Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi della correlazione tra la qualità del codice e la fragilità del test

Relatore:
Prof. Luca Ardito

Candidato:
Simone Pirrigheddu

Co-relatori:
Prof. Maurizio Morisio
Dott. Riccardo Coppola

Anno accademico 2020/21
Torino

Sommario

***Contesto*:** Nell'ambito dello sviluppo software, la fase di test è una parte fondamentale del ciclo di vita di un programma in quanto permette di individuare eventuali problemi, carenze o errori presenti all'interno del software da correggere prima che esso venga rilasciato agli utenti finali. Uno dei problemi principali nella scrittura dei test è il fatto che i test che servono per verificare il corretto funzionamento della GUI (Graphical User Interface) risultano essere molto fragili in quanto quest'ultima viene modificata spesso ed è in continua evoluzione in ogni versione del software. Questo causerebbe il fallimento dei test riguardanti la GUI in caso essi non venissero aggiornati ogni versione che presenta una modifica dell'interfaccia grafica nel codice sorgente.
Inoltre, nel campo dell'Ingegneria del Software si è da diversi anni ritenuto necessario definire quando e come il codice sorgente di un programma risulti di qualità. Sono stati definiti infatti una serie di parametri interni ed esterni che permettono agli sviluppatori di stabilire quanta qualità presenti il codice. Uno di questi parametri interni è la manutenibilità del sistema, cioè la facilità di apportare delle modifiche una volta che il software viene ultimato.

***Obiettivo*:** L'obiettivo di questo lavoro di tesi è fare un'analisi che permetta di dimostrare una possibile correlazione, attraverso lo studio del codice di un set di diversi software e delle loro suite di test in tutte le versioni, tra la manutenibilità del codice sorgente del programma e la fragilità dei test che vengono formulati durante l'evoluzione del software stesso.

***Metodo*:** Lo strumento sviluppato analizza applicazioni di tipo Android e si concentra sul calcolo di metriche statiche di fragilità dei test e di manutenibilità del codice sorgente.

***Risultati*:** Lo studio di correlazione attraverso la regressione lineare semplice applicata alle metriche di qualità e alle metriche di fragilità ha portato alla presenza di molte relazioni di tipo lineare tra la prima tipologia di metriche e la seconda. Si può dunque parlare di correlazione parziale tra le metriche, con quelle di qualità che hanno un effetto su quelle di fragilità.

***Conclusioni*:** Lo script in Python ha raggiunto gli obiettivi per i quali era stato sviluppato: analizzare le metriche di manutenibilità e di fