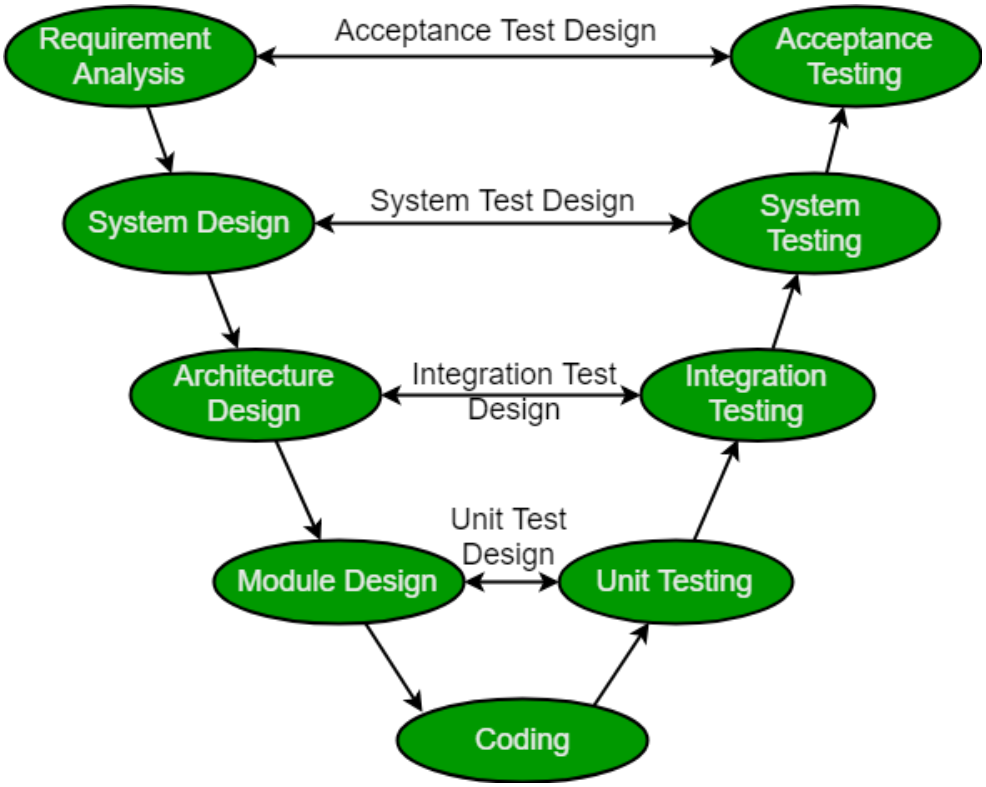


Immagine 1.1: V-Model rappresentato graficamente

1.2 Verifica e Validazione

I termini "verifica" e "validazione" sono comunemente usati nell'ingegneria del software e si riferiscono a due diversi tipi di analisi.

1.2.1 Che cos'è la verifica?

La verifica è un processo che determina la qualità del software. La verifica include tutte le attività associate alla produzione di software di alta qualità, ovvero: test, ispezione, analisi del progetto, analisi delle specifiche e così via. La verifica è un processo relativamente oggettivo, in quanto, se i vari processi e documenti sono espressi in modo sufficientemente preciso, non dovrebbe essere necessario alcun giudizio soggettivo per verificare il software. La verifica aiuta ad abbassare il numero dei difetti che possono essere riscontrati nelle fasi successive dello sviluppo. La verifica del prodotto nella fase iniziale dello sviluppo aiuta a comprendere il prodotto in modo più completo, riduce le possibilità di errori nell'applicazione software o nel prodotto e oltretutto guida nella costruzione del prodotto secondo le

specifiche e le esigenze del cliente.

1.2.2 Cos'è la validazione?

La validazione è un processo in cui i requisiti del cliente sono effettivamente soddisfatti dalla funzionalità del software. La validazione avviene sia al termine del processo di sviluppo sia termine delle verifiche. Se la verifica non identifica tutti i difetti, durante il processo di validazione questi possono essere rilevati come guasti. Se durante la verifica alcune specifiche vengono fraintese e lo sviluppo è già avvenuto, durante il processo di validazione è possibile identificare la differenza tra il risultato effettivo e il risultato atteso e intraprendere azioni correttive. La validazione viene eseguita durante i test come test di funzionalità, test di integrazione, test di sistema, test di carico, test di compatibilità oppure test di stress. La validazione aiuta a costruire il prodotto giusto in base alle esigenze del cliente che a sua volta soddisferà le esigenze dei processi aziendali.

1.2.3 Come si differenziano?

La distinzione tra i due termini è in gran parte dovuta al ruolo delle specifiche. La validazione è il processo per verificare se le specifiche soddisfano i requisiti del cliente, mentre la verifica è il processo per controllare che il software soddisfi le specifiche. La verifica include tutte le attività associate alla produzione di software di alta qualità. È un processo relativamente oggettivo in quanto non dovrebbe essere necessario alcun giudizio soggettivo per verificare il software. Al contrario, la validazione è un processo estremamente soggettivo: valuta quanto il sistema risponda adeguatamente alle esigenze del mondo reale. La validazione include attività come la modellazione dei requisiti, la prototipazione e la valutazione degli utenti.

1.3 Testing Pyramid

L'argomento principale di questo lavoro di tesi è lo studio del processo attraverso il quale i test delle applicazioni vengono sviluppati; per una spiegazione più dettagliata, si utilizza come riferimento il concetto di Testing Pyramid[3].

[3] `https://www.perfecto.io/blog/testing-pyramid`

Immagine 1.2: Rappresentazione grafica della Testing Pyramid

La piramide di test rappresenta, in modo efficace, come vengono suddivisi i test case durante lo sviluppo di progetti standard. Per rilasciare con successo un'applicazione agli utenti, è necessario un test approfondito basato su una solida strategia di automazione dei test. Generalmente, gli sviluppatori dedicano molto più interesse su determinate tipologie di test rispetto ad altre, le piattaforme che vengono utilizzate, variano in base alla tipologia di test che si vuole approfondire. Tutto questo viene semplicemente mappato nella piramide. La piramide dei test è un concetto che raggruppa i test del software in tre diverse categorie. Questo aiuta gli sviluppatori e i professionisti del controllo qualità a garantire una qualità superiore, ridurre il tempo necessario per trovare la causa principale dei bug e creare una suite di test più affidabile. Alla base della piramide sono presenti i test più automatizzati e veloci, mentre all'apice quelli più lenti e manuali. Progredendo dal basso verso l'alto, si riduce il numero di test. Dunque, i team dovrebbero disporre di test unitari, piccoli e isolati, e non di molti test esplorativi end-to-end, i quali devono essere programmati manualmente richiedendo più tempo. Il caricamento anticipato dei test a livello di unità rende la correzione dei bug più rapida e semplice, al contrario dei difetti che vengono scoperti molto più avanti nel processo.

1.3.1 Unit Testing

La base della piramide è lo unit testing. Le sezioni di codice testate dagli unit test sono ridotte e garantiscono che le unità di codice isolate funzionino adeguatamente. Gli unit test dovrebbero testare una variabile e non basarsi su dipendenze esterne. In questo livello della piramide, i test vengono eseguiti prima e dopo i commit. I test vengono attivati dagli sviluppatori.

1.3.2 Integration Testing

L'integration testing è la fase del test del software in cui i singoli moduli software vengono combinati e testati come gruppo. Il test di integrazione viene condotto per valutare la conformità di un sistema o di un componente ai requisiti funzionali specificati. Si verifica dopo lo unit testing e prima delle sezioni UI & Exploratory Tests. Il test di integrazione prende come input moduli che sono stati testati attraverso lo unit testing, li raggruppa in aggregati più grandi, applica i test definiti e fornisce come output il sistema integrato pronto per i test successivi.

1.3.3 UI & Exploratory Tests

Lo strato superiore della piramide riguarda l'interfaccia utente e i test esplorativi. Questi test in genere sono più complicati e hanno più dipendenze rispetto agli unit test e a quelli di integrazione. Mentre i team possono accelerare i test dell'interfaccia utente con l'automazione, i test esplorativi vengono in genere eseguiti manualmente e richiedono più tempo. Tenendo presente la forma della piramide, i test in questa sezione sono generalmente molto meno numerosi di quelli alla base; il fattore tempo ne è la causa principale.

1.4 Obiettivi

La fragilità è una delle problematiche principali quando si sviluppano test per applicazioni web e mobile. In generale, si dice che un caso di test sia fragile quando fallisce o ha bisogno di manutenzione a causa della naturale evoluzione dell'applicazione che si sta testando, mentre le funzionalità specifiche non sono state alterate. Nella tesi corrente si vuole creare uno strumento per l'identificazione, la misurazione e l'analisi di quanto la fragilità sia presente all'interno di progetti software, dei quali si dispone del codice e dei test.

1.5 Struttura della tesi

Nei capitoli successivi si esaminano i seguenti argomenti:

- *Capitolo 2*: Background della tesi, fondamenti teorici dietro le metriche e alla base dello studio della fragilità.
- *Capitolo 3*: Spiegazione dettagliata delle metriche e del loro rapporto con il codice.
- *Capitolo 4*: Avanzamento temporale dettagliato dello sviluppo del codice alla base del lavoro di tesi.

- *Capitolo 5*: Conclusioni ed eventuali possibili upgrade realizzabili in futuro.

Capitolo 2

Background

Il lavoro presentato in questa tesi è posizionato nelle aree dei processi di sviluppo del software, della verifica e validazione del software, del test automatizzato del software e dello sviluppo di applicazioni mobile.
In questa sezione vengono fornite informazioni sui concetti di base del test End-2-End, in particolare quando il software viene testato interagendo con esso tramite la relativa interfaccia utente. È inoltre fornita una panoramica delle diverse generazioni e tecniche per l'esecuzione di test della GUI[1].
Infine, viene introdotto il concetto di fragilità, insieme ad una discussione di analisi preliminare eseguita sulla verifica della fragilità in altri domini, portando alla definizione che è stata adottata per tutti gli studi che costituiscono la presente tesi.

2.1 Classificazione delle tecniche di test automatizzato della GUI

Diversi approcci sono stati identificati per l'esecuzione di test GUI di applicazioni di qualsiasi dominio. Una mappatura sistematica delle tecniche di test GUI disponibili è stata eseguita da Banerjee et al [1].
Gli strumenti di test automatizzati della GUI possono essere classificati in base a vari aspetti:

- l'uso di modelli (e il modello specifico adottato) per la descrizione delle interfacce utente;
- il modo in cui vengono generati gli input per la GUI;

[1]GUI: Graphical User Interface

- il linguaggio che viene utilizzato per scrivere script di test per la ri-esecuzione degli stessi test

Per essere definiti *automatizzati*, gli strumenti di test GUI devono automatizzare almeno alcune fasi del flusso di lavoro dei test, ad esempio l'esecuzione di script di test. Per quanto riguarda la generazione di test cases, diverse tecniche si basano ancora sulla creazione manuale di script di test da parte del tester/sviluppatore in una sintassi specifica.

Gli strumenti di test *Capture & Replay*, invece, si basano sulla registrazione delle operazioni eseguite sulla GUI da un tester (umano), che vengono tradotte in uno script di test da un motore e sono poi replicabili sull'AUT[2], per imitare l'uso umano.

Le tecniche di test *basate su modelli (MBT)* sono approcci black-box che si affidano a modelli di un sistema in fase di test e/o del suo ambiente per generare i test cases. I test cases vengono generati sulla base di un'astrazione del SUT[3], secondo uno specifico criterio di selezione del test (ad esempio, copertura del modello strutturale o copertura dei 10 requisiti di background del SUT). Generalmente, i modelli utilizzati da questi approcci sono macchine a stati finiti o diagrammi UML. Il modello può essere generato dal tester/sviluppatore, fornito come parte dei requisiti, o automaticamente decodificato dall'AUT.

Le tecniche di test *casuali* si basano su sequenze aleatorie di input che vengono inviati all'applicazione, al fine di innescare potenziali difetti e crash. I modelli della GUI possono essere utilizzati da strumenti di test casuali per distribuire gli input strutturati cosicché possano assomigliare alle tipiche interazioni umane con il SUT.

Le tecniche di test automatizzato della GUI possono adottare diversi oracoli[4] per capire se un test case è stato superato o no. I riferimenti di stato sono una delle forme più comuni: in questo caso, gli stati della GUI estratti durante una prima esecuzione dell'AUT, noti per essere corretti, vengono utilizzati per verificare ulteriori esecuzioni di test cases. Nei test dei crash non vengono utilizzati oracoli espliciti; un test case viene contrassegnato come fallito se l'AUT si arresta in modo anomalo durante la sua esecuzione. I metodi di verifica formale utilizzano modelli o specifiche dell'AUT, per verificare la correttezza dell'output di un test case.

Le tecniche di GUI testing sono un aiuto rilevante per verificare l'affidabilità di applicazioni che, con il loro aspetto grafico, si basano su un'interazione costante con l'utente finale. Molti sforzi, ad esempio, sono stati dedicati all'applicazione dei

[2]AUT: Application Under Testing

[3]SUT: Software Under Testing

[4]oracolo: è un meccanismo usato nel collaudo del software e nell'ingegneria del software per determinare se un test ha avuto successo o è fallito

concetti di test basati su modelli per applicazioni web, i quali vengono utilizzati per descrivere le transizioni tra le diverse schermate del SUT e il loro contenuto.

2.1.1 L'evoluzione degli strumenti per i test della GUI

Gli strumenti di test automatizzati della GUI, secondo una definizione formulata da Alegroth et al [2]. e adottata per il resto di questa tesi, possono anche essere classificati in diverse generazioni, in base al livello di astrazione che utilizzano per interagire con la GUI quando definiscono sequenze di comandi o li eseguono:

- Gli strumenti di test di *prima generazione* (o coordinate-based) utilizzano coordinate esatte sugli schermi dell'AUT per identificare i luoghi in cui devono essere eseguite le interazioni. Le coordinate vengono registrate durante l'interazione manuale con l'AUT. Gli strumenti di test di prima generazione non hanno alcuna conoscenza dei componenti delle schermate dell'applicazione.

- Gli strumenti di test di *seconda generazione* (o layout-based) si basano su un modello dell'interfaccia utente grafica che viene scomposta in layout e gerarchie di componenti. Proprietà e valori sono associati a ciascun elemento della GUI, permettendo così di identificarli. Ad esempio, gli ID nella definizione delle gerarchie di layout dello schermo possono essere sfruttati per identificare in modo univoco gli elementi dell'interfaccia utente su cui devono essere eseguite le interazioni.

- Gli strumenti di test di *terza generazione* (o visual-based) utilizzano il riconoscimento delle immagini per trovare gli elementi della GUI su cui eseguire le interazioni e per fornire asserzioni sulla correttezza del SUT dopo l'esecuzione di una determinata sequenza di azioni. Le catture esatte dello schermo dei componenti vengono quindi utilizzate in ogni fase dei test cases. Gli strumenti di test basati su coordinate, al giorno d'oggi, sono raramente adottati a causa della loro scarsa adattabilità anche a piccoli cambiamenti nelle GUI e quindi alla loro mancanza di robustezza.

Gli strumenti di test basati sul layout e gli strumenti di test visual-based presentano, a loro modo, sia vantaggi che svantaggi per i tester. Il visual testing è più appropriato per testare l'aspetto effettivo dell'applicazione, ricreando meglio l'utilizzo di un utente finale; d'altra parte, il test basato sul layout definisce ogni interazione in base alle proprietà dei componenti della GUI e non al loro aspetto. Di conseguenza è più appropriato verificare una corretta composizione delle schermate dell'applicazione ed un corretto funzionamento della gerarchia dello schermo.

Gli studi effettuati hanno dimostrato la vantaggiosa applicabilità degli strumenti di test Visual GUI in contesti industriali ed i vantaggi di approcci che combinano tecniche basate su Layout e Visual. Tuttavia, l'adozione di tali strumenti è

ostacolata da una minore robustezza e prestazioni inferiori rispetto agli strumenti layout-based.

2.2 La struttura delle applicazioni Android

Android è un sistema operativo open source abbinato a una piattaforma di sviluppo di applicazioni basata sul kernel Linux[5]. Il kernel Linux è sfruttato da elementi di livello superiore della piattaforma Android per funzionalità come threading, gestione della memoria e funzionalità di sicurezza. Sul kernel Linux si trova l'Hardware Abstraction Layer, un insieme di interfacce che viene utilizzato per rendere disponibili le capacità hardware del dispositivo ai livelli superiori della piattaforma. Per utilizzare funzioni specifiche offerte dal sistema operativo, un'applicazione dovrà richiedere la relativa autorizzazione su un file specifico che si trova nella directory principale di qualsiasi progetto Android, ovvero il file Manifest XML[6]. Viene avviata un'istanza di Android Runtime[7] (ART) per ogni applicazione, a cui è associato anche un processo a sé stante. Android Runtime è abbinato a un set di librerie native C/C++, necessarie per molti componenti e servizi del sistema operativo Android. API Framework è l'insieme di classi rese disponibili per gli sviluppatori Android per la creazione delle loro applicazioni. Il View System offre agli sviluppatori la possibilità di costruire le GUI delle loro applicazioni, attraverso le quali vengono raccolte tutte le interazioni degli utenti e viene esposta la maggior parte delle funzionalità. Posizionate in cima allo stack Android, le app di sistema costituiscono l'insieme delle app predefinite di base di cui è dotata ogni versione del framework Android.

2.2.1 Applicazioni Android

Una definizione generale di App Mobile è stata fornita da Muccini et al. come "un'applicazione in esecuzione su dispositivi mobili e/o che prende informazioni contestuali in input" [3]. Secondo questa definizione, un'applicazione mobile aggiunge al suo mobile (cioè può essere eseguita su un dispositivo elettronico mobile) anche una natura sensibile al contesto, nel senso che l'applicazione si adatta e reagisce costantemente all'ambiente di elaborazione in cui viene eseguita. A seconda del modo in cui sono programmate e del modo in cui sfruttano i componenti offerti dai framework specifici per cui sono sviluppate, le app mobili possono essere classificate in tre diverse categorie: app native, basate sul Web e ibride.

[5] https://developer.android.com/guide/platform/

[6] https://developer.android.com/guide/topics/manifest/manifest-intro

[7] https://source.android.com/devices/tech/dalvik

- Le *app native* sono scritte in un linguaggio di programmazione specifico, per una specifica piattaforma di dispositivi. Java (recentemente associato a Kotlin) viene utilizzato per scrivere app native Android.
- Le *app basate sul Web* sono applicazioni che vengono caricate nei browser Web e che forniscono all'utente funzionalità e interazioni specificamente destinate all'utilizzo da parte di un dispositivo mobile.
- Le *app ibride* combinano i principi delle app native con quelle delle applicazioni basate sul Web, sfruttando determinati componenti di una piattaforma specifica per caricare, in fase di esecuzione, contenuti da Internet. Su Android, il caricamento dinamico del contenuto delle app ibride viene eseguito con l'utilizzo di WebViews [8].

La piattaforma di sviluppo Android fornisce quattro componenti di base con cui è possibile creare app native. Ogni componente ha un ciclo di vita specifico, che è guidato dal sistema operativo con l'invocazione di un insieme di metodi (ad esempio, la funzione onCreate, che è la prima richiamata da una nuova istanza di un componente).

I componenti appartengono alle seguenti classi:

- Le *Activity* [9] sono incaricate di costruire l'interfaccia utente. In genere, ogni activity è dedicata ad una schermata o ad uno scenario d'uso dell'applicazione. Le activity gestiscono tutte le risposte attivate dagli input dell'utente.
- I *Services* [10] gestiscono operazioni in background a lunga durata effettuate dall'app, che non necessitano di alcuna interazione da parte dell'utente (ad esempio, gestione delle connessioni di rete).
- I *Content Providers* [11] gestiscono i dati memorizzati dall'applicazione e la condivisione delle informazioni con altre applicazioni del sistema.
- I *Broadcast Receivers* [12] rispondono agli eventi inviati dal sistema Android e gestiscono il modo in cui l'app deve rispondere ad essi.

Le Activity sono quindi i componenti principali di qualsiasi app Android. Ogni Activity definisce e costruisce un'interfaccia utente, composta da View disposte

[8] https://developer.android.com/reference/android/webkit/WebView

[9] https://developer.android.com/reference/android/app/Activity

[10] https://developer.android.com/guide/components/services

[11] https://developer.android.com/guide/topics/providers/content-provider-basics

[12] https://developer.android.com/reference/android/content/BroadcastReceiver

secondo un particolare layout. Un layout viene definito a livello di codice o staticamente all'interno di un file di layout XML, che viene quindi generato nella prima operazione eseguita durante la transizione ad un'attività. Oltre alla relativa disposizione delle View all'interno dello schermo del dispositivo, i layout allegano proprietà agli elementi dell'interfaccia utente, ad esempio ID univoci che possono quindi essere utilizzati dall'applicazione per recuperare gli elementi della GUI sui quali eseguire le operazioni. I callback possono anche essere collegati agli elementi dei layout, al fine di innescare comportamenti specifici in risposta alle interazioni eseguite dall'utente. Dall'API Android 12 sono stati introdotti i Fragments [13] al fine di gestire più facilmente interfacce che devono adattarsi, in modi complessi, a dimensioni dello schermo, orientamento, densità e formato del dispositivo diversi.

2.3 Testing delle applicazioni Android e Mobile

Il Mobile Testing, come proposto da Gao et al. [4], può essere definito come *"attività di test per applicazioni native e Web su dispositivi mobili, utilizzando metodi e strumenti di test software ben definiti, per garantire la qualità delle funzioni, dei comportamenti, delle prestazioni e della qualità del servizio"*.

Esistono diverse peculiarità delle applicazioni mobile quando si tratta di testarle. Ad esempio, Anureet et al. [5] trattano i test di compatibilità, i test delle prestazioni e i test di sicurezza come esigenze primarie per le applicazioni mobile. Diverse fonti identificano il test della GUI come test fondamentale per tutte le applicazioni mobile, poiché i malfunzionamenti della GUI per un'app mobile possono ostacolare l'esperienza fornita all'utente. Molti degli approcci descritti per il test automatizzato della GUI sono stati adattati al dominio delle applicazioni mobile, in particolare alle app Android. Molti studi affrontano la sfida di automatizzare l'intera procedura di test per le app Android o per parti di essa (ad esempio, la generazione di modelli o l'esecuzione di test cases su più dispositivi diversi).

2.3.1 Categorie di tool e servizi per test di applicazioni mobile

Una panoramica degli strumenti di test disponibili per il testing automatico delle applicazioni mobile, non solo limitata ai test della GUI, è stata fornita da Linares-Vasquez et al. [6][7], che hanno suddiviso gli strumenti nelle seguenti categorie: Automation API/Framework, Record and Replay Tools, Automated Test Input Generation Techniques, Bug and Error Reporting/Monitoring Tools, Mobile Testing

[13] https://developer.android.com/guide/fragments

Services. Maggiori dettagli sulle tipologie di strumenti, insieme ad opere rilevanti della letteratura, che ne presentano esempi, sono forniti di seguito.

Automation APIs/Frameworks

Le Automation APIs sono strumenti che forniscono mezzi per interagire con la GUI o per ottenere informazioni relative alla GUI per descrivere il contenuto delle schermate e verificare lo stato dell'AUT. I tester in genere sfruttano tali API per annotare manualmente gli script di test, che possono quindi essere avviati e verificati automaticamente. Molti dei framework di automazione sfruttano approcci white o grey box, che estraggono proprietà di alto livello dell'app (ad esempio, l'elenco delle activity e l'elenco degli elementi dell'interfaccia utente contenuti in ogni activity) per generare eventi ed attraversare la GUI [8]. Pur essendo tra gli strumenti più potenti per l'espressività degli script di test sviluppati e per la possibilità di utilizzare tali script anche per il Regression Testing, il principale difetto dei GUI Automation Framework, come verrà dettagliato in seguito, è l'elevatissimo costo di manutenzione durante la normale evoluzione dell'App a cui sono associati i test. Due strumenti di test sviluppati ufficialmente da Android, Espresso[14] e UI Automator[15] sono tra i più diffusi Automation Framework ed API. UI Automation[16] è una controparte di UI Automator progettata per testare la GUI delle app iOS; il suo motore di automazione è ora la base per Appium[17], uno strumento di test multi-piattaforma in grado di automatizzare sia le app iOS che Android. Un'altra alternativa open source ampiamente adottata è Robolectric[18] [9]. Diverse Automation API commerciali, come Quantum[19] e Qmetry[20], offrono ai tester la possibilità di scrivere test cases in linguaggio naturale.

Record and Replay Tools

Gli strumenti Record & Replay offrono ai tester/sviluppatori la possibilità di generare script di test catturando le interazioni eseguite sull'AUT durante un'esecuzione dei suoi scenari di utilizzo da testare. Gli importanti vantaggi esibiti dagli strumenti Record & Replay sono la registrazione dei test case dal punto di vista dell'utente

[14] http://developer.android.com/training/testing/ui-testing/espresso-testing

[15] http://developer.android.com/training/testing/ui-automator

[16] https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html

[17] http://appium.io/

[18] http://robolectric.org/

[19] https://www.perfecto.io/integrations/quantum

[20] http://qmetry.github.io/qaf/

finale, il basso sforzo richiesto per la creazione di script di test rispetto alla scrittura manuale di script utilizzando API GUI Automation e la possibilità di creare test senza avere informazioni sull'implementazione dell'applicazione. Inoltre, queste tecniche espongono diverse carenze quando si tratta dell'accuratezza e della portabilità degli script di test generati. Molti degli strumenti Record & Replay disponibili sono concepiti come estensioni delle API GUI automation esistenti, per fornire un altro modo di creare script di test: è il caso di Espresso Test Recorder[21], Xamarin Test Recorder[22]. Altri esempi di strumenti di test citati che sfruttano tale approccio sono RERAN [10], VALERA [11], Mosaic [12], Barista [13], ODBR [14] e SPAG-C [15].

Tecniche automatizzate di generazione di input di test

Molti strumenti disponibili vengono utilizzati per generare sequenze di input per le applicazioni da testare. Il più delle volte vengono utilizzati oracoli impliciti (ad es. innescando arresti anomali nell'AUT vengono considerati test case falliti). La generazione automatica di input può essere vista come un modo per ridurre lo sforzo ed il costo della scrittura manuale di script di test per i framework di automazione GUI o per l'acquisizione di sequenze di interazioni con l'AUT. Tuttavia, come riportato da Choudary et al. [8], gli strumenti disponibili in letteratura espongono ancora diversi problemi soprattutto in termini di efficienza nella ricerca di bug.

La generazione di input, nella sua forma più semplice, può essere casuale. Monkey[23] è un tester casuale ufficiale fornito da Android; un altro esempio è Dynodroid [16].

Le tecniche di generazione di input sono invece dette sistematiche quando gli input non vengono generati in modo casuale, ma al fine di massimizzare alcune funzioni di copertura (ad esempio, code coverage[24] o Activity coverage). Esempi tratti dalla letteratura di approcci sistematici di test per la generazione di input sono AndroidRipper [1] e CrashScope [17].

Le tecniche di Model-Based Input Generation definiscono sequenze di input secondo un modello dell'interfaccia utente, che può essere fornito dal tester/sviluppatore o ottenuto automaticamente dallo strumento stesso. Alcuni esempi sono MobiGUItar [18] o Swifthand [19].

Diversi strumenti recenti hanno adottato l'approccio basato sulla ricerca del test, adottando la meta-euristica (come gli algoritmi genetici) per automatizzare o

[21] https://developer.android.com/studio/test/espresso-test-recorder

[22] http://www.xamarin.com/test-cloud/recorder

[23] http://developer.android.com/tools/help/monkey.html

[24] https://en.wikipedia.org/wiki/Code_coverage

parzialmente automatizzare le attività di test, come la generazione di dati di test o sequenze di test [20]: Mahmood et al. hanno presentato EvoDroid, che sfrutta un algoritmo evolutivo per generare casi di test [21]; Jabbarvand et al. hanno descritto due algoritmi per la minimizzazione della suite di test energy-aware [22]; Mao et al. hanno introdotto SAPIENZ, un approccio multi-obiettivo basato sulla ricerca per la minimizzazione della lunghezza della sequenza di test, della rivelazione dei guasti e della copertura [23]. Lo strumento SAPIENZ ha avuto impatti industriali significativi ed è stato ufficialmente adottato da Facebook per testare la loro app mobile[25].

Strumenti per il monitoraggio e la segnalazione di bug ed errori

In questa categoria sono considerati tutti gli strumenti utilizzati per tracciare il comportamento imprevisto delle app mobile, sia tramite i rapporti dell'utente sia attraverso meccanismi automatizzati di rilevamento dei crash. Esempi di questi strumenti sono ODBR [14], che sfrutta il framework UI Automator per documentare sequenze di input che possono portare ad arresti anomali in una determinata app Android, e FUSION [24], che collega le informazioni sull'esperienza utente con l'app fornita dall'utente stesso mediante una analisi del programma eseguita automaticamente.

Servizi per i test Mobile

Sono disponibili, infine, diversi servizi di test online per le app mobile, in genere destinati ad affrontare problemi quali la diversità dei dispositivi e la compatibilità del sistema operativo, nonché il costo generalmente elevato e lo sforzo richiesto per testare le app mobile. I servizi di test mobile utilizzano in genere set di dispositivi diversi su cui vengono eseguiti test automatizzati e possono essere utilizzati per i test funzionali tradizionali, ma anche per verificare le proprietà non funzionali delle app Mobile come l'usabilità della GUI, la sicurezza, il consumo energetico e la localizzazione.

Per sottolineare l'importanza per l'industria Mobile di tali servizi, vale la pena segnalare che sia Google (con Android Robo Test[26]) che Amazon (con Fuzz Test[27]) hanno recentemente rilasciato un servizio cloud per il test automatizzato delle applicazioni Android.

---

[25] https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale

[26] http://firebase.google.com/docs/test-lab/robo-ux-test

[27] http://docs.aws.amazon.com/devicefarm/latest/developerguide/test-types-built-in-fuzz.html

2.4 Problematiche del Mobile testing

Gli sviluppatori devono, in genere, affrontare una serie di sfide quando creano app per una o più piattaforme mobile.

Uno studio condotto da Joorabchi et al. [25] include tra le difficoltà più rilevanti nelle pratiche di sviluppo mobile:

- la selezione della corretta natura dell'app da sviluppare (ad es. Applicazioni native vs. Web o ibride);
- le capacità limitate del dispositivo medio per una determinata piattaforma;
- la scelta tra riutilizzare il codice altrui o scrivere da zero;
- l'incremento di tempo, sforzi e budget dovuta alla moltitudine di dispositivi e piattaforme su cui le app devono essere in grado di funzionare (ad es. Frammentazione);
- i rapidi cambiamenti dei requisiti ed in genere il rapido ciclo di vita di un'app mobile media;

Le difficoltà incontrate durante lo sviluppo di app mobile (in particolare, quelle Android) si riflettono, ingigantite, nella fase di testing. Il test delle app pobile, inoltre, deve anche tenere conto di diversi aspetti che possono essere completamente trascurati quando si testano le applicazioni desktop tradizionali.

Muccini et al. [3], Kirubakaran et al. [26] e Kaur et al. [5] hanno identificato una serie di caratteristiche delle app mobile che portano a forme specifiche di test non funzionali:

- scenari di connettività mobile (ad esempio, far fronte a connessioni Wi-Fi o 3G/4G/5G inaffidabili) e rapidi cambiamenti di tipologie di connettività;
- risorse limitate dei dispositivi;
- quantità di dati delle applicazioni;
- interruzioni costanti causate dal sistema;
- time-to-market[28] molto breve;

[28]time to market (o TTM) indica il periodo di tempo che intercorre tra l'ideazione di un prodotto e la sua effettiva commercializzazione. Poiché la ricerca ha dimostrato che i nuovi entranti nel mercato godono di chiari vantaggi in termini di quota di mercato, ricavi e crescita delle vendite, il time to market è una delle metriche essenziali per lo sviluppo del prodotto. Molte strategie di sviluppo dipendono dall'essere i primi sul mercato, realizzando quindi una vera e propria corsa contro il tempo, riducendo il tempo dedicato allo sviluppo di test case.

- quantità molto elevata di multi-tasking e comunicazione con altre app.

A causa delle difficoltà elencate, c'è una sostanziale unanimità su una tendenza generale degli sviluppatori Android a trascurare i test automatizzati e ad affidarsi invece solo ai test manuali. Come emerge da una serie di interviste ai contributori di progetti open source eseguite da Linares-Vasquez et al. [27] e da Kochhar et al. [28], i vincoli di tempo, la mancanza di strumenti di test adeguatamente documentati e gli alti costi per lo sviluppo e la gestione dei test sono le ragioni principali di tale preferenza verso le procedure di test manuali.

2.4.1 Frammentazione

Il concetto di frammentazione comprende, per l'ecosistema Android, due diverse problematiche [29].

La *frammentazione hardware* si riferisce al fatto che i dispositivi basati sullo stesso sistema operativo Android funzionano su diversi processori, schede grafiche, dimensioni dello schermo e densità di pixel. Secondo un rapporto dell'agosto 2015[29], all'epoca erano esistenti più di 24 mila dispositivi diversi, costruiti da oltre 12 centinaia di fornitori, e si potevano trovare molte dimensioni di visualizzazione, rapporti e densità di pixel (vedi figura 2.1).

La *frammentazione software* si riferisce al fatto che diverse versioni del sistema operativo Android esistono in parallelo e che, allo stesso tempo, fornitori ed operatori possono offrire personalizzazioni per app e GUI del sistema operativo. La frammentazione dei dispositivi è un problema specifico dell'ecosistema Android (vedi figura 2.2) e non dello sviluppo mobile in generale, essendo il numero di dispositivi disponibili e le versioni mantenute del sistema operativo per le app iOS molto limitate.

La frammentazione dei dispositivi ha un impatto rilevante su molti aspetti dello sviluppo mobile. Prima di tutto, per far fronte alla frammentazione del software del sistema operativo, gli sviluppatori devono far fronte a metodi deprecati o addirittura rimossi del framework che utilizzano; pertanto, potrebbe essere necessario sviluppare le app in modo diverso a seconda delle versioni del sistema operativo sul quale saranno eseguite [30][31]. La frammentazione software, soprattutto per quanto riguarda le versioni personalizzate del sistema operativo di diversi fornitori, crea anche preoccupazioni sulla sicurezza delle app [32].

Dal punto di vista del test di sistema e GUI delle app Android, la frammentazione si traduce, nello specifico, nella necessità di verificare la corretta inflazione dei layout, che possono essere diversi per rispettare le dimensioni dello schermo, la

[29] https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf

densità e l'orientamento del dispositivo (poiché la stessa activity può adottare layout diversi in base all'orientamento corrente del dispositivo: cfr. figure 2.3 e 2.4). Le tecniche di test basate sul layout, o di 2a generazione, devono quindi considerare i possibili widget diversi dei layout utilizzati per popolare le attività. Le tecniche di test visive, o di 3a generazione, devono far fronte agli elementi dell'interfaccia che potrebbero non apparire sullo schermo, o apparire a risoluzioni diverse e quindi invalidare il corretto riconoscimento delle catture dello schermo utilizzate come identificatori visual-based.

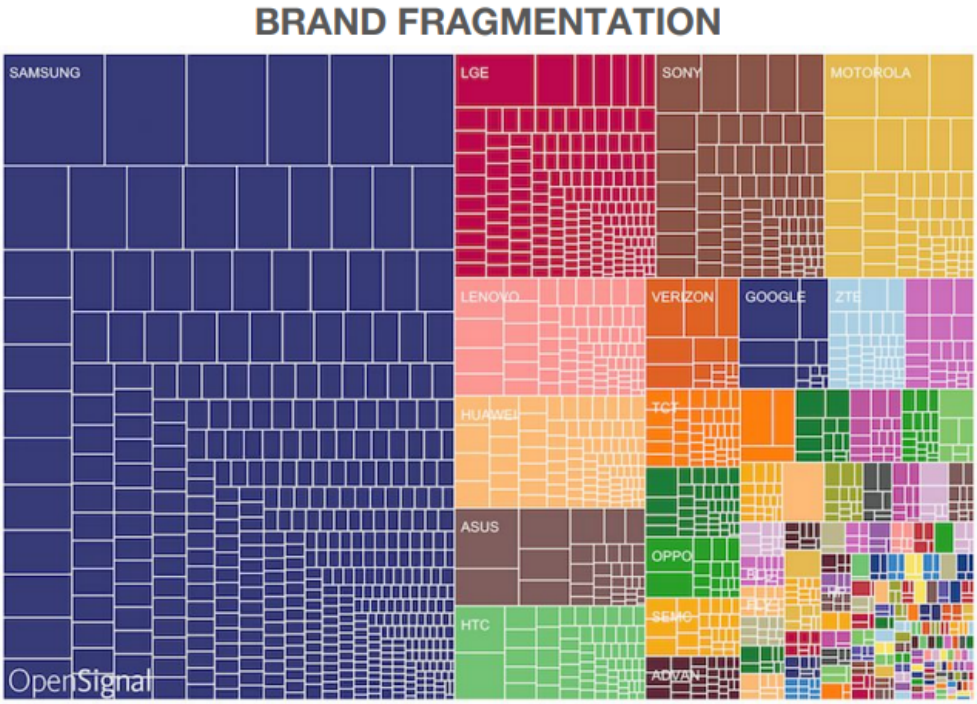


Immagine 2.1: Dimensione degli schermi di dispositivi Android dell'agosto 2015[30]

[30] https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf

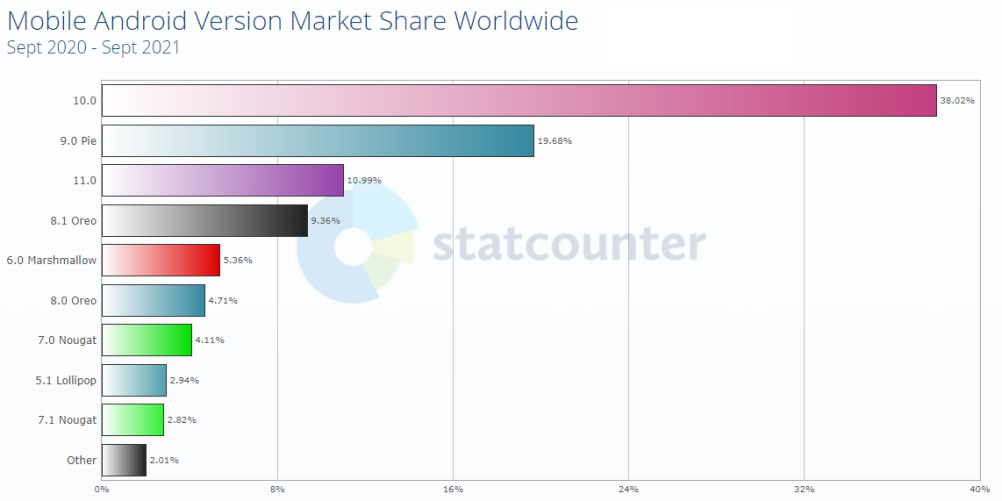


Immagine 2.2: Versione del OS Android tra settembre 2020 e settembre 2021[31]

Immagine 2.3: TextView che si adatta alla rotazione dello schermo[32]

[31] https://gs.statcounter.com/android-version-market-share/mobile/worldwide/monthly-202009-202109

[32] https://developer.android.com/training/multiscreen/screensizes

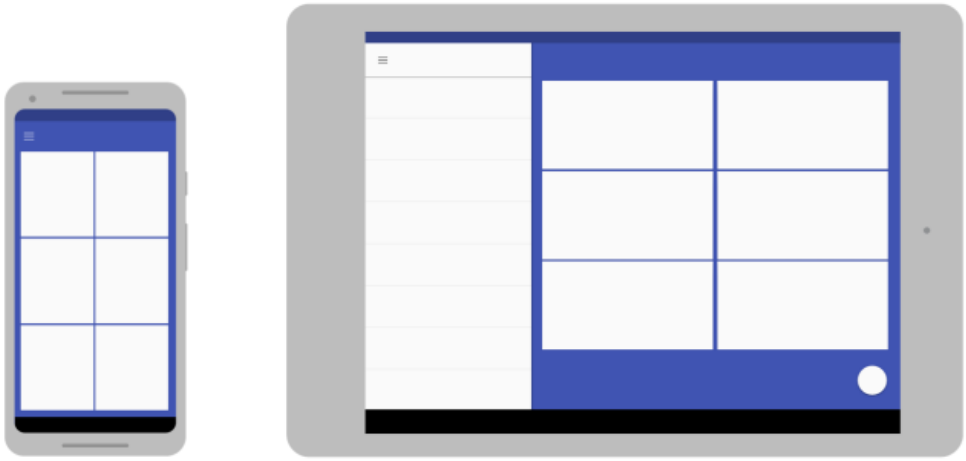


Immagine 2.4: Activity che in base alla dimensione dello schermo e/o rotazione assume due layout completamente diversi [33]

2.4.2 Testing di applicazioni Ibride e Web-Based

Lo sviluppo di applicazioni ibride è utile quando i piccoli team non sono in grado di creare e mantenere diverse basi di codice per app destinate a funzionare su piattaforme diverse. Sono disponibili diversi framework per la creazione di app ibride, ad esempio React, Ionic e Flutter di Google.

Sebbene meno inclini al problema della frammentazione, le app Web, anche se già testate per i normali browser, dovrebbero essere testate in modo specifico in ambienti mobile per quanto riguarda l'utilizzo della batteria, i problemi di connettività e le prestazioni [33]. Fondamentale è anche il test della GUI delle applicazioni web, al fine di verificare che tutti gli elementi dell'interfaccia utente siano visibili, allineati e renderizzati esattamente come nell'interfaccia del browser. Una limitazione per testare le applicazioni Web in ambiente Android è la parziale inapplicabilità dei GUI Automation Framework specifici per Android, ed in generale degli strumenti di test layout-based, con molti degli strumenti di test più diffusi (come Espresso o Robotium) che forniscono un supporto praticamente nullo a ciò che viene caricato all'interno di WebViews[34].

Come riportato da Ahmad et al. [34], la mancanza di accesso alle funzionalità della piattaforma, i cambiamenti nei fattori contestuali o ambientali delle app, i

[33] https://developer.android.com/training/multiscreen/screensizes

[34] https://developer.android.com/reference/android/webkit/WebView

problemi di integrazione e conformità e la diversità nell'interfaccia utente aggiungono altri livelli di complessità e frammentazione quando si testano le app ibride. Gli approcci Uniti, come Squish[35], sono in genere a livello di astrazione esclusivo della GUI (strumenti di terza generazione), e sfruttano gli approcci Capture & Replay per le generazioni di test case.

2.5 Manutenzione dei test automatizzati

Una delle maggiori sfide dell'automazione dei test, in generale, è quella di tenere il passo con i cambiamenti dell'AUT. La manutenzione rappresenta un costo significativo indipendentemente dallo strumento di test automatizzato utilizzato, che cresce con le dimensioni della suite di test e con la copertura delle funzionalità dell'AUT. La manutenzione degli script di test è necessaria per mantenerli allineati con i diversi requisiti dell'AUT, alla sua progettazione o semplicemente al suo comportamento previsto. Si prevede che i casi di test GUI dovranno affrontare costi di manutenzione rilevanti, poiché i test GUI sono influenzati da qualsiasi modifica in ognuno dei livelli di astrazione che sono alla base della GUI e gli scenari di test eseguiti tramite la GUI sono in genere più lunghi degli unit test basati su singole classi o moduli delle applicazioni.

La manutenzione del testware ha un costo crescente con la quantità di codice modificato nell'AUT e spesso richiede un certo livello di conoscenza dei dettagli di implementazione modificati, al fine di far eseguire ai test case gli stessi test sulla nuova versione dell'AUT. Il carico di manutenzione del testware automatizzato, inoltre, aumenta in genere con il tempo, poiché la quantità relativa di codice di test rispetto al codice di produzione tende ad aumentare nel tempo. Come sottolineato in un lavoro di Berner et al. [35] (che ha anche definito un modello teorico dei costi per la manutenzione dei test automatizzati), la manutenzione del testware può anche dipendere da fattori diversi dalla quantità e dalla frequenza delle modifiche eseguite sull'AUT: tra questi, citano:

- l'architettura mancante per la generazione dei casi di test;
- la reimplementazione di azioni ripetitive attraverso test cases anziché il riutilizzo;
- scarsa organizzazione del codice di test e dei dati di test;
- mancata verifica del testware stesso;

[35] https://www.froglogic.com/squish/

Alegroth et al., che hanno eseguito un'analisi empirica dei fattori che contribuiscono al mantenimento dei test cases GUI, hanno identificato una serie di tredici fattori, che vanno da quelli tecnici (ad esempio, somiglianza del test case, lunghezza del test case e presenza di loop e flussi nei test cases) a quelli umani (ad esempio, mentalità sequenziale alla descrizione del test case), che contribuiscono a tali costi. L'applicabilità di una tecnica di test a un contesto reale, quindi, dipende principalmente dalla probabilità di un ROI[36] positivo quando l'AUT subisce una quantità ragionevole di modifiche. Nei contesti industriali, inoltre, è fondamentale una corretta formazione di sviluppatori e tester sui possibili costi di manutenzione del testware: come riporta Fewster, se la manutenzione dell'automazione del testware viene ignorata e non eseguita il prima possibile per conformarsi alle modifiche dell'AUT, l'aggiornamento di un'intera suite di test può costare tanto quanto o anche di più che rieseguire manualmente tutti i test case [36].

Diversi studi hanno misurato i costi e i benefici delle tecniche di test GUI all'avanguardia e li hanno confrontati con i costi dei test manuali. Un caso di studio di Andersson e Pareto, ad esempio, ha confrontato i costi di manutenzione necessari per le tecniche di test Capture & Replay per il test della GUI delle applicazioni desktop [37]. Dallo studio è emerso che, se le applicazioni hanno rilasci frequenti, l'adozione di tecniche di test automatizzate può diventare un onere per gli sviluppatori piuttosto che un vantaggio, a causa di un rapporto costo/efficacia inferiore rispetto ai test di regressione manuali. Diversi scritti correlati hanno indagato l'applicabilità di varie tecniche di test in contesti industriali: esempi sono il lavoro di Borjesson e Feldt [38] e di Alegroth et al. [39], che hanno valutato i vantaggi dell'adozione di tecniche di Visual GUI Testing rispettivamente presso Saab e Siemens; Nguyen et al. hanno considerato i costi di manutenzione tra i fattori più importanti nella progettazione del loro strumento basato su modelli GUITAR [40].

2.5.1 Fragilità dei test della GUI

I test fragili, come definiti da Garousi et al. [41], sono test che vengono interrotti durante l'evoluzione di un'applicazione da modifiche che non sono correlate alla logica di test stessa o alle caratteristiche specifiche che esercitano. La fragilità dei test rappresenta un problema di manutenzione significativo per il testware di tutti i domini ed è stata ampiamente esplorata in letteratura. Grechanik et al. [42] e Memon et al. [43] propongono approcci per correggere automaticamente i casi di test non più funzioannti per le applicazioni basate su GUI; Gao et al. hanno sviluppato SITAR [44], una tecnica per riparare automaticamente le suite di test,

[36]ROI: Ritorno sull'investimento.

modellando e riparando i casi di test utilizzando degli Event-Flow Graphs (EFG); Leotta et al. [45, 46] hanno riportato i risultati di uno studio empirico sulla fragilità dei test GUI per le applicazioni web. Attraverso gli studi riportati in questa tesi, è stata adottata la seguente definizione per i test GUI fragili: Un test case per la GUI è fragile se richiede interventi man mano che l'applicazione evolve (cioè tra le versioni successive) a causa di qualsiasi modifica applicata all'AUT.

Il test GUI differisce significativamente dal test del software tradizionale. Essendo test a livello di sistema, i test cases sviluppati con framework di automazione GUI possono essere influenzati da variazioni nelle funzionalità dell'app, ma anche da piccoli interventi nell'aspetto e nella presentazione delle schermate da cui è composta la GUI. Un gran numero di esecuzioni di test case, può portare a malfunzionamenti se si riferiscono a elementi della GUI che sono stati rinominati, spostati o alterati [47, 48]. La definizione adottata distingue (come fragili vs. non fragili) i test che necessitano di interventi a causa di qualsiasi tipo di modifica nell'AUT dai test che non richiedono modifiche a causa dell'evoluzione dell'app, ma che invece sono solo soggetti a variazioni nella logica di test o nelle funzioni che sono proprie dei GUI Automation Framework adottati. L'assunto alla base dell'analisi presentata in questa tesi è che i casi di test mobile, per i quali la fragilità non è stata precedentemente esplorata da studi accademici su larga scala, possono essere fortemente soggetti a fragilità, perché:

1. le app mobili si basano principalmente sulle loro GUI per tutte le interazioni con gli utenti e la presentazione dei dati;
2. le GUI mobili sono soggette a frequenti modifiche durante la durata dell'app;
3. pur avendo diverse somiglianze con le app basate sul web, le GUI per dispositivi mobili sono descritte in modi specifici e le loro proprietà di funzionalità di layout che non sono presenti in altri domini.

Capitolo 3

Metriche per la misurazione della fragilità

3.1 Misurazione della Fragilità

La fragilità dei test può essere un problema per diversi tipi di software. In generale, si dice che un caso di test sia fragile quando fallisce o ha bisogno di manutenzione a causa della naturale evoluzione dell'AUT, mentre le funzionalità specifiche che testa non sono state alterate. Sono state effettuate indagini nel campo dei test delle applicazioni web con lo sforzo di Leotta et al. volte a confrontare la robustezza dei casi di capture & replay rispetto sia ai casi di test programmabili [45] sia ai test scritti utilizzando diversi localizzatori per i componenti dell'AUT [46]. Un elenco delle possibili cause di fragilità specifiche del GUI testing delle applicazioni mobile è il seguente:

- modifiche all'ID e al testo all'interno della gerarchia visiva delle activity;
- cancellazione o ricollocazione degli elementi della GUI;
- utilizzo di pulsanti fisici;
- cambiamenti nel layout e nell'aspetto grafico, soprattutto se vengono utilizzati strumenti di riconoscimento visivo per generare oracoli utilizzabili nel testing;
- adattamento a diversi modelli di hardware e dispositivi;
- variazioni del flusso di attività;
- variabilità dei tempi di esecuzione.

Le modifiche eseguite sui casi di test possono essere dovute a diversi motivi. Yusifoglu et al. [49] classificano le modifiche del codice di test in quattro categorie:

1. *manutenzione perfettiva*: quando il codice di test viene rifattorizzato per migliorarne la qualità (ad esempio, per aumentare la copertura o adottare modelli di test ben noti);

2. *manutenzione adattiva*: per far evolvere il codice di test in base all'evoluzione del codice di produzione;

3. *manutenzione preventiva*: per modificare aspetti del codice che potrebbero richiedere un intervento nelle versioni future;

4. *manutenzione correttiva*: per eseguire correzioni di bug.

Secondo la nostra definizione di fragilità del test GUI, siamo interessati ai casi di manutenzione adattiva.

Per implementare la classificazione dei test come fragili o meno in uno strumento automatizzato, si presume, come viene comunemente fatto per i test basati su JUnit, che ogni caso di test sia descritto da un singolo metodo di test . Si definisce classe di test, una raccolta di metodi di test in un singolo file Java. Si considera fragile qualsiasi metodo modificato all'interno di una classe di test GUI. Quando una classe di test viene modificata, la si considera non frammentata se non ci sono metodi modificati al suo interno; ad esempio, le modifiche possono riguardare solo istruzioni di importazione. I costruttori di test o i metodi di test possono essere stati aggiunti o rimossi ma non modificati. Si suppone che l'aggiunta di un nuovo metodo debba riflettere l'introduzione di nuove funzionalità o nuovi casi d'uso da testare nell'applicazione, e non la modifica di elementi esistenti delle attività già testate. D'altra parte, se alcune righe di codice all'interno di un singolo metodo di test dovevano essere modificate o aggiunte, è più probabile che i test debbano essere modificati a causa di piccoli cambiamenti nell'applicazione e possibilmente nella sua GUI (ad esempio, modifiche nella gerarchia dello schermo e nelle transizioni tra le activity).

3.2 Definizione delle metriche

Per classificare le modifiche nel codice di test, alcune metriche di modifica sono già state definite, ad esempio, Tang et al. [50], definiscono un insieme di 18 metriche con l'obiettivo di descrivere le cronologie delle modifiche di correzione dei bug nei file sorgente.

Tang et al. descrivono tre diverse categorie di metriche:

- *dimensione*: quantità di righe di codice aggiunte o rimosse, numero di classi, file o metodi modificati;

- *atomica*: valori booleani che indicano se una classe presenta metodi aggiunti;

- *semantica*: numero di dipendenze aggiunte o rimosse all'interno di un file.

In questa sezione, si introducono alcune metriche per dare una caratterizzazione all'adozione di singoli strumenti tra i repository open source Android e per valorizzare la quantità di modifiche eseguite sui test cases presenti nei progetti. All'insieme di metriche assolute fornite da Tang et al., la tesi aggiunge una serie di metriche che permettono di eseguire indagini sulla volatilità e la fragilità delle classi e metodi di test. Queste, mirano a catturare il peso che ogni singola modifica nelle classi o nei metodi di test ha rispetto all'intera quantità di codice di test dell'applicazione. La maggior parte delle metriche sono normalizzate per consentire confronti tra progetti di diverse dimensioni; sono normalizzate in relazione alle dimensioni del set di test, alla durata di vita delle singole classi di test e alla quantità di modifiche eseguite al codice. Le metriche che sono state definite, come dettagliato più avanti nella procedura di calcolo, richiedono classi e metodi di test scritti in Java, in modo che sia possibile un confronto tra codice di test e codice di produzione. Le metriche, pertanto, non sono applicabili su applicazioni che utilizzando linguaggi differenti.

Le metriche che sono state introdotte, possono essere definite come metriche composte basate su metriche di cambiamento già esistenti. Ad esempio, considerando di nuovo le metriche e la nomenclatura fornite da Tang et al., *Tdiff* (quantità di righe di codice aggiunte, cancellate o modificate di file relativi a uno specifico strumento di test), può essere calcolata come somma di *LA* (cioè righe di codice aggiunte) e *LD* (righe di codice cancellate) per i file che contengono script generati con quel dato strumento; inoltre, *Pdiff* (quantità di righe di codice aggiunte, eliminate o modificate su cui basiamo molte delle nostre metriche), può essere calcolata come somma di *LA* e *LD* per tutti i file della release. La metrica *MRTL (Modified Relative Test LOCs)* può essere calcolata come il rapporto tra le due somme citate sopra. Le metriche possono essere suddivise in tre gruppi. La figura 3.1 mostra tutte le definizioni delle metriche, il loro tipo e gli intervalli a cui appartengono. Le metriche sono spiegate in dettaglio di seguito.

TABLE I
METRIC DEFINITION

Group	Name	Explanation	Type	Range
Adoption and size	TA	Tool Adoption	Real	(0, 1)
	NTR	Number of Tagged Releases	Integer	[2, ∞)
	NTC	Number of Tool Classes	Integer	[1, ∞)
	TTL	Total Tool LOCs	Integer	[1, ∞)
Test evolution	TLR	Tool LOCs Ratio	Real	(0, 1]
	MTLR	Modified Tool LOCs Ratio	Real	[0, ∞)
	MRTL	Modified Relative Tool LOCs	Real	[0, 1]
	TMR	Tool Modification Relevance Ratio	Real	[0, ∞)
	MRR	Modified Releases Ratio	Real	[0, 1]
	TCV	Tool Class Volatility	Real	[0, 1]
	TSV	Tool Suite Volatility	Real	[0, 1]
	TJR	Tool Code to JUnit code Ratio	Real	[0, ∞)
	MTJR	Modifications of Tool code to JUnit code Ratio	Real	[0, ∞)
Fragility	MCR	Modified Tool Classes Ratio	Real	[0, 1]
	MMR	Modified Tool Methods Ratio	Real	[0, 1]
	FCR	Fragile Classes Ratio	Real	[0, 1]
	RFCR	Relative Fragile Classes Ratio	Real	[0, 1]
	FRR	Fragile Releases Ratio	Real	[0, 1]
	ADRR	Releases with Added-Deleted Methods Ratio	Real	[0, 1]
	TCFF	Tool Class Fragility Frequence	Real	[0, 1]
	TSF	Tool Suite Fragility	Real	[0, 1]

Immagine 3.1: Metriche Principali

1. *Adoption and size*: per stimare l'adozione di strumenti di test GUI automatizzati Android tra progetti open source e per calcolare le dimensioni delle suite di test che li utilizzano, definiamo le seguenti metriche:

   - *Number of Tagged Releases (NTR)*: numero di versioni con tag di un progetto (elencate utilizzando il comando *git tag* nel repository GIT). Questa metrica può dare un'idea di quali tipi di applicazioni (siano esse piccole app sviluppate per il rilascio immediato e poi abbandonate, o progetti di lunga durata) hanno maggiori probabilità di avere le loro GUI testate con i framework di automazione GUI. Viene utilizzata per identificare i progetti forniti con una cronologia di rilasci da studiare con le altre metriche.
   - *Number of Tool Classes (NTC)*: numero di classi presenti in una versione di un progetto, con codice relativo a uno strumento specifico (le classi sono associate a un determinato strumento di test se contengono importazioni o chiamate di metodi specifici dello strumento).
   - *Total Tool LOCs (TTL)* : numero di righe di codice appartenenti a classi che possono essere attribuite a uno specifico strumento di test in una versione di un progetto. Questa metrica, insieme alla precedente, consente di quantificare, all'interno di progetti Android, il codice che può essere

associata a una serie di framework di test rilevanti, utilizzati per il test della GUI.

2. *Test Evolution*: Le metriche che rispondono a *TE*, mirano a descrivere l'evoluzione dei progetti open source e delle rispettive suite di test; sono calcolate per ogni versione o per ogni coppia di versioni con tag consecutivi.

   - Il rapporto *Tool LOCs Ratio (TLR)* è definito come

   $$TLR_i = \frac{TTL_i}{Plocs_i}$$

   dove $Plocs_i$ è la quantità totale di Linee di Codice di produzione per la release *i*. Questa metrica, che si trova nell'intervallo [0, 1], consente di quantificare la rilevanza del codice di test associato a uno strumento specifico.

   - *Modified Tool LOCs Ratio (MTLR)* è definito come

   $$MTLR_i = \frac{Tdiff_i}{Tlocs_{i-1}}$$

   dove $Tdiff_i$ è la quantità di LOC aggiunte, eliminate o modificate in classi che possono essere associate a uno strumento specifico tra le versioni con tag *i-1* e *i*. Questo quantifica le modifiche eseguite su LOC esistenti che possono essere associate a un determinato strumento per una versione specifica di un progetto. Un valore superiore a 1 di questa metrica significa che più righe vengono aggiunte, modificate o rimosse nelle classi di test nella transizione tra due versioni con tag consecutive rispetto al numero di righe già presenti da esse.

   - *Modified Relative Test LOCs (MRTL)* è definito come

   $$MRTL_i = \frac{Tdiff_i}{Pdiff_i}$$

   dove $Tdiff_i$ e $Pdiff_i$ sono, rispettivamente, la quantità di strumenti e LOC di produzione aggiunte, cancellate o modificate nella transizione tra le versioni *i-1* e *i*. Viene calcolato solo per le versioni con codice associato a un determinato strumento di test. Questa metrica si trova nell'intervallo [0, 1]; valori vicini a 1 implicano che una parte significativa del tasso di abbandono totale del codice durante l'evoluzione dell'applicazione è necessaria per mantenere aggiornati i test case scritti con uno strumento specifico.

- *Tool Modification Relevance ratio (TMR)* è definito come

$$TMR_i = \frac{MRTL_i}{TLR_{i-1}}$$

Questo rapporto può essere utilizzato come indicatore della porzione di abbandono del codice necessaria per adattare le classi relative a un determinato strumento di test durante l'evoluzione dell'applicazione. Viene calcolato solo quando $TLR_{i-1} > 0$. Valori maggiori di 1, indicano un maggiore sforzo necessario per modificare il codice di test rispetto alla modifica del codice dell'applicazione. Valori più bassi di questo indicatore, provano una più facile adattabilità del codice associato a un determinato strumento di test alle modifiche nell'AUT. Il *rapporto di rilascio modificato (MRR)* viene calcolato come il rapporto tra il *NTR* in cui è stata modificata almeno una classe associata a uno strumento di test specifico e la quantità totale di versioni con tag con classi associate a tale strumento. Questa metrica si trova nell'intervallo [0, 1]; valori elevati indicano una minore adattabilità della suite di test (insieme di classi di test associate a un determinato strumento di test) ai cambiamenti nell'AUT.

- *Tool Class Volatility (TCV)* può essere calcolato per ogni classe associata a un determinato strumento come:

$$TCV_j = \frac{Mods_j}{Lifespan_j}$$

dove $Mods_j$ è la quantità di release nelle quali la classe j viene modificata e $Lifespan_j$ è il numero di release totali dell'applicazione con la classe j. La Tool class volatility (TSV) è definita per ogni progetto come il rapporto tra il numero di classi associate a un determinato strumento che vengono modificate almeno una volta nel corso della loro durata e il numero totale di classi associate a tale strumento nella cronologia del progetto.

- *Tool code to JUnit code Ratio (TJR)* può essere calcolato per ogni versione di un progetto come

$$TJR = \frac{Tlocs_i}{Jlocs_i}$$

dove $Tlocs_i$ è il numero di righe di codice associate allo strumento considerato nella release i, e $Jlocs_i$ è il numero di righe di codice associate ad altri test JUnit nella release i. La metrica non è definita quando $Jlocs_i = 0$, cioè quando nessun codice di test JUnit è presente nella release i. Un valore elevato di questa metrica implica che la versione i contiene più codice associato allo strumento fornito rispetto ad altri codici di test generici che sfruttano il framework JUnit.

- *Modifications of tool code to JUnit code ratio (MTJR)* può essere calcolato per ogni versione di un progetto come

$$MTJR_i = \frac{Tdiff_i}{Jdiff_i}$$

dove $Tdiff_i$ è la quantità di codice aggiunto, eliminato o modificato associato a un determinato strumento tra le versioni con tag *i-1* e *i*, e $Jdiff_i$ è la quantità di righe di codice aggiunte, cancellate o modificate associate ad altri test JUnit tra le versioni con tag *i-1* e *i*. La metrica non è definita quando $Jdiff_i = 0$, cioè quando nessuna riga di codice nelle classi associate a JUnit viene modificata nella transizione dalla release *i-1* e *i*. Un valore elevato per questa metrica significa che tra le versioni *i-1* e *i*, sono stati eseguiti più interventi sul codice associato allo strumento dato rispetto ad altro codice di test che sfrutta il framework JUnit.

3. *Fragilità delle classi e dei metodi*: con un'ispezione automatizzata del codice di prova, è possibile ottenere informazioni sui metodi e sulle classi modificate. Sulla base di tali dati, le metriche fornite mirano a ottenere una caratterizzazione approssimativa della fragilità delle suite di test. Il numero di classi alterate con metodi modificati può essere diverso dal numero totale di classi modificate in tre casistiche diverse (e nelle combinazioni delle tre):

   - quando le modifiche apportate alle classi coinvolgono porzioni insignificanti di codice come commenti, importazioni o dichiarazioni;
   - quando le modifiche apportate alle classi comportano solo aggiunte di metodi di prova;
   - quando le modifiche apportate alle classi comportano solo la rimozione dei metodi di prova.

   Le aggiunte e le rimozioni di metodi di prova sono considerate la conseguenza di una nuova funzionalità o di un nuovo caso d'uso dell'applicazione; pertanto, non sono considerati come prova di fragilità delle classi di test. D'altra parte, i metodi di test modificati, possono riflettere cambiamenti nelle funzionalità dell'applicazione o nella definizione della sua GUI e quindi rendere fragili le classi di test che li contengono secondo la definizione.

   - *Modified tool Classes Ratio (MCR)* è definito come

$$MCR_i = \frac{MC_i}{NTC_{i-1}}$$

dove MC_i è il numero di classi associate a un dato strumento di test che vengono modificate nella transizione tra le versioni *i-1* e *i*; NTC_{i-1} è il

numero di classi associate allo strumento nella versione *i-1* (la metrica non è definita quando $NTC_{i-1} = 0$). La metrica si trova nell'intervallo [0, 1]: più grandi sono i valori, meno le classi sono stabili durante l'evoluzione dell'app.

- *Modified tool Methods Ratio (MMR)* è definito come

$$MMR_i = \frac{MM_i}{TM_{i-1}}$$

dove MM_i è il numero di metodi in classi associate a un dato strumento che vengono modificati tra le versioni *i-1* e *i* e TM_{i-1} è il numero totale di metodi nelle classi associate allo strumento nella versione *i-1* (la metrica non è definita quando $TM_{i-1} = 0$). La metrica si trova nell'intervallo [0, 1]: maggiori sono i valori di MMR, minore è la stabilità dei metodi durante l'evoluzione dell'app.

- *Fragile Classes Ratio (FCR)* è definito come

$$FCR_i = \frac{MCMM_i}{NTC_{i-1}}$$

dove $MCMM_i$ è il numero di classi associate a un dato strumento di test che presentano almeno un metodo modificato tra le release *i-1* e *1*. Questa metrica rappresenta una stima della percentuale di classi fragili associate allo strumento rispetto all'intero set di classi di test presenti in una versione. La metrica è delimitata da MCR, poiché per sua definizione $MCR_i = MC_i/TC_i$ e $MCMM_i \leq MC_i$.

- *Relative Fragile Classes Ratio (RFCR)* è definito come

$$RFCR_i = \frac{MCMM_i}{MC_i}$$

dove $MCMM_i$ e MC_i sono definiti come sopra.

Il *Fragile Releases Ratio (FRR)* è calcolato come il rapporto tra il *NTR* con almeno una classe fragile tra quelle associate a un determinato strumento e la quantità totale di versioni con tag contenenti classi di test associate a tale strumento. Questa metrica si trova nell'intervallo [0, 1] ed è limitata in alto da *MRR*. Il rapporto *ADRR (Releases with Added-Deleted methods Ratio)* viene calcolato come il rapporto tra il *NTR* in cui almeno un metodo è stato aggiunto o rimosso nelle classi associate a un determinato strumento di test e la quantità totale di versioni con classi di test associate allo strumento. Questa metrica si trova nell'intervallo [0, 1] e valori più elevati implicano cambiamenti più frequenti nelle funzionalità dell'applicazione e casi d'uso definiti da testare.

- Il *Tool Class Fragility Frequence (TCFF)* è definito come

$$TCFF_j = \frac{FR_j}{Lifespan_j}$$

dove FR_j è la quantità di rilasci in cui la classe j, associata a un determinato strumento di test, contiene metodi modificati e $Lifespan_j$ è il numero di rilasci dell'applicazione con la classe j. Questa metrica è delimitata in alto da TCV, poiché, per costruzione, MR_j (numero di rilasci in cui la classe viene modificata) è maggiore o uguale a FR_j .

La *Tool Suite fragility (TSF)* è definita per ogni progetto, come il rapporto tra il numero di classi associate a un determinato strumento che presentano fragilità almeno una volta nel corso della loro durata e il numero totale di classi di test associate allo strumento nella cronologia del progetto.

Capitolo 4

Sviluppo del Software

Per la realizzazione di questa tesi, sono state sviluppate due applicazioni: la prima utilizzando script python e la seconda Java. L'applicazione Python riceve come input l'url del repository di codice open source, esegue i pull di tutte le versioni, calcola le metriche e le inserisce in un file CSV. Tale file verrà successivamente esaminato dall'applicazione Java, che manipola la GUI con cui l'utente interagisce, consentendo la generazione di più grafici e il calcolo di varie funzioni statistiche. Alcuni dati relativi all'intero progetto sono ottenuti con l'ausilio del software SonarQube.

4.1 Script python per analisi delle release

Il core del software di tesi é composto da una coppia di script python i quali, dato in input l'url del repository GitHub che si vuole analizzare, analizzano tutte le tagged releases[1] e ne calcolano le metriche. Questi script fondamentalmente richiedono alle API di GitHub di generare un file JSON[2] con il numero di tagged release presenti nel repository; una volta ottenuto, inizia un ciclo che permetterà di confrontare ogni release con la precedente per calcolare la variazione delle metriche. I due script si differenziano in quanto uno gestisce l'analisi di tutte le tagged release presenti sin dalla creazione del repository, l'altro solo quelle delle ultime N release

[1] https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases

[2] JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. Si basa su un sottoinsieme del Linguaggio di Programmazione JavaScript, Standard ECMA-262 Terza Edizione - Dicembre 1999.(https://it.wikipedia.org/wiki/JavaScript_Object_Notation)

richieste dall'utente. I due script sono stati separati in due file[3] per semplicità e comprensione.

Per il calcolo di alcune metriche relative all'intero sviluppo del software presente sul repository, gli script si basano su SonarQube[4], uno strumento di revisione automatica per rilevare bug, vulnerabilità e problemi di codice. Quest'ultimo può integrarsi con il flusso di lavoro esistente per consentire l'ispezione continua del codice tra i rami del progetto.

Gli script per l'analisi di tutte le release ricevono in input due parametri: il nome del repository ed il nome della suite di generazione di test case che si vuole analizzare (se lasciato vuoto, la ricerca sarà effettuata su tutti i file contenenti dei test cases a prescindere da se siano stati scritti a mano e/o generati con una suite di testing). Se, invece, si desidera ottenere metriche solo sulle ultime N release, la GUI richiamerà automaticamente gli script continuity inserendo in input questo valore aggiuntivo.

Gli script funzionano eseguendo continui git pull delle nuove versioni e confrontando i file generati dal comando gitdiff, questo permette di ottenere in maniera semplice ed esaustiva tutti i parametri che servono per il calcolo delle metriche.

[3] /input/main.py per analizzare tutte le releases e /input/continuityPython.py per l'analisi solo delle ultime N releases, ma la GUI Java gestirà automaticamente i file senza che l'utente debba interagirci.

[4] https://docs.sonarqube.org/latest/

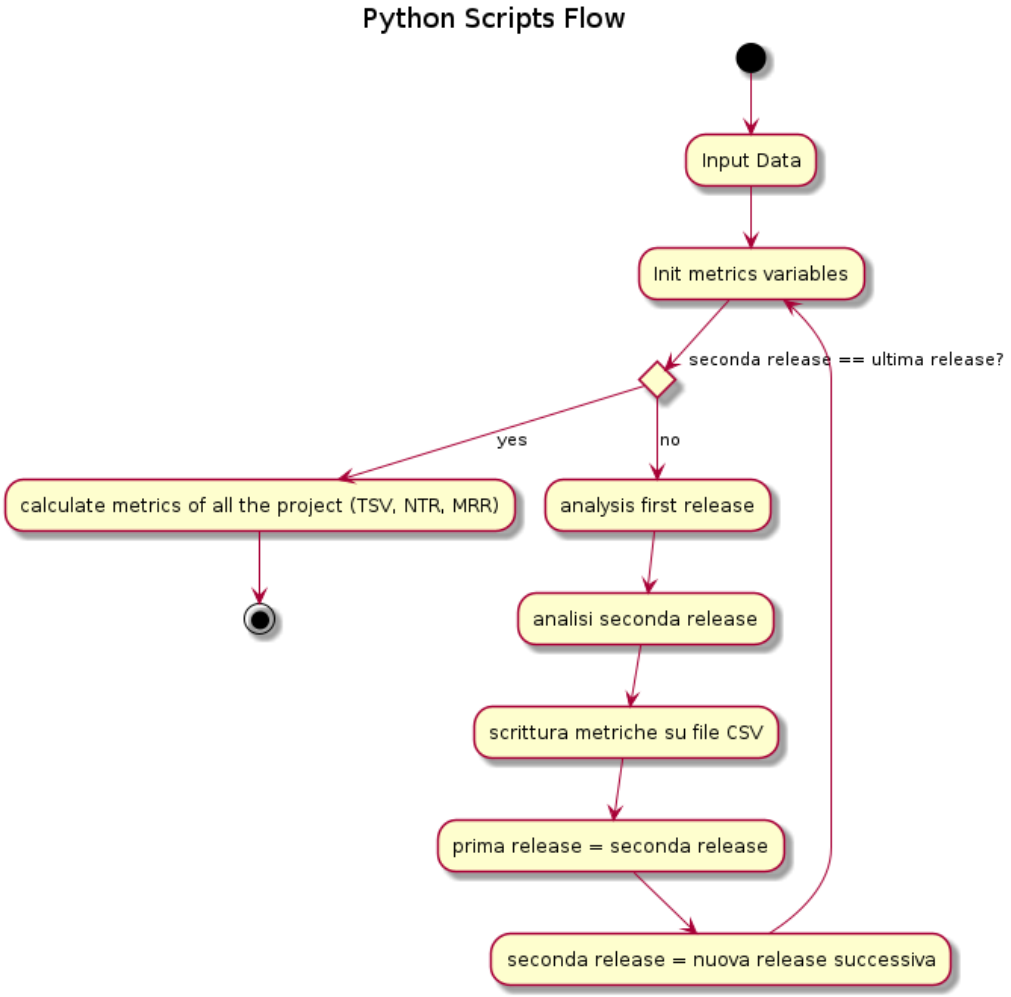


Immagine 4.1: Flow graph rappresentante il funzionamento ad alto livello degli script python

4.1.1 Dipendenze

Per il corretto funzionamento degli script python, sul sistema dovranno essere installate le seguenti librerie:

- *os*[5]: Questo modulo fornisce un modo portatile di utilizzare le funzionalità dipendenti dal sistema operativo, quali, ad esempio, copia, cancellazione o ridenominazione di un singolo file, features per le quali viene utilizzato nella tesi.

[5] https://docs.python.org/3/library/os.html

- *shutil*[6]: Il modulo shutil offre l'accesso ad una serie di operazioni di alto livello su file e raccolte di file. In particolare, vengono fornite funzioni che supportano la copia e la rimozione dei file.

- *subprocess*[7]: Il modulo subprocess consente di generare nuovi processi, collegarsi ai loro buffer di input/output/errore e ottenere i loro codici di ritorno. All'interno degli script viene utilizzato per sfruttare il comando subprocess.run() che permette di avviare un comando parallelo a quello in esecuzione dallo script.

- *csv*[8]: Il modulo csv implementa classi per leggere e scrivere dati tabulari in formato CSV. Consente agli script di utilizzare il comando write passando una lista di argomenti che verranno automaticamente salvati in formato csv.

- *requests*[9]: Il modulo requests consente di inviare richieste HTTP in modo estremamente semplice. Non è necessario aggiungere manualmente stringhe di query agli URL o codificare i dati POST. All'interno della tesi, questo modulo viene utilizzato per effettuare le richieste alle API di SonarQube per ottenere il file JSON con le metriche.

- *sys*[10]: Questo modulo fornisce l'accesso ad alcune variabili utilizzate o mantenute dall'interprete e a funzioni che interagiscono fortemente con l'interprete. È sempre disponibile. All'interno della tesi viene utilizzato per ottenere i parametri inviati tramite linea di comando.

- *urllib.request*[11]: Il modulo urllib.request definisce funzioni e classi che aiutano ad aprire gli URL (principalmente HTTP) in un mondo complesso: autenticazione di base e digest, reindirizzamenti, cookie e altro. All'interno della tesi, questo modulo viene utilizzato per effettuare le richieste alle API di GitHub e ottenere il JSON con i dettagli riguardanti le release presenti nel repository in analisi.

- *json*[12]: Il modulo JSON permette di manipolare facilmente i dati in formato JSON, utilizzato sia per GitHub che per SonarQube.

---

[6] https://docs.python.org/3/library/shutil.html

[7] https://docs.python.org/3/library/subprocess.html

[8] https://docs.python.org/3/library/csv.html

[9] https://docs.python.org/3/library/csv.html

[10] https://docs.python.org/3/library/sys.html

[11] https://docs.python.org/3/library/urllib.request.html

[12] https://docs.python.org/3/library/json.html

- *from requests.auth, HTTPBasicAuth*[13]: Il modulo HTTPBasicAuth permette di eseguire autenticazione HTTP basic, molti servizi web richiedono autenticazione accettando l'HTTP Basic Auth. E' la più semplice tipologia di autenticazione e Requests la supporta di default. All'interno della tesi viene utilizzato per autenticare la richiesta a SonarQube.

- *from time, sleep*[14]: Python ha un modulo chiamato time che fornisce diverse funzioni utili per gestire le attività relative al tempo. Una delle funzioni popolari tra loro è sleep(). La funzione sleep() sospende l'esecuzione del thread corrente per un determinato numero di secondi. All'interno della tesi viene utilizzato per attendere il calcolo delle metriche di fragilità del codice.

- *gitpython*[15]: GitPython è una libreria Python utilizzata per interagire con i repository git. Fornisce astrazioni di oggetti git per un facile accesso ai dati del repository e consente inoltre di accedere al repository git in modo più diretto utilizzando un'implementazione python pura o l'implementazione del comando git più veloce, ma più dispendioso in termini di risorse. L'implementazione del database degli oggetti è ottimizzata per la gestione di grandi quantità di oggetti e set di dati di grandi dimensioni, che si ottiene utilizzando strutture di basso livello e streaming di dati. Viene utilizzata nella tesi per eseguire i comandi di pull e diff delle varie versioni del progetto in analisi.

4.1.2 Calcolo metriche

In aggiunta alle metriche di fragmentation si è voluto aggiungere il calcolo delle metriche di qualità del codice, tramite l'adozione del tool SonarQube. Le metriche ottenute da SonarQube sono:

- *CC (McCabe's Cyclomatic Complexity)*[16]: è una metrica software sviluppata da Thomas J. McCabe nel 1976 ed è utilizzata per misurare la complessità di un programma. Misura direttamente il numero di cammini linearmente indipendenti attraverso il grafo di controllo di flusso. In Figura 4.2 si può comprendere come il calcolo della metrica CC è effettuato.

[13] https://docs.python-requests.org/en/latest/user/authentication/

[14] https://docs.python.org/3/library/time.html

[15] https://github.com/gitpython-developers/GitPython

[16] http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php

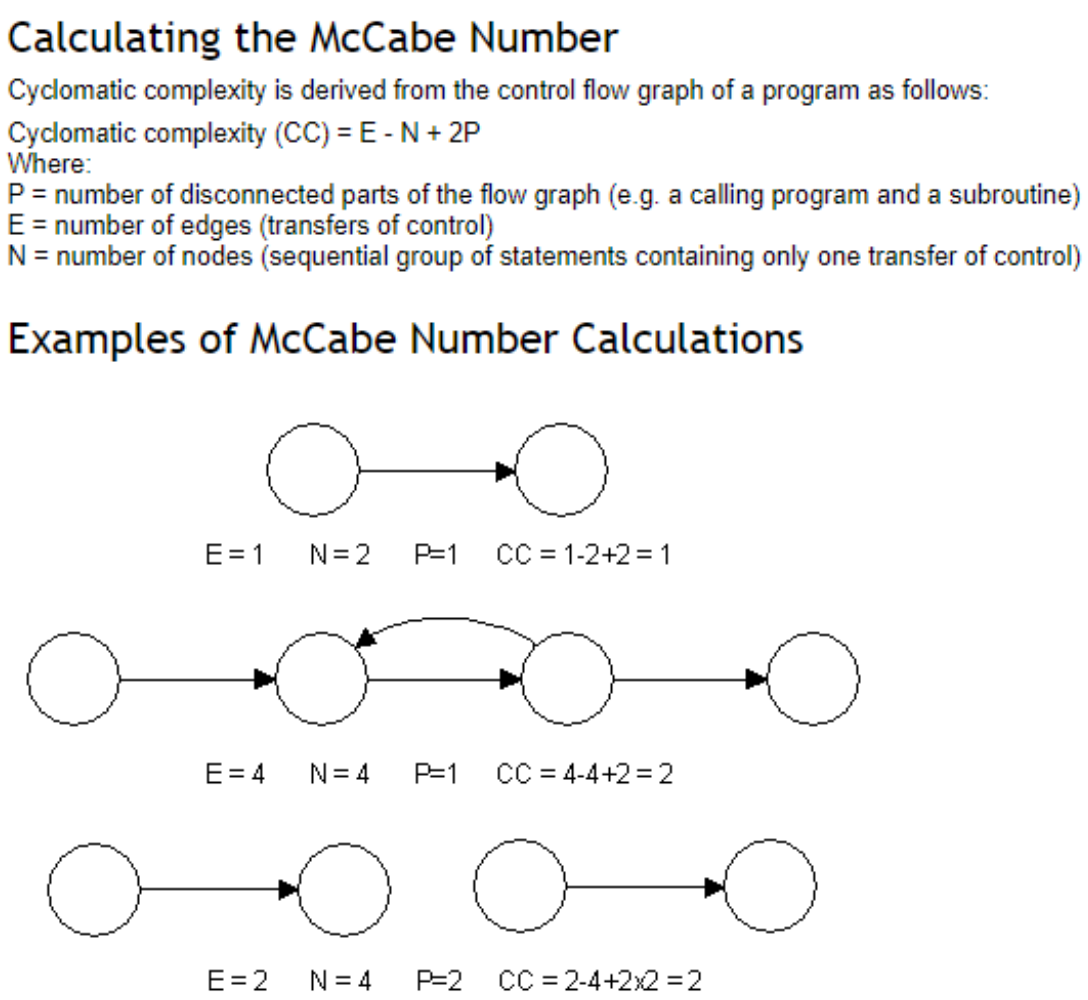


Immagine 4.2: Rappresentazione semplificata di un calcolo di CC.

- *TD (Technical Debt)*[17]: nello sviluppo del software, il debito tecnico (noto anche come debito di progettazione o debito di codice) è il costo implicito di una rilavorazione aggiuntiva causata dalla scelta di una soluzione semplice (limitata) al momento dello sviluppo invece di utilizzare un approccio migliore che richiederebbe più tempo. Analogamente al debito monetario, se il debito tecnico non viene rimborsato, può accumulare "interessi", rendendo più difficile l'attuazione dei cambiamenti. Il debito tecnico non risolto aumenta l'entropia del software e il costo di ulteriori rielaborazioni. Analogamente al debito monetario, il debito tecnico non è necessariamente negativo e talvolta (ad

[17] https://en.wikipedia.org/wiki/Technical_debt

esempio come prova di concetto) è necessario per portare avanti i progetti. D'altra parte, alcuni esperti affermano che la metafora del "debito tecnico" tende a minimizzare le ramificazioni, il che si traduce in una definizione insufficiente delle priorità del lavoro necessario per correggerlo. Quando viene avviata una modifica basata sul codice, spesso è necessario apportare altre modifiche coordinate in altre parti del codice o della documentazione. Le modifiche richieste che non vengono completate sono considerate debito e, fino al pagamento, comporteranno interessi, rendendo ingombrante la costruzione di un progetto.

- *TDR (TECHNICAL DEBT RATIO)*: rapporto tra il costo di riparazione e il costo di sviluppo. In genere nessuno vuole un Technical Debt Ratio elevato, alcuni team preferiscono valori inferiori o uguali al 5%. I punteggi TDR elevati indicano che il software ha una qualità scadente.

- *CODE SMELLS*[18]: l'espressione code smells viene usata per indicare una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione. I code smell non sono bug, cioè veri e propri errori, bensì debolezze di progettazione che riducono la qualità del software, a prescindere dall'effettiva correttezza del suo funzionamento. Il code smell spesso è correlato alla presenza di debito tecnico e la sua individuazione è un comune metodo euristico usato dai programmatori come guida per l'attività di refactoring, ovvero l'esecuzione di azioni di ristrutturazione del codice volte a migliorarne la struttura, abbassandone la complessità senza modificarne le funzionalità.

- *NOF (Number of Files)*: numero di file presenti nel progetto in analisi.

- *NOC (Number of Classes)*: numero di classi presenti nel progetto in analisi.

- *NOM (Number of Methods)*: numero di metodi presenti in tutte le classi facenti parte del progetto in analisi.

- *STAT (Number of Statements)*: numero di dichiarazioni presenti in tutte le classi presenti nel progetto in analisi.

- *LOC (Lines of Code)*: numero di linee di codice presenti in tutte le classi del progetto in analisi.

- *CLOC (Commented Lines Of Code)*: numero di linee di commenti presenti in tutte le classi del progetto in analisi. Questa metrica è inutile dal punto

[18] https://it.wikipedia.org/wiki/Code_smell

di vista della fragilità, ma può essere utile ad uno sviluppatore per capire l'evoluzione dei commenti in relazione al numero di classi piuttosto che metodi e/o dichiarazioni.

Le restanti metriche sono calcolate singolarmente dagli script python e memorizzate in un file CSV[19] cosí che le operazioni di scambio dati con il software Java siano semplificate e se l'utente volesse, potrebbe importare semplicemente il file in un qualsiasi foglio di calcolo per avere una rappresentazione semplice ed intuitiva delle metriche.

Le metriche calcolate dagli script (metriche di fragilità di test) sono le seguenti:

- *NTR (Number of Tagged Releases)*;
- *NTC (Number of Test Classes)*;
- *TTL (Total Test LOCs)*;
- *MTLR (Modified Test LOCs Ratio)*;
- *MRTL (Modified Relative Test LOCs)*;
- *MCR (Modified Test Classes Ratio)*;
- *MMR (Modified Test Methods Ratio)*;
- *MCMMR (Modified Classes with Modified Methods Ratio)*;
- Metriche riguardanti tutte le release del progetto in analisi:
  - *NTR (Number of Tagged Releases)*: numero di release presenti nel repository in analisi; serve per calcolare le metriche successive.
  - *MRR (Rapporto Numero Modifiche)*: indica il rapporto tra le release con delle effettive modifiche di codice di test con il numero totale di release del progetto.
  - *TSV (Test Suite Volatility)*: rapporto del numero di classi di test modificate con il numero totale di classi di test nel corso di tutte le release.

[19]I file con estensione .csv (Comma Separated Values) rappresentano file di testo normali che contengono record di dati con valori separati da virgole. Ogni riga in un file CSV è un nuovo record. Tali file vengono generati quando si intende trasferire i dati da un sistema di archiviazione a un altro. Poiché tutte le applicazioni possono riconoscere record separati da virgole, l'importazione di tali file di dati nel database viene eseguita in modo molto conveniente. Quasi tutte le applicazioni per fogli di calcolo come Microsoft Excel o OpenOffice Calc possono importare CSV senza troppi sforzi. I dati importati da tali file sono organizzati in celle di un foglio di calcolo per la rappresentazione all'utente. Fonte:https://docs.fileformat.com/spreadsheet/csv/

4.1.3 File generati dagli script

La memorizzazione delle metriche calcolate dagli script avviene mediante la generazione di un file CSV. Durante l'avvio dello script, quest'ultimo controllerà i file e le cartelle presenti nella propria directory ed eseguirà la pulizia di eventuali file residui da esecuzioni passate (a causa di arresti improvvisi o crash del sistema). Una volta eseguita la pulizia, verranno generati due nuovi file csv:

- un primo file codificato come <nome progetto sonarqube>_<nome repository>.csv[20] permette di mantenere una lista di file CSV riguardanti tutti i progetti analizzati dagli script (vengono rimossi eventuali caratteri speciali come ad esempio "/" per non rischiare di generare incompatibilità con gli standard POSIX in merito alla nomina dei file[21]. Questo file conterrà una riga per ogni release, in ogni riga saranno presenti tutte le metriche separate secondo gli standard CSV. Nell'Immagine 4.3 si trova un esempio di primo file CSV generato.

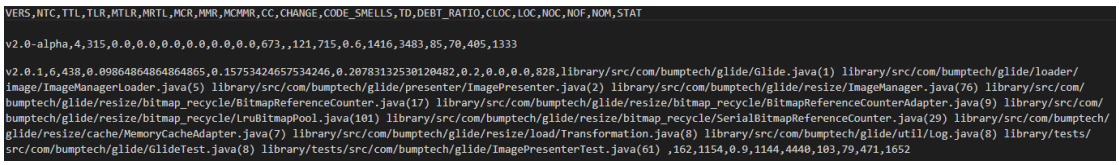


Immagine 4.3: Esempio di primo file CSV generato sulle prime due versioni presenti sul repository Bumptech/Glide.

- un secondo file codificato come <nome progetto sonarqube>_<nome repository>_2.csv[22] permette di mantenere una lista di file CSV riguardanti tutti i progetti analizzati dagli script (vengono rimossi eventuali caratteri speciali come ad esempio "/" per non rischiare di generare incompatibilità con gli standard POSIX in merito alla nomina dei file[23]. Questo file conterrà una riga unica oltre a quella di intestazione, nella riga saranno presenti le tre metriche

[20]Nel caso della mia tesi, il progetto SonarQube si chiama ScannerPythonTesi. Supponendo quindi di richiedere l'analisi di un repository chamato s276000/tesi, il file generato sarà ScannerPythonTesi_s276000tesi.csv

[21]https://www.ibm.com/docs/en/zos/2.3.0?topic=locales-posix-portable-file-name-character-set

[22]Nel caso della mia tesi, il progetto SonarQube si chiama ScannerPythonTesi. Supponendo quindi di richiedere l'analisi di un repository chamato s276000/tesi, il file generato sarà ScannerPythonTesi_s276000tesi_2.csv

[23]https://www.ibm.com/docs/en/zos/2.3.0?topic=locales-posix-portable-file-name-character-set

riguardanti tutte le release presenti (NTR, MRR, TSV) separate secondo gli standard CSV. Nell'Immagine 4.4 si trova un esempio di secondo file CSV generato.

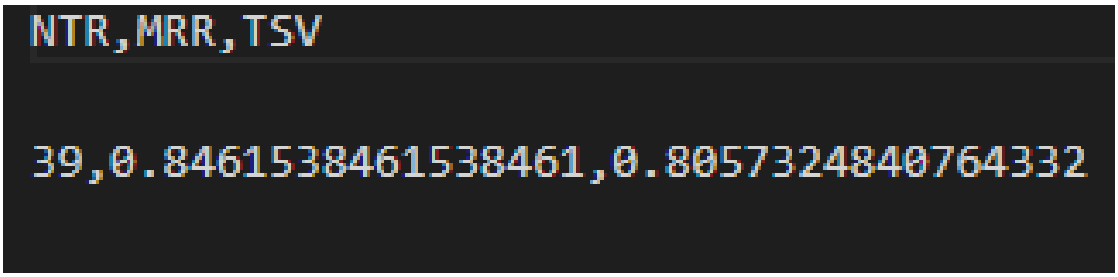


Immagine 4.4: Esempio di secondo file CSV generato riguardante il repository Bumptech/Glide.

4.2 Sviluppo programma Java

Per lo sviluppo della componente grafica della tesi si è deciso di utilizzare il linguaggio di programmazione ad oggetti Java, così da risolvere il più possibile i problemi di compatibilità con hardware e software differente.

4.2.1 Organizzazione Classi e Packages

Per la divisione in packages e per la divisione in classi si è optato per la seguente gerarchia:

Immagine 4.5: Diagramma delle dipendenze all'interno del package data

- *data* contiene tutte le classi che gestiscono il caricamento e l'elaborazione dei dati. Il diagramma seguente rappresenta le interazioni delle tre classi fondamentali, che gestiscono il caricamento delle metriche dai file CSV prodotti dagli script python, come la classe Constants sia utilizzata dalla classe che contiene le metriche di tutte le versioni del software e la classe Statistics che sarà utilizzata per la visualizzazione dei dati della tabella delle statistiche.

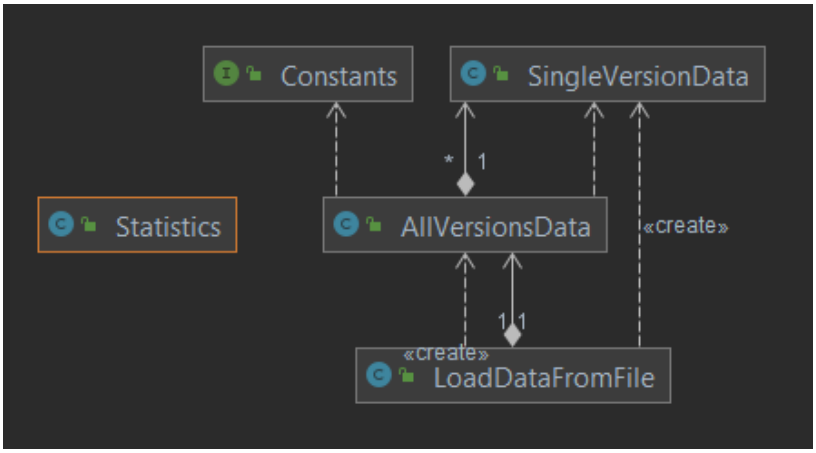


Immagine 4.6: Diagramma delle dipendenze all'interno del package data

- *gui* contiene tutte le classi che sono addette alla generazione di una interfaccia grafica.

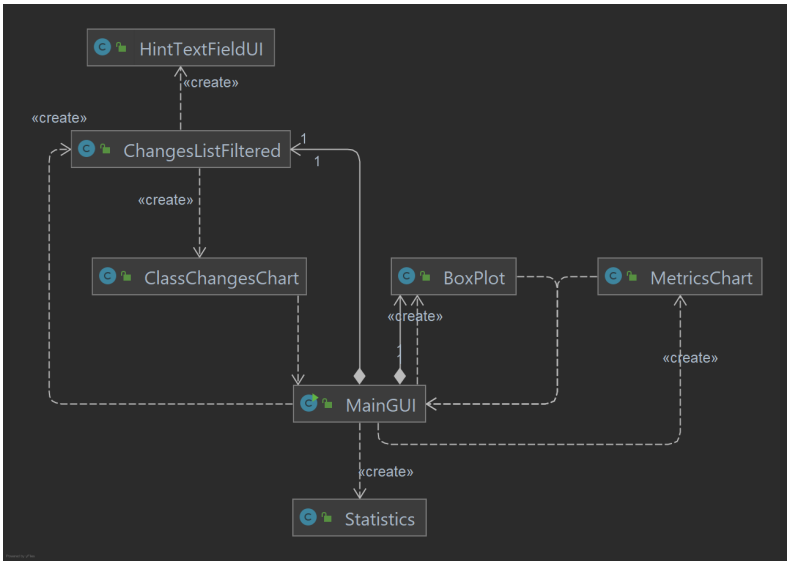


Immagine 4.7: Diagramma che rappresenta come sono collegate tra di loro le classi che hanno il compito di generare una GUI.

- *main* contiene semplicemente la classe Main, che lancia l'applicazione MainGUI.

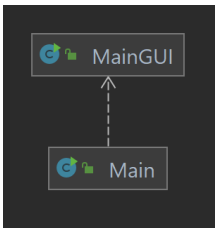


Immagine 4.8: Diagramma che rappresenta come sono collegate tra di loro le classi Main e MainGUI

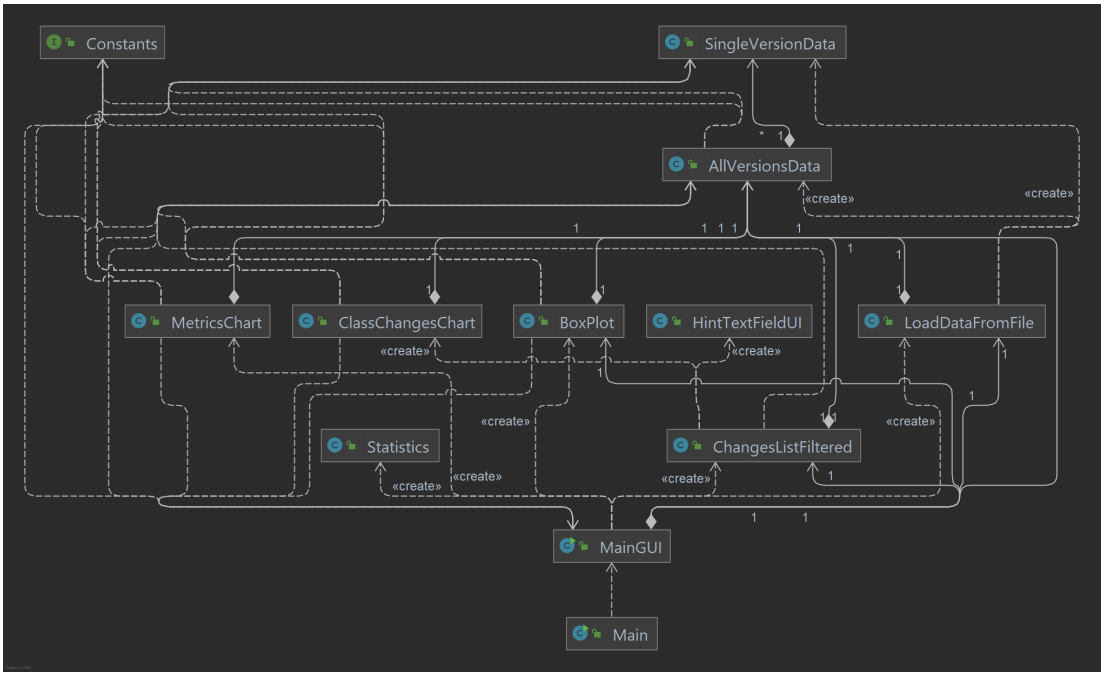


Immagine 4.9: Diagramma che rappresenta come sono collegate tra di loro tutte le classi che compongono il programma.

4.2.2 Librerie Java utilizzate

In primo luogo si é ricercata una GUI, che fosse il più possibile user-friendly[24]: si é dunque deciso di basarla sul framework open source JavaFX[25]. Si é poi cercato di rappresentare graficamente la sequenza di dati, desiderata dall'utente, permettendogli di ottenere tutte le informazioni desiderate nel minor tempo possibile: per questo si é optato per la libreria JFreeChart[26]. Con l'utilizzo di questa libreria, il codice permette di generare una combinazione di grafici a linee e a punti in base ai dati desiderati dall'utente. Una volta ottenuto il grafico, l'utente, semplicemente eseguendo un mouse hover[27] sui punti del grafico, otterrà un tooltip che permetterà di visualizzare il valore del dato alla release indicata; altrettanto semplicemente, all'utente basterà selezionare un'area del grafico per ingrandire quest'ultima. Ciò

[24] https://dictionary.cambridge.org/it/dizionario/inglese/user-friendly

[25] JavaFX è una piattaforma applicativa client open source di nuova generazione per sistemi desktop, mobile ed embedded basati su Java. https://openjfx.io/

[26] https://www.jfree.org/jfreechart/

[27] utilizzare il mouse di un computer per spostare il cursore su una parte particolare dello schermo

risulta particolarmente utile per analizzare le metriche, che si discostano poco tra di loro.

Una seconda rappresentazione dei dati è realizzata mediante l'utilizzo di un BoxPlot. Il termine "box plot" si riferisce a un box plot degli outlier, chiamato anche diagramma a scatola e baffi o box plot di Tukey. Gli elementi fondamentali di un box plot[28] sono:

- La linea centrale nella scatola rappresenta la mediana dei dati. Se i dati sono simmetrici, la mediana è al centro della scatola. Se, invece, i dati sono asimmetrici, la mediana sarà più vicina alla parte superiore o a quella inferiore della scatola.
- La parte inferiore e superiore della scatola mostrano il 25° e il 75° quantile, o percentile. Questi due quantili sono chiamati anche quartili, poiché ciascuno di essi esclude un quarto (25 %) dei dati. La lunghezza della scatola è la differenza tra i due percentili e si chiama range interquartile (IQR).
- Le linee che si estendono a partire dalla scatola sono chiamate baffi. I baffi rappresentano la variazione dei dati attesa e si estendono per 1,5 volte dall'IQR sia dalla parte superiore che inferiore della scatola. Se i dati non arrivano fino alla fine dei baffi, significa che i baffi si estendono fino ai valori di dati minimi e massimi. Se, invece, i dati ricadono sopra o sotto la fine dei baffi, sono rappresentati come punti, denominati spesso outlier. Un outlier è più estremo della variazione attesa. Vale la pena esaminare questi punti di dati per determinare se sono errori o outlier.

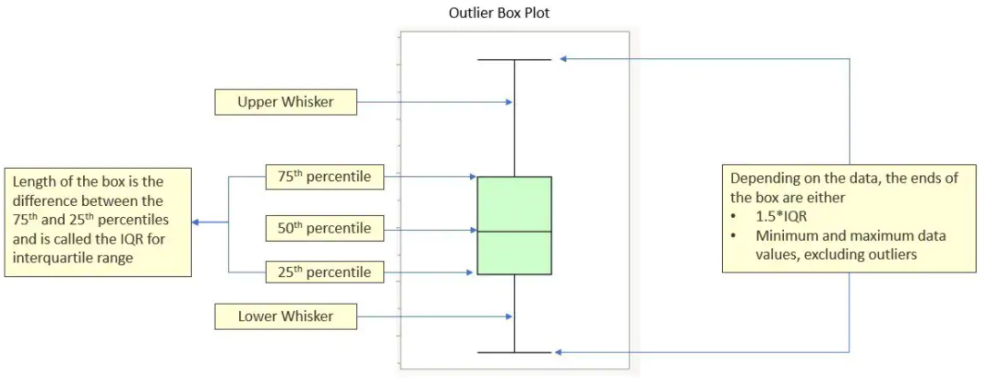


Immagine 4.10: BoxPlot Generico

[28] https://www.jmp.com/it_it/statistics-knowledge-portal/exploratory-data-analysis/box-plot.html

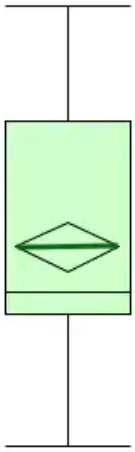


Immagine 4.11: Box plot con una linea che indica la differenza tra media statistica e mediana; la media statistica è rappresentata dalla linea nel rombo, la mediana invece dalla linea sottile sottostante

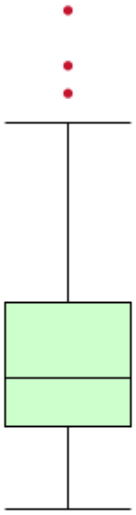


Immagine 4.12: Box plot con punti che rappresentano gli outlier

4.2.3 Classi GUI

Si analizza ora come sono state costruite le interfacce grafiche che appaiono a schermo in ogni sezione dell'applicazione (per gli screenshot ho utilizzato il repository

bumptechglide in quanto il progetto presente in questa repository si presta molto bene per l'analisi delle metriche):

- *MainGUI*: classe principale che gestisce tutta l'esecuzione, dall'avvio degli script alla visualizzazione dei grafici. L'interfaccia è divisa in TabPane[29], ognuno per la propria sezione:

    1. *Selezione Input*: quando questo Tab è attivo, tutti gli altri sono disattivati in quanto inutili da visitare. All'avvio dell'applicazione, questa sarà l'unica interfaccia disponibile: sono presenti i campi per inserire il nome della repository da analizzare, eventualmente il numero di ultime release da analizzare e il nome dello strumento di testing per il quale vogliamo calcolare le metriche. È presente anche un pulsante che permette di caricare direttamente i dati selezionando un file csv dal proprio sistema attraverso un File Chooser che comparirà una volta premuto il pulsante apposito.

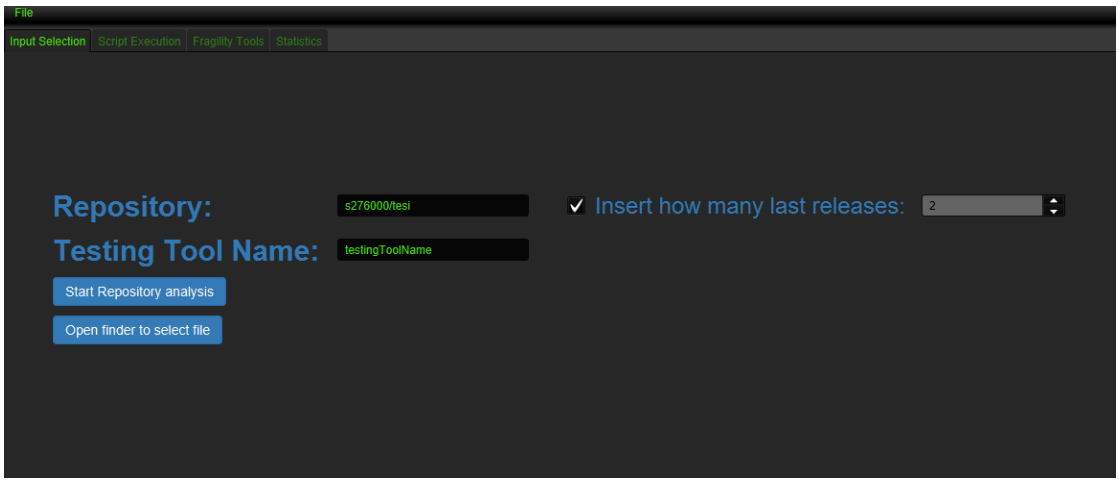


Immagine 4.13: Rappresentazione del Tab Selezione input

    2. *Esecuzione Script*: interfaccia che appare automaticamente una volta avviati gli script per il calcolo delle metriche. Questa interfaccia si aggiorna in tempo reale con le versioni attualmente sotto analisi dagli script così da fornire all'utente un feedback sul procedimento di analisi (se si analizzano repository con qualche centinaia di righe di codice e/o test, l'analisi di una versione può richiedere anche una decina di minuti, dipendenti anche dalla potenza di calcolo disponibile). Uno spinner inoltre

[29] https://docs.oracle.com/javase/tutorial/uiswing/components/tabbedpane.html

ha un'animazione sincronizzata con il thread che gestisce lo script python, così che l'utente possa capire se qualcosa di anomalo è accaduto. In caso di eccezioni e/o terminazioni eccezionali degli script, il programma reimposta tutte le strutture dati e il focus ritorna nel tab di selezione input.

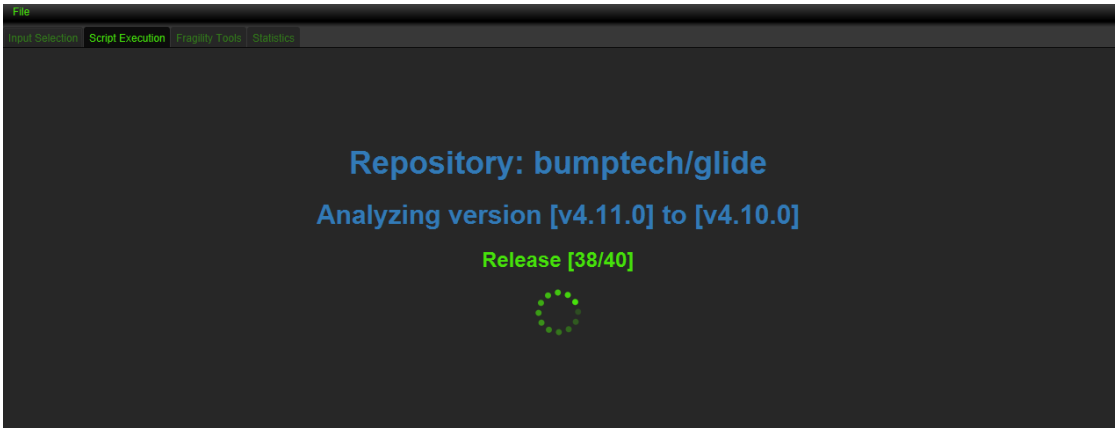


Immagine 4.14: Rappresentazione del Tab Esecuzione script

3. *Strumento Analisi Fragilità*: Tab che si apre automaticamente una volta terminata l'esecuzione degli script. Da questa interfaccia é possibile generare tutti i tipi di grafici. Come titolo viene visualizzato il nome del repository, in caso si sia eseguita la scansione con gli script, oppure il nome del file csv selezionato tramite il file chooser; nella sezione sottostante, è presente l'elenco delle tre metriche riguardanti l'intero progetto (NTR, MRR e TSV). Nella parte principale della GUI sono presenti le metriche suddivise in colonne per facilitare la ricerca: una volta selezionate quelle desiderate, é possibile generare il grafico che le includerà tutte[30] semplicemente con il pulsante dedicato.

[30]Attenzione: il grafico imposta automaticamente i limiti degli assi per adattarsi alla distribuzione di metriche con range più elevati; se si decide di rappresentare una metrica intera assieme ad una ratio, generalmente quella ratio verrà condensata in una riga orizzontale che tende a coincidere con l'asse delle ascisse.

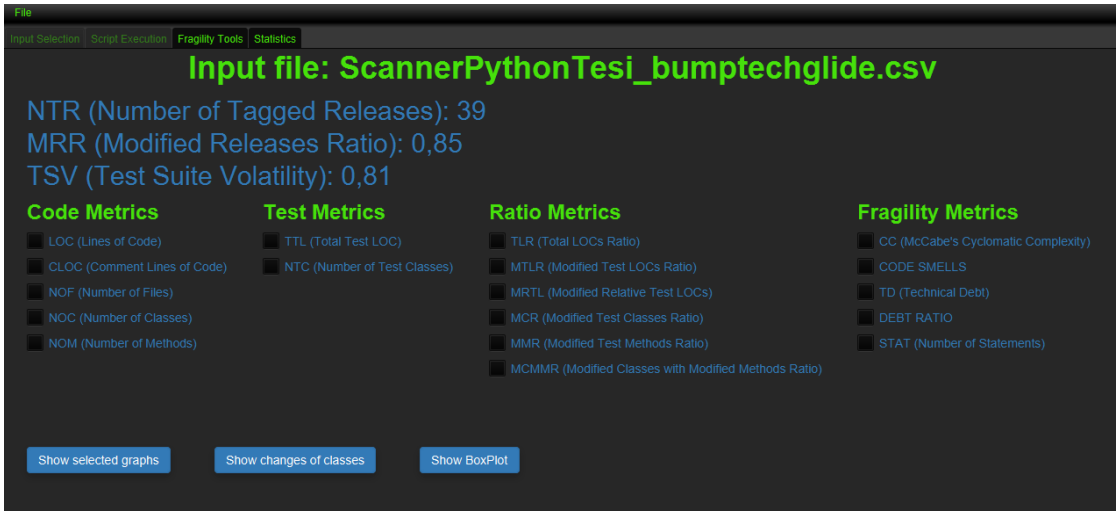


Immagine 4.15: Rappresentazione del Tab Strumento Analisi Fragilità

E' possibile generare più grafici in contemporanea, così da permettere una rappresentazione di metriche intere con metriche ratio e poterle confrontare su grafici diversi, a parità di release.

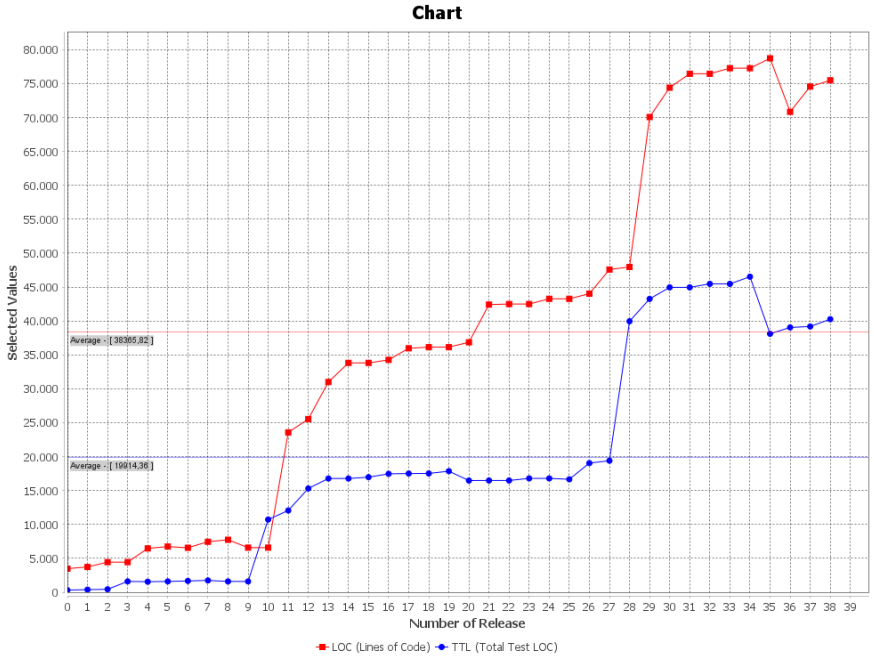


Immagine 4.16: Rappresentazione di un grafico generato su metriche intere

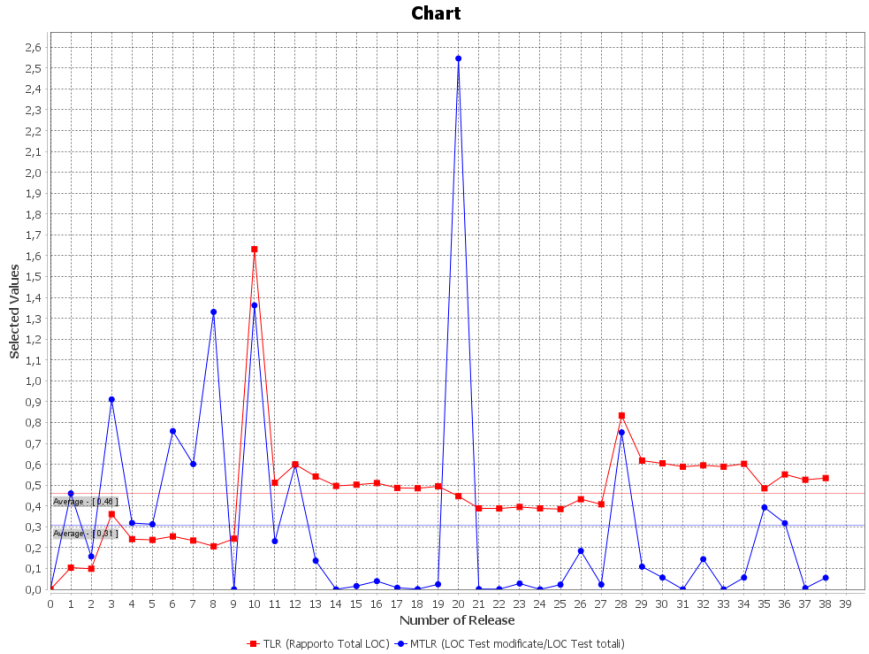


Immagine 4.17: Rappresentazione di un grafico generato su metriche ratio

Il secondo pulsante "Visualizza i cambiamenti di ogni classe" genera una GUI con un campo di ricerca per filtrare l'elenco delle classi che hanno subito dei cambiamenti in almeno una release del progetto ordinate per numero di modifiche decrescente. Selezionata una classe, viene generato un grafico rappresentante il numero di linee di codice modificate in ogni release.

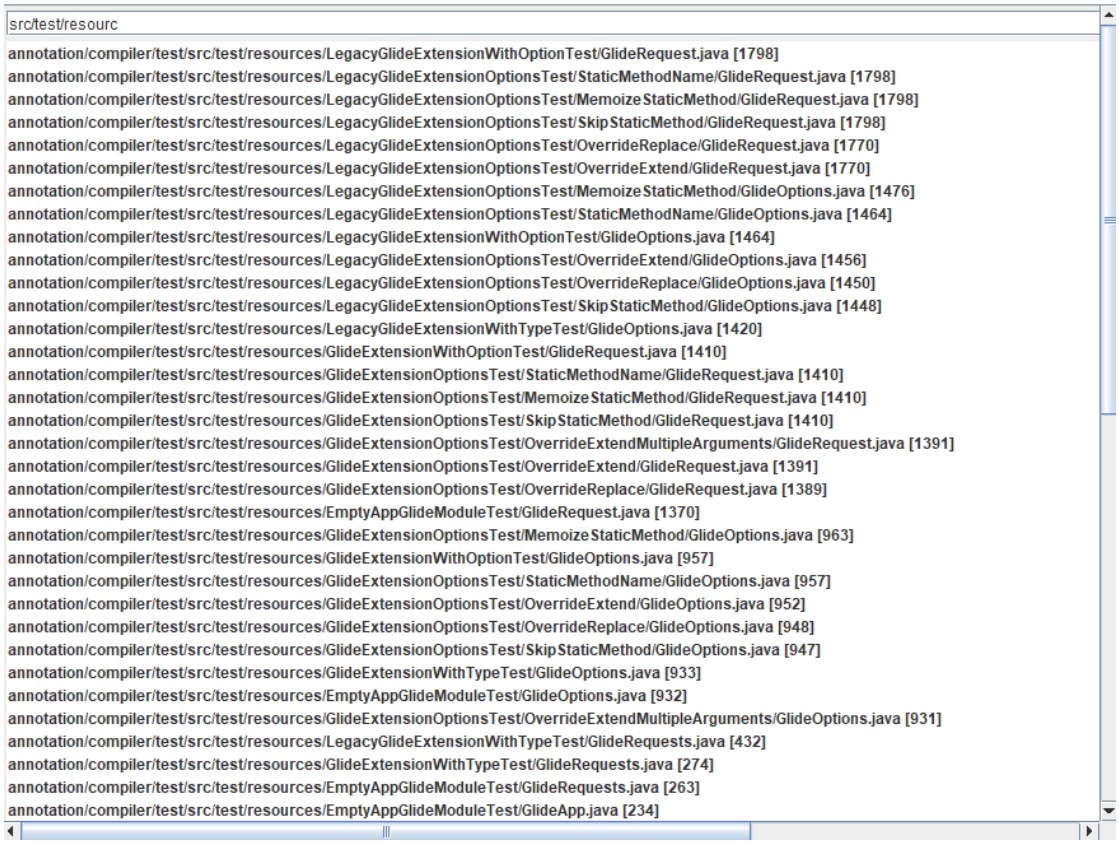


Immagine 4.18: Rappresentazione della GUI Cambiamenti delle classi

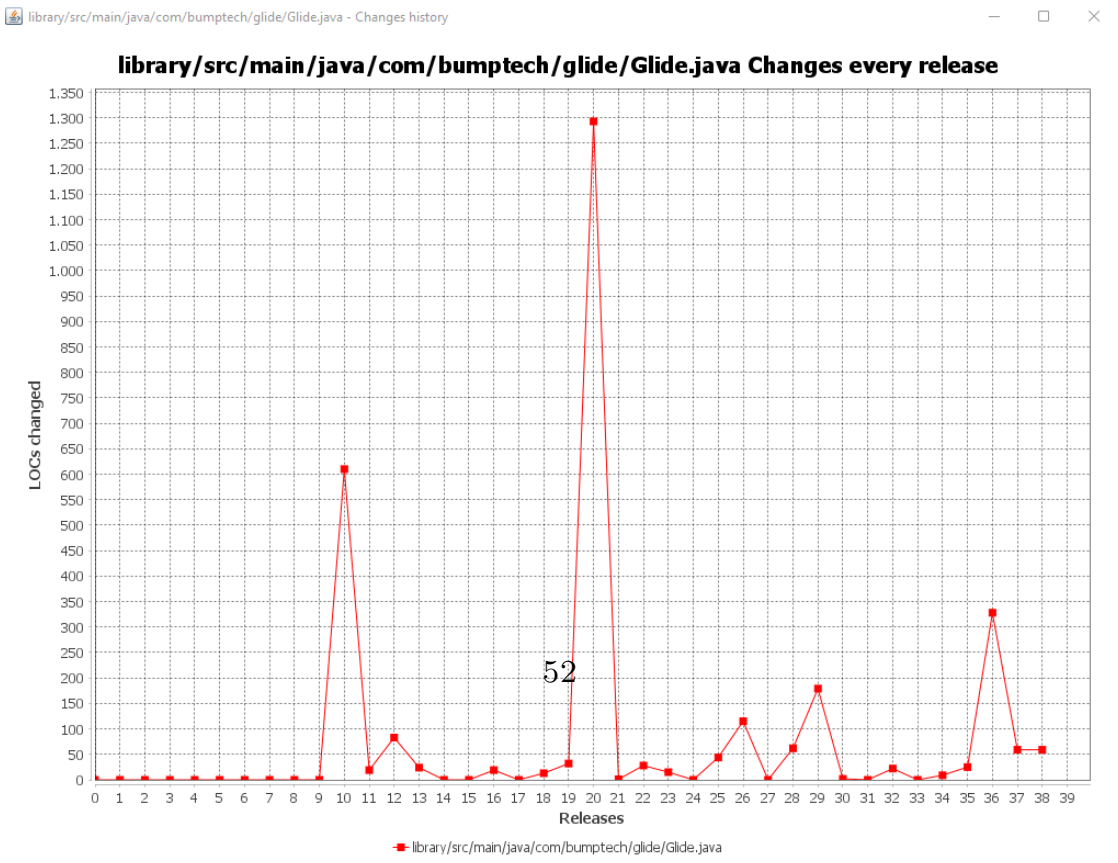


Immagine 4.19: Rappresentazione del grafico dei cambiamenti della classe selezionata

Infine il pulsante BoxPlot, apre una schermata con il boxplot delle metriche ratio; anche su questo è possibile effettuare uno zoom, con un mouse hover verranno visualizzate tutte le proprietà della distribuzione statistica in dettaglio.

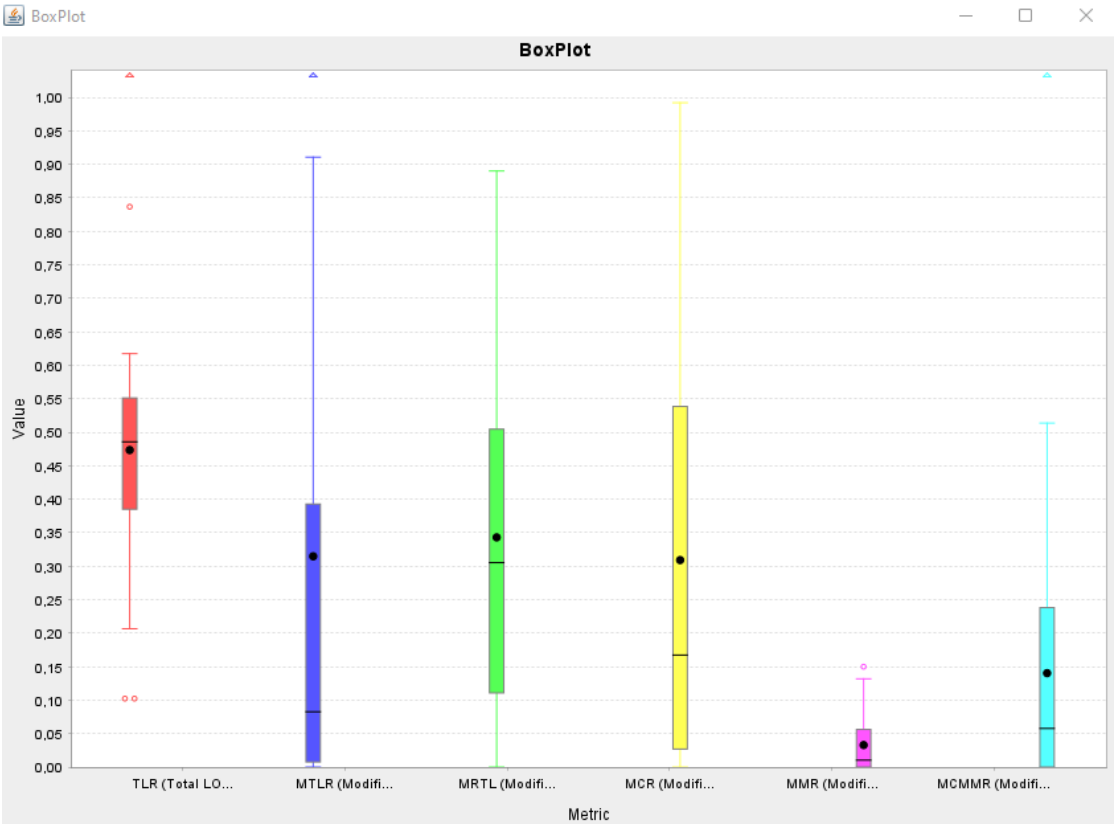


Immagine 4.20: Rappresentazione del BoxPlot delle metriche ratio

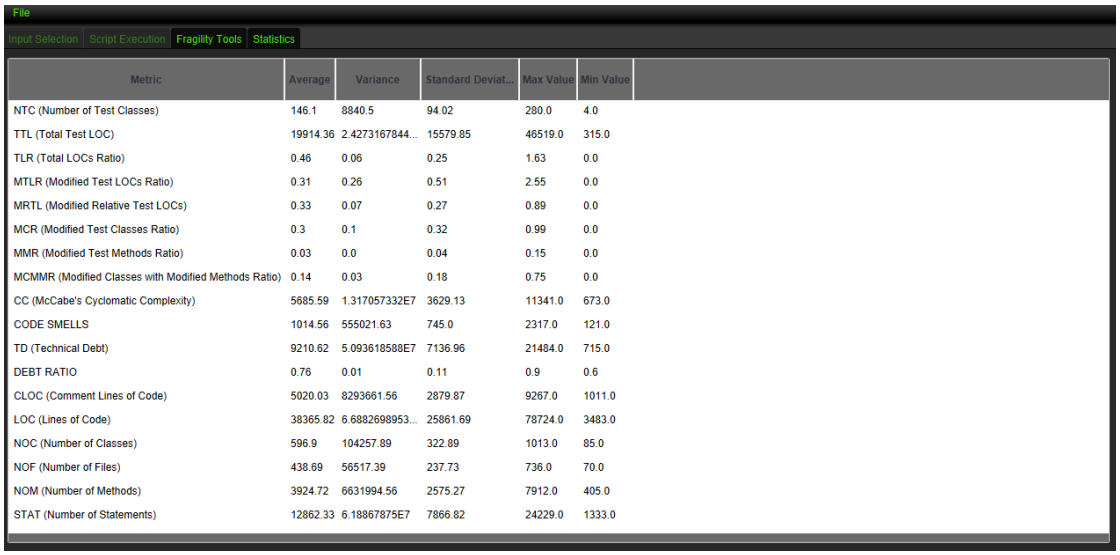
4. *Statistiche*: ultima interfaccia disponibile, inserita per permettere una rapida valutazione di tutte le metriche. Essa contiene una tabella con tutte le metriche, per ognuna delle quali sono riportate media aritmetica[31], varianza[32], deviazione standard[33], valore massimo e valore minimo. È

[31]somma di tutti i valori assunti dalla metrica, divisi per il numero totale di release

[32]misura della variabilità dei valori assunti dalla variabile stessa; nello specifico, la misura di quanto essi si discostano quadraticamente rispettivamente dalla media aritmetica

[33]chiamato anche scarto quadratico medio, è un indice di dispersione statistico, vale a dire una stima della variabilità di una popolazione di dati o di una variabile casuale. È uno dei modi per esprimere la dispersione dei dati intorno ad un indice di posizione, come può essere, ad esempio,

possibile ordinare i dati in base a una colonna a scelta semplicemente con un click sull'intestazione della colonna desiderata.



Metric	Average	Variance	Standard Deviat...	Max Value	Min Value
NTC (Number of Test Classes)	146.1	8840.5	94.02	280.0	4.0
TTL (Total Test LOC)	19914.36	2.4273167844...	15579.85	46519.0	315.0
TLR (Total LOCs Ratio)	0.46	0.06	0.25	1.63	0.0
MTLR (Modified Test LOCs Ratio)	0.31	0.26	0.51	2.55	0.0
MRTL (Modified Relative Test LOCs)	0.33	0.07	0.27	0.89	0.0
MCR (Modified Test Classes Ratio)	0.3	0.1	0.32	0.99	0.0
MMR (Modified Test Methods Ratio)	0.03	0.0	0.04	0.15	0.0
MCMMR (Modified Classes with Modified Methods Ratio)	0.14	0.03	0.18	0.75	0.0
CC (McCabe's Cyclomatic Complexity)	5685.59	1.317057332E7	3629.13	11341.0	673.0
CODE SMELLS	1014.56	555021.63	745.0	2317.0	121.0
TD (Technical Debt)	9210.62	5.093618588E7	7136.96	21484.0	715.0
DEBT RATIO	0.76	0.01	0.11	0.9	0.6
CLOC (Comment Lines of Code)	5020.03	8293661.56	2879.87	9267.0	1011.0
LOC (Lines of Code)	38365.82	6.6882698953...	25861.69	78724.0	3483.0
NOC (Number of Classes)	596.9	104257.89	322.89	1013.0	85.0
NOF (Number of Files)	438.69	56517.39	237.73	736.0	70.0
NOM (Number of Methods)	3924.72	6631994.56	2575.27	7912.0	405.0
STAT (Number of Statements)	12862.33	6.18867875E7	7866.82	24229.0	1333.0

Immagine 4.21: Rappresentazione del Tab Statistiche

4.2.4 Classi Dati

Le classi riguardanti i dati si occupano principalmente di leggere, memorizzare ed elaborare i valori delle metriche presenti nei file CSV generati dagli script python. Queste si dividono in cinque file:

1. *SingleVersionData*: il compito di questa classe è quello di fornire un supporto per accedere facilmente ai valori delle metriche che sono presenti nei file CSV generati dagli script python. I dati vengono organizzati in campi privati omonimi alla metrica che devono memorizzare, sono presenti i *getters&setters* con i quali le classi che vedremo in seguito accederanno ai campi desiderati. Il campo change[34] viene memorizzato in una *Map<String, Integer>* così da facilitare le operazioni necessarie che vedremo in seguito di somma e ordinamento dei cambiamenti subiti da ogni classe.

la media aritmetica. Ha pertanto la stessa unità di misura dei valori osservati (al contrario della varianza avente come unità di misura il quadrato dell'unità di misura dei valori di riferimento).

[34]gli script python generano un campo change che altro non è che l'elenco dei path relativi di ogni classe del progetto seguito dal numero di righe modificate

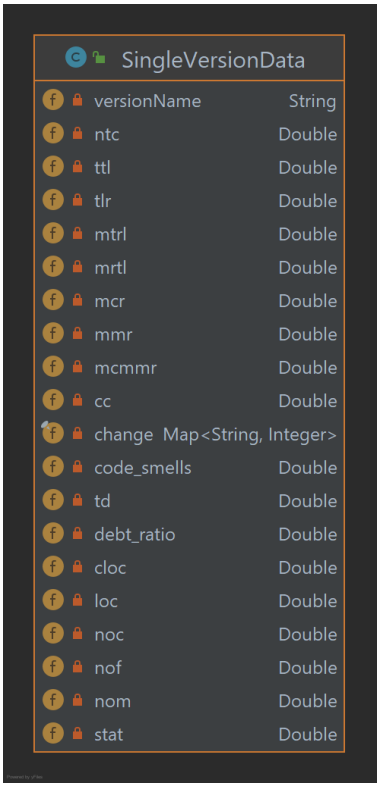


Immagine 4.22: Diagramma UML della classe SingleVersionData(sono stati omessi i metodi in quanto solo *getters&setters)*

2. *AllVersionsData*: l'obiettivo di questa classe è quello di essere una copia speculare dei file generati dagli script python, avendo funzione di *wrapper*[35] nei confronti delle metriche di ogni versione del codice in analisi; esso inoltre gestisce il calcolo delle statistiche su queste ultime. Analizziamo in dettaglio la funzione di ogni campo e metodo della classe:

   - *singleVersionDataArray*: questo campo è definito come un *ArrayList <SingleVersionData>*, contiene per ogni elemento del vettore, un "record" contenuto nel file CSV.
   - *classChanges*: analogamente alla definizione riportata nella classe SingleVersionData, questa struttura contiene la somma totale di tutti i

[35] *Un wrapper (dal verbo inglese to wrap, "avvolgere"), in informatica, e in particolare in programmazione, è un modulo software che ne "riveste" un altro, ossia che funziona da tramite fra i propri clienti (che usano l'interfaccia del wrapper) e il modulo rivestito (che svolge effettivamente i servizi richiesti, su delega dell'oggetto wrapper).*

cambiamenti avvenuti in ogni classe (identificata con il path relativo utilizzato come chiave della mappa) in ogni release, così da avere un elenco globale dei cambiamenti. Questo serve per la creazione della lista dei cambiamenti di ogni classe presente nella sezione "Cambiamenti delle classi" precedentemente illustrata.

- *completeStatistics*: struttura dati composta da due mappe annidate. La mappa esterna ha come chiave una stringa che indica il nome della statistica che si ricerca[36] e come *value* una mappa che ha come chiave l'indice della metrica[37] e come *value* l'effettivo valore della statistica della metrica alla quale corrisponde.
- *NTR, MRR, TSV*: campi che contengono le omonime metriche lette dal file CSV.
- *repoName*: stringa che conterrà il nome della repository in analisi.
- *calculateAverage*: metodo che, tramite un ciclo, permette di calcolare la media aritmetica di ogni metrica.
- *calculateMax*: metodo che, tramite un ciclo, permette di ottenere il valore massimo di ogni metrica.
- *calculateMin*: metodo che, tramite un ciclo, permette di ottenere il valore minimo di ogni metrica.
- *calculateVariance*: metodo che, tramite un ciclo, permette di calcolare la varianza di ogni metrica, vien utilizzata la formula della differenza degli scarti dalla media al quadrato in quanto risultava essere la più immediata da calcolare.
- *calculateStandardDeviation*: metodo che, ricevuta in input la mappa contenente le varianze precedentemente calcolate, ne esegue la radice quadrata (e quindi lo scarto quadratico medio/deviazione standard), calcolando la deviazione standard.
- *initStatisticMap*: metodo che ha l'unico compito di creare una nuova Map ed inizializzare ogni *value* corrispondente ad ogni metrica a 0[38].
- *initMinStatisticMap*: metodo che ha l'unico compito di creare una nuova Map ed inizializzare ogni *value* corrispondente ad ogni metrica al valore standard *Double.MAX_VALUE*.

[36] queste sono state memorizzate nella classe Constants così da non creare incongruenze

[37] anche in questo caso le metriche sono state codificate sotto forma di interi tra 1 e 19, memorizzate nella classe Constants per non creare incongruenze

[38] l'inizializzazione a 0 funziona per tutte le metriche tranne per la metrica min, in quanto il valore nullo, in quest'ultima, andrebbe a falsificare di fatto tutti i valori.

- *getters&setters*: funzionamento di *getters&setters* secondo standard Java.

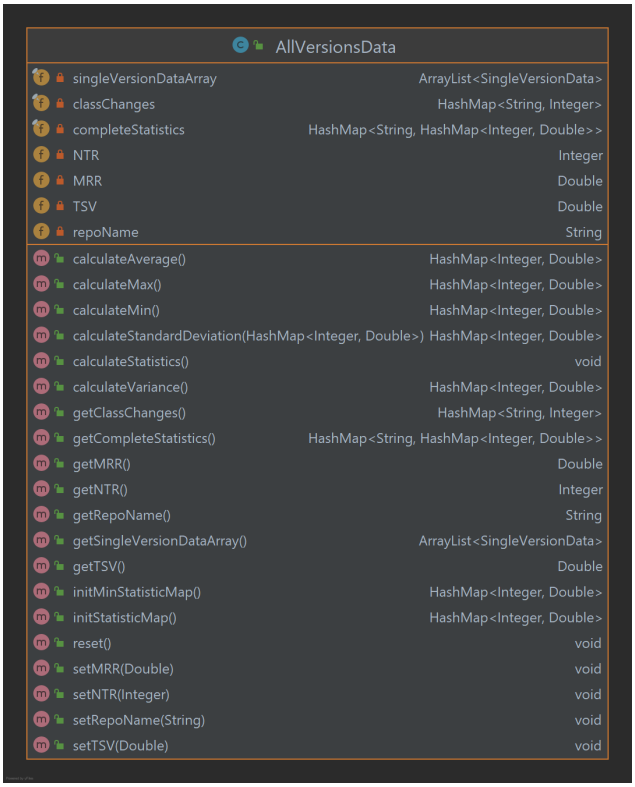


Immagine 4.23: Diagramma UML della classe AllVersionsData

3. *LoadDataFromFile*: questa classe gestisce il caricamento e la memorizzazione effettiva delle metriche nelle strutture adatte. Attraverso il metodo *load(String)* viene espressamente passato il nome del file dal quale i dati andranno caricati. Le metriche vengono caricate attraverso un classico ciclo su file, per ogni riga presente nel file CSV (ad eccezione della prima di intestazione) viene creata una istanza della classe SingleVersionData. Tutte le istanze di SingleVersionData vengono successivamente inserite in un oggetto AllVersionData che verrà ritornato al chiamante.

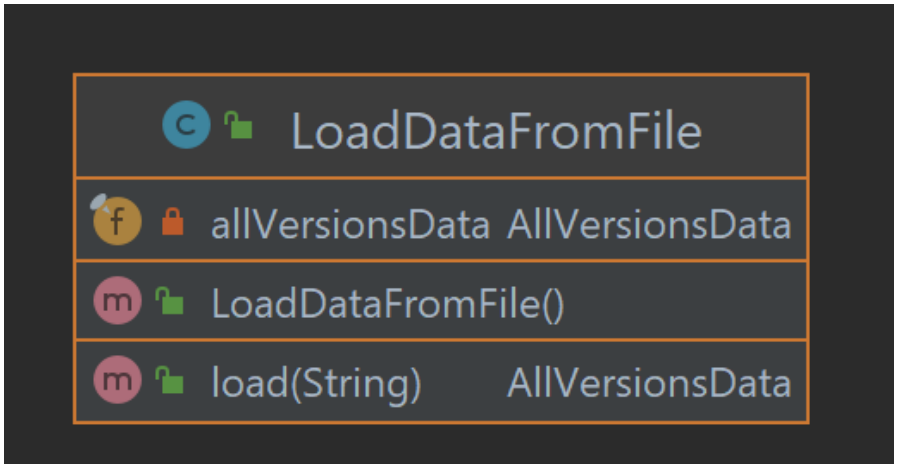


Immagine 4.24: Diagramma UML della classe LoadDataFromFile

4.2.5 Esempi di Progetti analizzati

In questa sezione si vuole riportare una dimostrazione dell'utilizzo del tool scegliendo due progetti open-source specifici, selezionati per determinate caratteristiche. Il primo (Bumptech/Glide) presenta sia un rapporto costante di linee di test rispetto alle linee di codice durante tutto lo sviluppo sia un BoxPlot molto significativo e facilmente interpretabile. Il secondo, invece, contiene meno release e quindi i grafici risultano più comprensibili.

Bumptech/Glide

Glide[39] è un framework open source veloce ed efficiente per la gestione dei media e il caricamento delle immagini per Android che racchiude la decodifica dei media, la memorizzazione nella cache della memoria e del disco e il pool di risorse in un'interfaccia semplice e facile da usare. Vista la qualità dei dati relativi alla fragilità del loro codice, si è deciso di utilizzarlo come esempio principale.

Si analizzano ora alcuni grafici relativi al codice presente sul repository.

- In Figura 4.25 sono riportati i grafici delle metriche *LOC, CLOC* e *TTL*, molto utili per avere una rapida impressione di come il progetto si è evoluto durante le release che lo compongono. Si osserva una crescita continua sia delle *LOC* sia delle *TTL*, questo sta ad indicare che lo sviluppo dei test case ha progredito durante tutte le versioni del software in analisi.

[39]https://github.com/bumptech/glide

- In figura 4.26, sono riportati i grafici delle metriche *TLR* e *MTLR*, da queste si osserva come i trend siano generalmente collegati tra di loro. Un incremento della metrica *MTLR*, come si può notare nella release 20, indica che la suite di test è stata ricostruita completamente, nonostante dalla Figure 4.25 non si noti un incremento notevole delle *TTL*, queste sono state modificate completamente.

- In figura 4.27, sono riportati i grafici delle metriche *NOF* e *NOC*. Permette di intuire mediamente quante classi sono presenti per file all'interno del progetto.

- In figura 4.28, sono riportati i grafici delle metriche *NOC* e *NTC*, i quali portano l'attenzione principalmente sullo sviluppo di codice di test parallelamente a quello del codice standard del progetto.

- In figura 4.29, sono riportati i grafici delle metriche *TD, LOC* e *TTL*. Possiamo osservare con interesse come il TD cresca simultaneamente al numero di LOC presenti, questo suggerisce un incremento dei "problemi" tralasciati dai programmatori durante gli aggiornamenti del software.

- In figura 4.30, sono riportati i grafici delle metriche *CC* e *TD*. Si evince la dipendenza che lega le due metriche (ad un aumento della complessità *CC* corrisponde un aumento del *TD*).

- Nella figura 4.31, sono riportati i grafici delle metriche *MCR* e *MTLR*. Questo grafico permette di percepire quanto le modifiche di metodi di test abbiano effetto sulla percentuale di classi di test modificate.

- In figura 4.32, è riportato il BoxPlot rappresentante le principali metriche ratio presenti nel tool. Si può notare la presenza di outliers in *TLR, MTLR, MMR* e *MCMMR*. Dal grafico rappresentato in Figura 4.26 possiamo evincere che il principale outlier della metrica MTLR sia il valore prossimo a 2.6 mentre della metrica *TLR*, siano 1.7 (escluso dal boxplot ma rappresentato con un triangolo verso l'alto) e 0.82, 0.10 (rappresentati dalle circonferenze vuote).

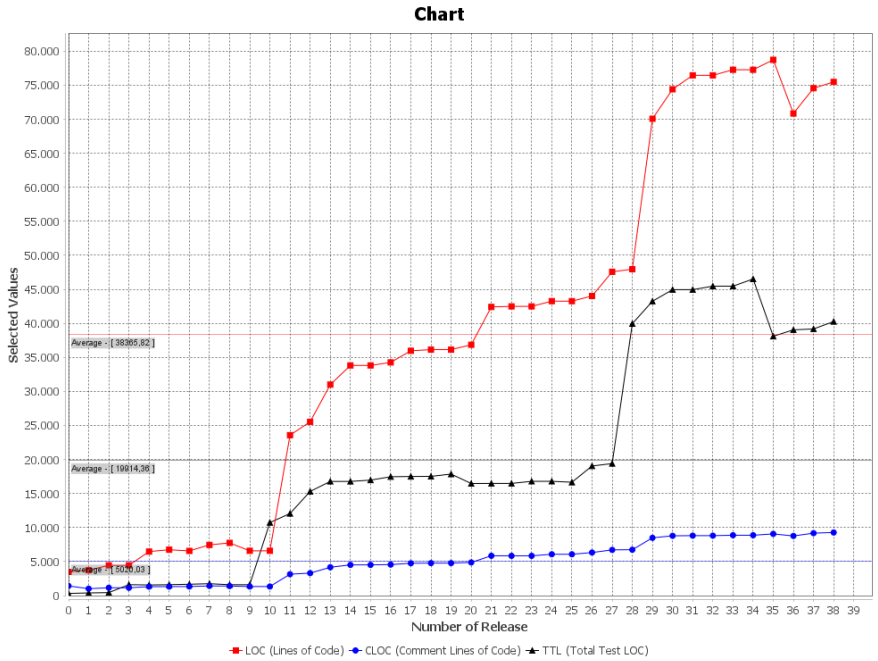


Immagine 4.25: Nella figura, sono riportati i grafici delle metriche *LOC, CLOC* e *TTL*.

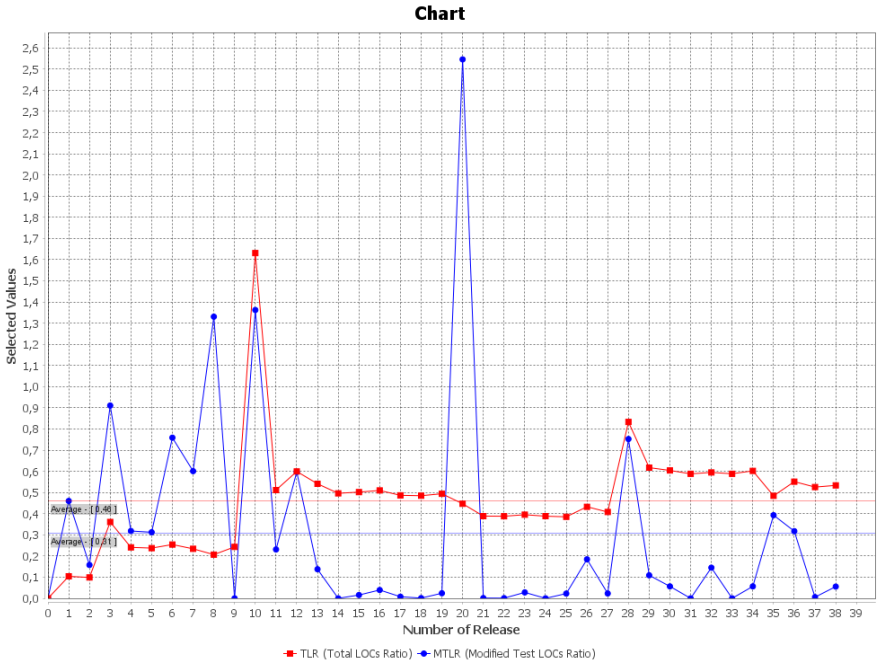


Immagine 4.26: Nella figura, sono riportati i grafici delle metriche *TLR* e *MTLR*.

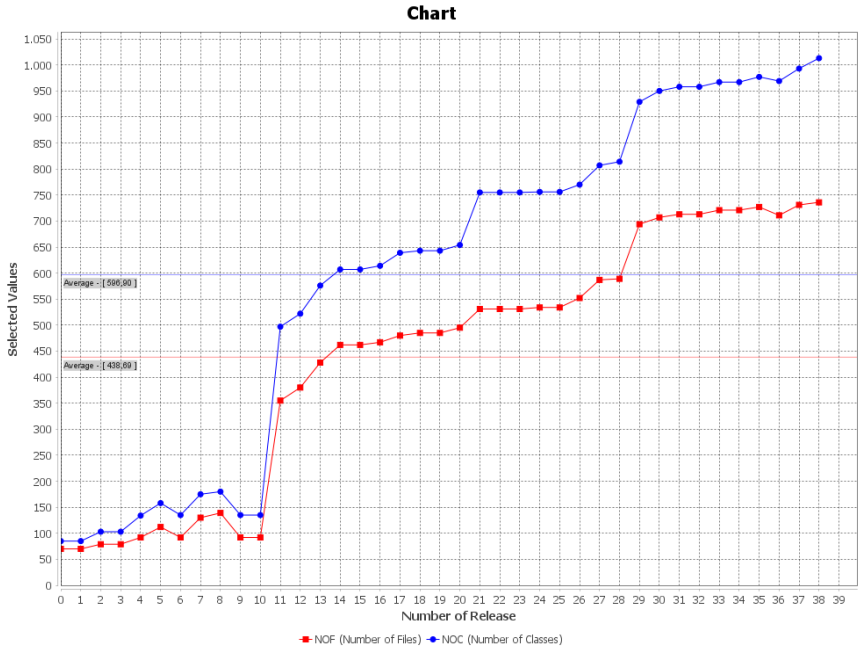


Immagine 4.27: Nella figura, sono riportati i grafici delle metriche *NOF* e *NOC*.

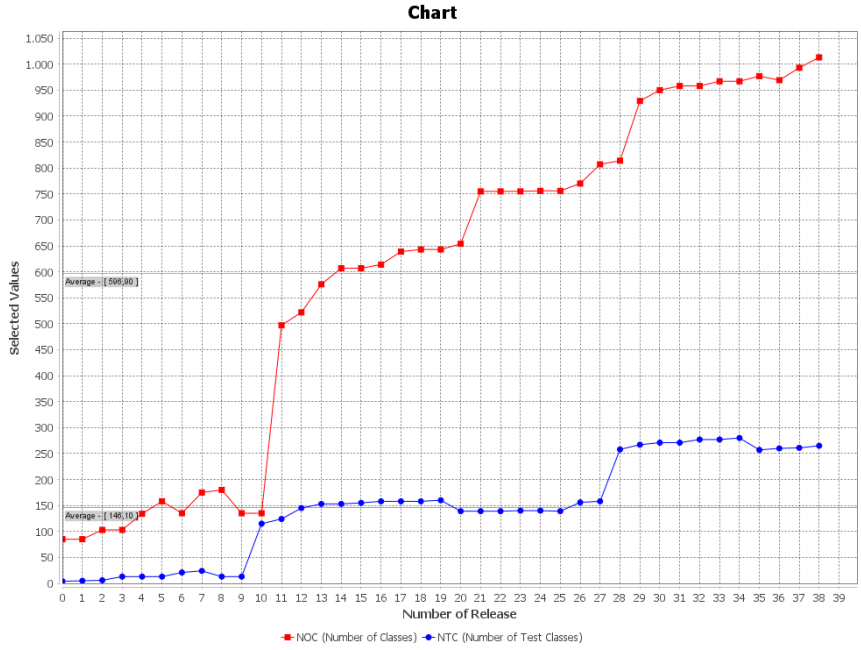


Immagine 4.28: Nella figura, sono riportati i grafici delle metriche *NOC* e *NTC*.

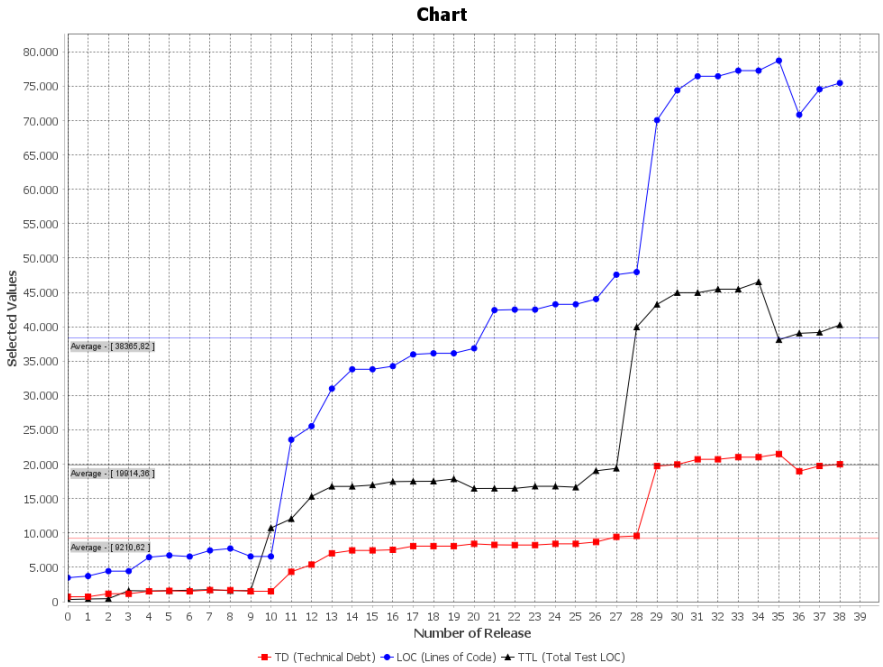


Immagine 4.29: Nella figura, sono riportati i grafici delle metriche *TD, LOC* e *TTL*.

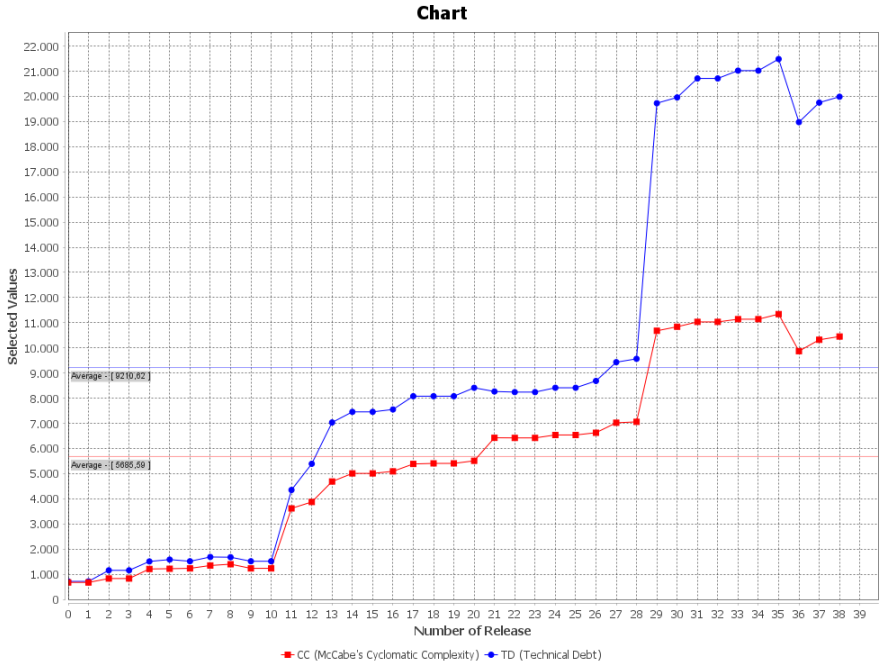


Immagine 4.30: Nella figura, sono riportati i grafici delle metriche *CC* e *TD*.

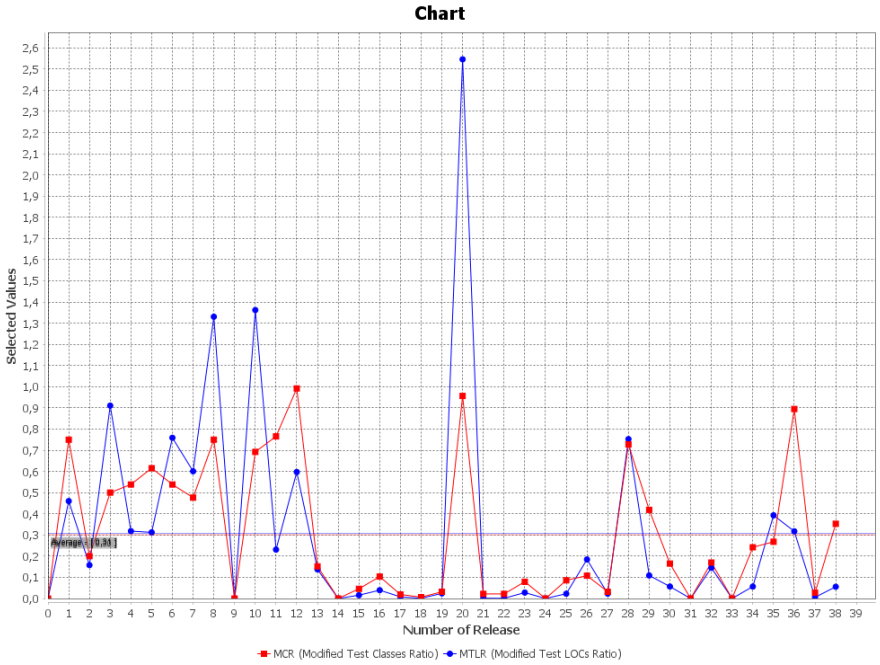


Immagine 4.31: Nella figura, sono riportati i grafici delle metriche *MCR* e *MTLR*.

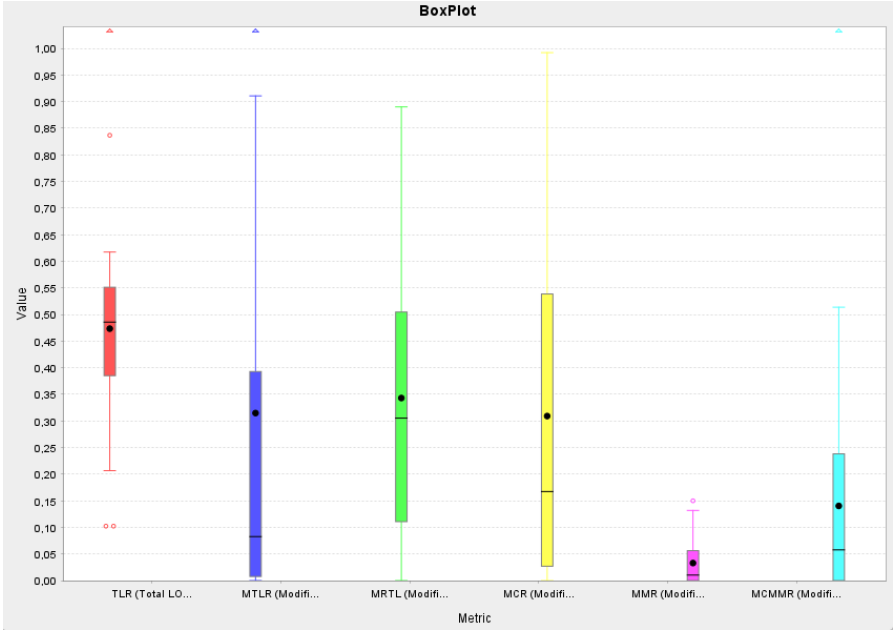


Immagine 4.32: Nella figura, è riportato il BoxPlot rappresentante le principali metriche ratio presenti nel tool.

Linkedin/test-butler

Test Butler[40] è uno strumento di test Android che permette di stabilizzare l'emulatore per aiutare a prevenire errori di test. Avendo una quindicina di release nello storico, si presta egregiamente alla rappresentazione grafica delle metriche.

Si analizzano ora alcuni grafici relativi al codice presente sul repository.

- In figura 4.33, sono riportati i grafici delle metriche *LOC, CLOC* e *TTL* dalle quali si evince che i test cases sono estremamente importanti nella programmazione, nella nona release, addirittura, le linee di test superano quelle di codice.

- In figura 4.34, sono riportati i grafici delle metriche ratio *TLR* e *MTLR*; il valore di *TLR* si assesta tra 0.7 e 0.8, indice di uno sviluppo costante del codice durante tutte le release escludendo la nona. In quest'ultima notiamo che le *TTL* sono raddoppiate, sinonimo di una grossa release del software in analisi.

- In figura 4.35, sono riportati i grafici delle metriche *NOF* e *NOC*, si nota che i numeri di file ed il numero di classi crescono regolarmente durante tutte le release. La rampa presente alla nona release, conferma il rilascio di un aggiornamento decisamente più corposo rispetto agli altri.

- In figura 4.36, sono riportati i grafici delle metriche *NOC* e *NTC*, i quali dimostrano che la progressione dello sviluppo di classi di test non è mai venuta meno durante tutto lo sviluppo. Il numero maggiore di classi di test rispetto alle classi classiche, che risulta nella nona versione, lascia intendere che i programmatori avessero già inserito i test cases per la nuova versione prima di rilasciarla.

- In figura 4.37, sono riportati i grafici delle metriche *TD, LOC* e *TTL*. Possiamo osservare con interesse come il TD cresca simultaneamente al numero di LOC presenti, mentre con il rilascio corposo della nona release le *LOC* sono all'incirca raddoppiate, il *TD* è pressochè triplicato, questo suggerisce che la nuova versione non fosse completamente pronta per il rilascio.

- In figura 4.38, sono riportati i grafici delle metriche *CC* e *TD*. In riferimento a quanto affermato nella didascalia di Figure 4.37, si può notare come la *CC* segua linearmente il *TD* indicando, quindi, che la release corposa abbia aumentato non di poco la complessità del software in analisi.

[40]https://github.com/linkedin/test-butler

- In figura 4.39, sono riportati i grafici delle metriche *MCR* e *MTLR*. Questo grafico permette di percepire quanto le modifiche di metodi di test hanno effetto sulla percentuale di classi di test modificate. Si nota un trend molto simile tra le due grandezze, la crescita di *MCR* non è un fattore positivo, indica infatti che durante il rilascio delle versioni con indice elevato (con *MCR* elevato si intende un valore che supera il 0.6.), una grossa percentuale di classi contenenti test cases ha dovuto subire variazioni per adattarsi alle nuove versioni del software.

- In figura 4.40, è riportato il BoxPlot rappresentante le principali metriche ratio presenti nel tool. Analizzando più nel dettaglio, si nota che la distribuzione statistica dei dati riguardanti *TLR* è molto compatta con varianza molto bassa, indice di uno sviluppo costante durante la maggiorparte delle release. Per quanto riguarda *MRTL* tutti i dati sono condensati nel valore 1 (non è una buona prospettiva in quanto significa che una parte significativa del tasso di abbandono del codice durante l'evoluzione dell'applicazione è stata necessaria per il mantenimento dei test case).

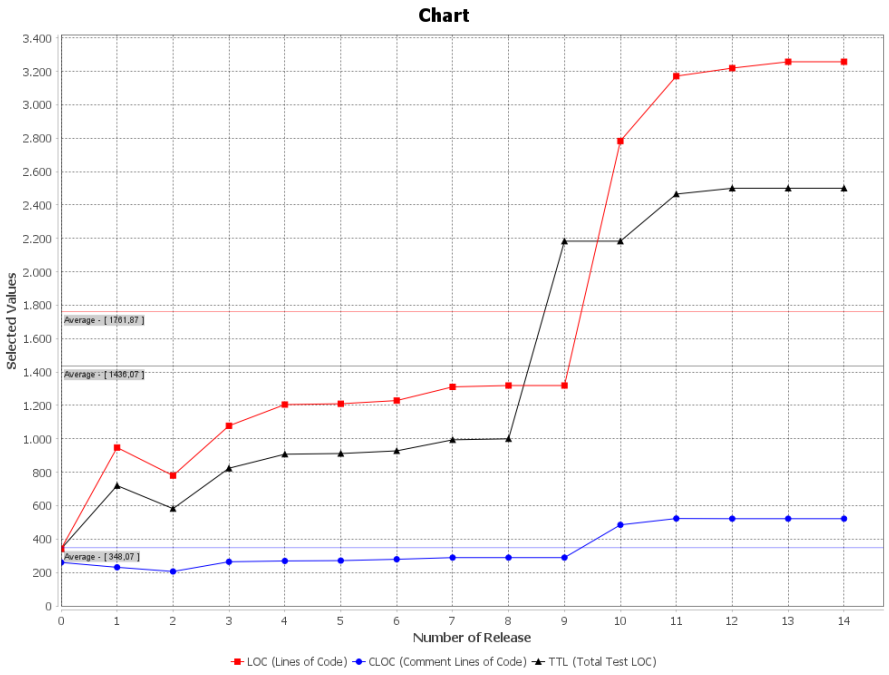


Immagine 4.33: Nella figura, sono riportati i grafici delle metriche *LOC, CLOC* e *TTL*.

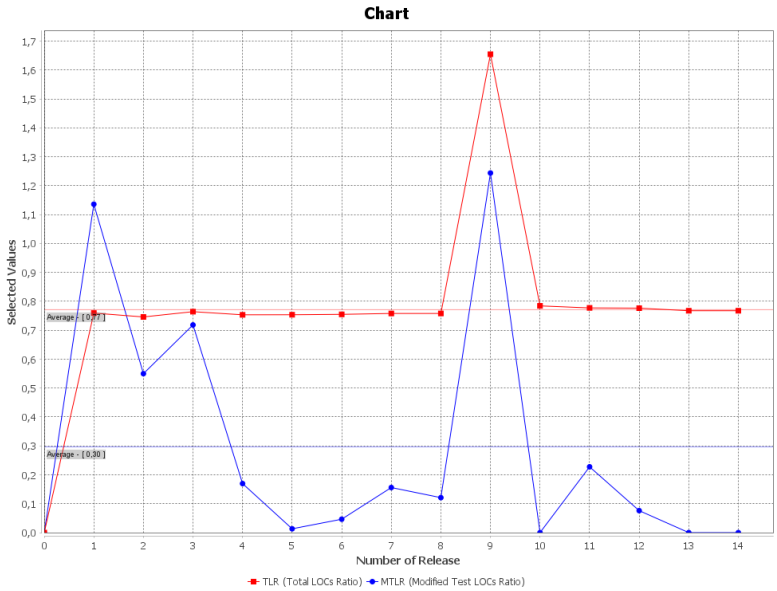


Immagine 4.34: Nella figura, sono riportati i grafici delle metriche ratio *TLR* e *MTLR*.

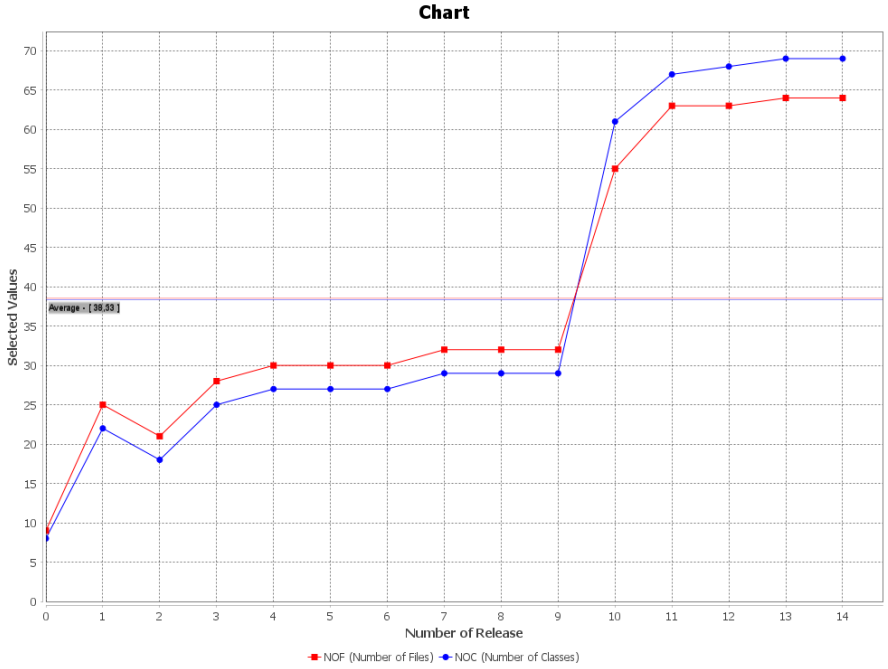


Immagine 4.35: Nella figura, sono riportati i grafici delle metriche *NOF* e *NOC*.

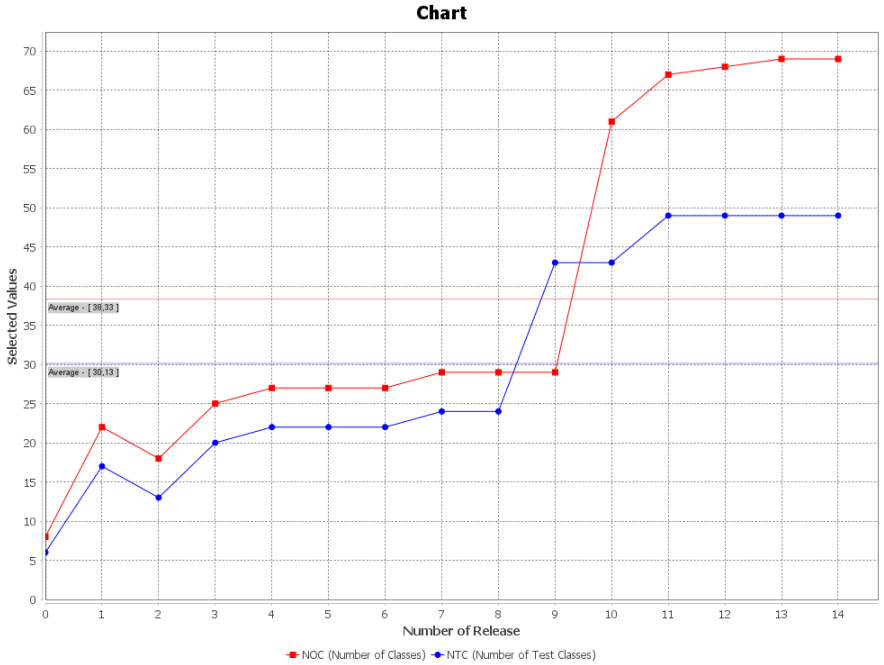


Immagine 4.36: Nella figura, sono riportati i grafici delle metriche *NOC* e *NTC*.

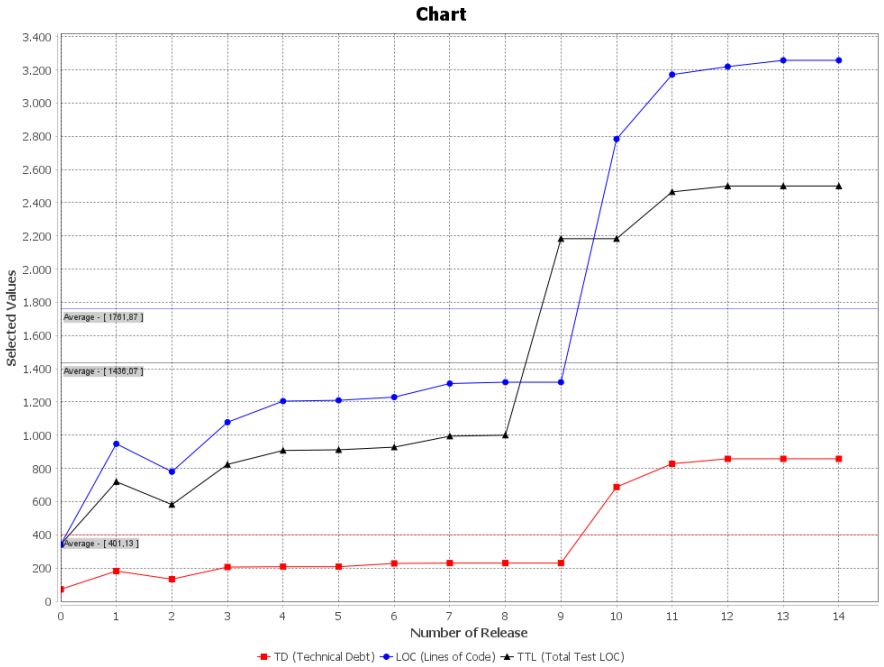


Immagine 4.37: Nella figura, sono riportati i grafici delle metriche *TD, LOC* e *TTL*.

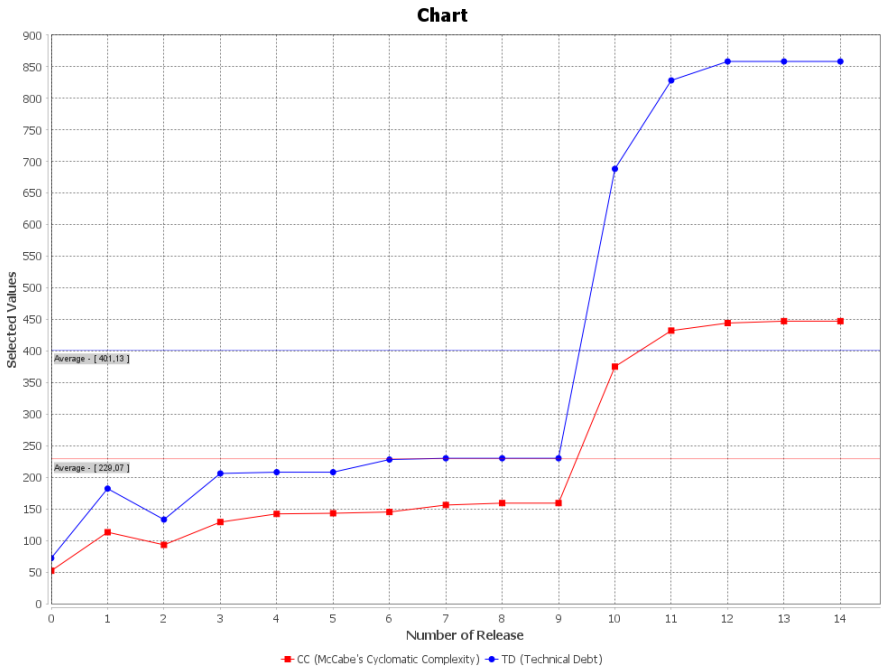


Immagine 4.38: Nella figura, sono riportati i grafici delle metriche *CC* e *TD*.

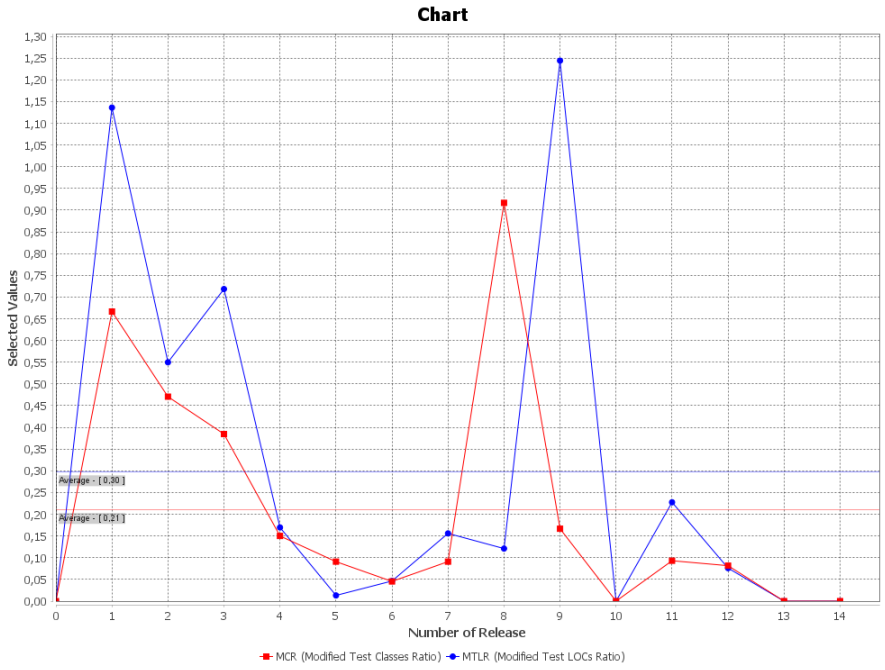


Immagine 4.39: Nella figura, sono riportati i grafici delle metriche *MCR* e *MTLR*.

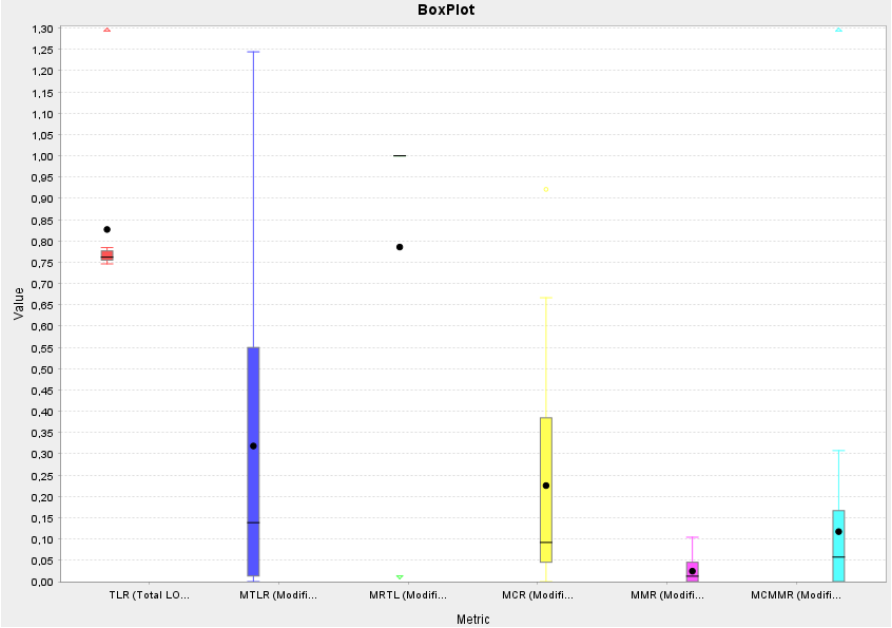


Immagine 4.40: Nella figura, è riportato il BoxPlot rappresentante le principali metriche ratio presenti nel tool.

Capitolo 5

Conclusioni

In questa sezione si analizzeranno le limitazioni, le applicazioni e i possibili upgrade futuri riguardanti questo lavoro di tesi.

5.1 Limitazioni

5.1.1 Generalizzabilità

Il lavoro di tesi è stato realizzato per ottenere informazioni su progetti ed applicazioni in ambito mobile Android. Essendo questo lo scope, gli script python lavorano attraverso un parser in formato JAR[1] che ha lo scopo di analizzare tutto il codice presente in ogni file e/o classe, digitalizzarlo e renderlo facilmente accessibile agli script. Essendo quindi lo scope molto ristretto, al momento, la tesi è applicabile solo ed esclusivamente su repository che contengono applicazioni sviluppate in linguaggio Java e/o Kotlin[2]; questo non limita però delle possibili future implementazioni con altri linguaggi anche non solo dedicati alla programmazione mobile. Sostituendo il parser presente al momento con un parser capace di esaminare linguaggi come C++[3] o python[4] piuttosto che Javascript o PHP, sarà possibile utilizzare gli stessi script per ottenere le metriche sui progetti più disparati.

---

[1] In informatica un file con estensione JAR (Java Archive) indica un archivio dati compresso (ZIP) usato per distribuire raccolte di classi Java. Tali file sono concettualmente e praticamente assimilabili a package, e quindi talvolta associabili al concetto di libreria.https://it.wikipedia.org/wiki/JAR_(formato_di_file)

[2] https://developer.android.com/kotlin

[3] https://it.wikipedia.org/wiki/C%2B%2B

[4] https://it.wikipedia.org/wiki/Python

5.1.2 Errori/Inaffidabilità

La dipendenza degli script python dal parser per il calcolo delle metriche non esclude la possibilità di ricevere dei dati non propriamente veritieri in seguito a problemi con file contenenti una semantica non consona agli standard. Questo porta ad una rimozione di tutti i file che presentano errori di semantica andando così ad alterare le metriche che derivano da queste classi. Una seconda problematica è rappresentata dal fatto che, se si analizza un progetto con un grosso quantitativo di linee di codice e/o test (il codice è stato testato con progetti che raggiungevano le 500.000 righe di codice e ha funzionato correttamente), la procedura di calcolo delle metriche e di parsing del codice richiede molto tempo di calcolo ed elaborazione.

5.1.3 Interpretazione delle metriche

L'interpretazione delle metriche costituisce la problematica più evidente. La possibilità che le metriche non vadano a coincidere con delle vere e proprie problematiche di manutenibilità del codice è significativa. Ad esempio, un progetto con livelli elevati della metrica MTLR durante le prime release avrà una fragilità decisamente minore rispetto ad un progetto che presenta gli stessi livelli ma in release più avanzate; nel primo esempio sono "normali" livelli elevati di modifiche percentuali nei test in quanto il codice di base dell'applicazione probabilmente era ancora in fase di definizione. Un altro esempio di dati "anomali" può essere un'applicazione molto compatta, nella quale le modifiche, per quando possano includere poche righe di codice, tendono a far raggiungere livelli molto elevati alle metriche ratio, suggerendo erroneamente una fragilità elevata.

5.2 Applicazioni

Visto il range di progetti che il codice di questa tesi può analizzare, si ipotizzano i seguenti ambiti di applicazione. Il Software che compone la tesi è disponibile liberamente sul seguente repository open-source[5].

- *Ambito industriale*: a livello industriale, il lavoro di questa tesi può trovare applicazioni per fornire ai manager degli indici di qualità e sostenibilità del codice di test sviluppato, fornendo un rapido feedback senza necessità di dover controllare manualmente le modifiche effettuate ad ogni release. La feature che consente di accostare più grafici con tutte le metriche desiderate potrebbe trovare svariate applicazioni per il supporto allo sviluppo.

[5] https://github.com/Frenky95/FragilityMetricGraphicTool

- *Ambito Open-Source*: questo software che compone la tesi può essere adattato con numerose piccole modifiche in ambito open source.
- *Ambito Ricerca*: il lavoro di questa tesi può essere incorporato all'interno di progetti open-source, o esteso da sviluppatori facenti parte della community, in modo da fornire misurazioni di progetti esistenti o di essere esteso per fornire supporto ad altri domini, altri linguaggi di test e altre metriche.

5.3 Upgrade Futuri

Nell'ingegneria del software, l'integrazione continua (continuous integration in inglese, spesso abbreviato in CI) è una pratica che si applica in contesti in cui lo sviluppo del software avviene attraverso un sistema di controllo versione. Consiste nell'allineamento frequente (ovvero "molte volte al giorno") dagli ambienti di lavoro degli sviluppatori verso l'ambiente condiviso. Il concetto è stato originariamente proposto nel contesto dell'extreme programming, come contromisura preventiva per il problema dell'integration hell (le difficoltà dell'integrazione di porzioni di software sviluppati in modo indipendente su lunghi periodi di tempo e che di conseguenza potrebbero essere significativamente divergenti). La CI può essere considerata come un'estremizzazione di idee già presenti in altri metodi precedenti all'XP. La CI è stato originariamente concepito per essere complementare rispetto ad altre pratiche, in particolare legate al Test Driven Development (sviluppo guidato dai test, TDD). Si suppone generalmente che siano stati predisposti test automatici eseguibili dagli sviluppatori immediatamente prima di rilasciare i loro contributi verso l'ambiente condiviso, in modo da garantire che le modifiche non introducano errori nel software esistente. Il software che compone questo lavoro di tesi è integrabile direttamente all'interno della piattaforma GitHub Actions[6]: può essere configurato per attivarsi ad ogni nuovo commit, così da eseguire un ricalcolo automatico delle metriche ed analizzare gli effetti dell'ultimo commit.

Si prevede, infine, di utilizzare il tool per effettuare delle sperimentazioni empiriche sulla fragilità in progetti open-source.

[6] https://docs.github.com/en/actions

Bibliografia

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. Memon. *Using gui ripping for automated testing of android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012b, pp. 258–261 (cit. on pp. 7, 14).

[2] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon. *Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, pp. 1–10 (cit. on p. 9).

[3] H. Muccini, A. Di Francesco, and P. Esposito. «Software testing of mobile applications: Challenges and future research directions. In Proceedings of the 7th International Workshop on Automation of Software Test». In: (2012), pp. 29–35 (cit. on pp. 10, 16).

[4] J. Gao, X. Bai, W. Tsai, and Uehara. «Mobile application testing: a tutorial. Computer». In: (2014), pp. 46–55 (cit. on p. 12).

[5] K Anureet. «Review of mobile applications testing with automated techniques. interface, 4.» In: (2015) (cit. on pp. 12, 16).

[6] M. Linares-Vásquez. *Enabling testing of android apps. In Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, volume 2.* IEEE, 2015, pp. 763–765 (cit. on p. 12).

[7] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. *Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 399–410 (cit. on p. 12).

[8] S. R. Choudhary, A. Gorla, and A. Orso. *Automated test input generation for android: Are we there yet? arXiv preprint arXiv:1503.07217.* (Cit. on pp. 13, 14).

[9] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland, and S. Gopalakrishnan. *Towards unit testing of user interface code for android mobile applications. In International Conference on Software Engineering and Computer Systems*. Springer, 2011, pp. 163–175 (cit. on p. 13).

[10] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. *Reran: Timingand touch-sensitive record and replay for android. In Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 72–81 (cit. on p. 14).

[11] Y. Hu, T. Azim, and I. Neamtiu. *Versatile yet lightweight recordand-replay for android. In ACM SIGPLAN Notices, volume 50*. ACM, 2015, pp. 349–366 (cit. on p. 14).

[12] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. *Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 215–224 (cit. on p. 14).

[13] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. *Barista: A technique for recording, encoding, and running platform independent android tests. In Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 149–160 (cit. on p. 14).

[14] K. Moran, R. Bonett, C. Bernal-Cárdenas, B. Otten, D. Park, and D. Poshyvanyk. *On-device bug reporting for android applications. In Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, 2017, pp. 215–216 (cit. on pp. 14, 15).

[15] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai. *On the accuracy, efficiency, and reusability of automated test oracles for android devices.* 2014, pp. 957–970 (cit. on p. 14).

[16] A. Machiry, R. Tahiliani, and M. Naik. *Dynodroid: An input generation system for android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234 (cit. on p. 14).

[17] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. *Crashscope: A practical tool for automated testing of android applications. In Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 15–18 (cit. on p. 14).

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. *Mobiguitar: Automated model-based testing of mobile apps. IEEE software.* 2015, pp. 53–59 (cit. on p. 14).

[19] W. Choi, G. Necula, and K. Sen. *Guided gui testing of android apps with minimal restart and approximate learning. In Acm Sigplan Notices, volume 48*. ACM, 2013, pp. 623–640 (cit. on p. 14).

[20] P McMinn. *Search-based software testing: Past, present and future. In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 153–163 (cit. on p. 15).

[21] R. Mahmood, N. Mirzaei, and S. Malek. *Evodroid: Segmented evolutionary testing of android apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609 (cit. on p. 15).

[22] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek. *Energy-aware test-suite minimization for android apps. In Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 425–436 (cit. on p. 15).

[23] K. Mao, M. Harman, and Y Jia. *Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105 (cit. on p. 15).

[24] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. *Auto-completing bug reports for android applications. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 673–686 (cit. on p. 15).

[25] M. E. Joorabchi, A. Mesbah, and P Kruchten. *Real challenges in mobile app development. In Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 15–24 (cit. on p. 16).

[26] B. Kirubakaran and V. Karthikeyani. *Mobile application testing—challenges and solution approach through automation. In Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on*. IEEE, 2013, pp. 79–84 (cit. on p. 16).

[27] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. *How do developers test android applications? In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 613–622 (cit. on p. 17).

[28] P. S. Kochhar, F. Thung, T. Nagappan N.and Zimmermann, and D. Lo. *Understanding the test automation culture of app developers.* 2015 (cit. on p. 17).

[29] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. *Understanding android fragmentation with topic analysis of vendor-specific bugs. In Reverse Engineering (WCRE), 2012 19th Working Conference on.* IEEE, 2012, pp. 83–92 (cit. on p. 17).

[30] H. K. Ham and Y. B. Park. *Mobile application compatibility test system design for android fragmentation. In International Conference on Advanced Software Engineering and Its Applications.* Springer, 2011, pp. 314–320 (cit. on p. 17).

[31] L. Wei, Y. Liu, and S.-C. Cheung. *Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2016, pp. 226–237 (cit. on p. 17).

[32] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. *The peril of fragmentation: Security hazards in android device driver customizations. In Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 2014, pp. 409–423 (cit. on p. 17).

[33] H. S. Alamri and B. A. Mustafa. *Software engineering challenges in multi platform mobile application development. Advanced Science Letters.* 2014, 20(10-11):2115–2118 (cit. on p. 20).

[34] A. Ahmad, K. Li, C. Feng, S. M. Asim, A. Yousif, and S. Ge. *An empirical study of investigating mobile applications development challenges. IEEE Access.* 2018, 6:17711–17728 (cit. on p. 20).

[35] S. Berner, R. Weber, and R. K. Keller. *Observations and lessons learned from automated testing. In Proceedings of the 27th international conference on Software engineering.* 2005, pp. 571–579 (cit. on p. 21).

[36] M. et al. Fewster. *Common mistakes in test automation. In Proceedings of Fall Test Automation Conference.* 2001 (cit. on p. 22).

[37] E. Sjösten-Andersson and L. Pareto. *Costs and benefits of structureaware capture/replay tools.* 2006, p. 3 (cit. on p. 22).

[38] E. Borjesson and R. Feldt. *Automated system testing using visual gui testing tools: A comparative study in industry. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* IEEE, 2012, pp. 350–359 (cit. on p. 22).

[39] E. Alegroth, R. Feldt, and H. H. Olsson. *Transitioning manual system test suites to automated testing: An industrial case study. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* IEEE, 2013, pp. 56–65 (cit. on p. 22).

[40] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. *Guitar: an innovative tool for automated testing of gui-driven software. Automated Software Engineering*. IEEE, 2014, 21(1):65–105 (cit. on p. 22).

[41] V. Garousi and M. Felderer. *Developing, verifying, and maintaining high-quality automated test scripts.* IEEE Software, 2016, (3):68–75 (cit. on p. 22).

[42] M. Grechanik, Q. Xie, and C. Fu. *Maintaining and evolving guidirected test scripts. In Proceedings of the 31st international conference on software engineering*. IEEE Computer Society, 2009, pp. 408–418 (cit. on p. 22).

[43] A. M. Memon. *Automatically repairing event sequence-based gui test suites for regression testing. ACM Transactions on Software Engineering and Methodology (TOSEM)*. 2008, 18(2):4 (cit. on p. 22).

[44] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon. *Sitar: Gui test script repair. Ieee transactions on software engineering*. 2016, pp. 170–186 (cit. on p. 22).

[45] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. *Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 272–281 (cit. on pp. 23, 24).

[46] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. *Visual vs. dombased web locators: An empirical study. In International Conference on Web Engineering*. Springer, 2014, pp. 322–340 (cit. on pp. 23, 24).

[47] S. McMaster and A. M. Memon. *An extensible heuristic-based framework for gui test case maintenance. In 2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 251–254 (cit. on p. 23).

[48] A. M. Memon and M. L. Soffa. *Regression testing of guis. ACM SIGSOFT Software Engineering Notes*. 2003, 28(5):118–127 (cit. on p. 23).

[49] V. G. Yusifoglu, Y. Amannejad, and A. B. Can. *Software test-code engineering: A systematic mapping. Inf. Softw. Technol., vol. 58*. IEEE, 2015, pp. 123–147 (cit. on p. 25).

[50] X. Tang, S. Wang, and K. Mao. *Will this bug-fixing change break regression testing? in Proc. IEEE ACM/IEEE Int. Symp. Empirical Softw. Eng.* Meas, 2015, pp. 1–10 (cit. on p. 25).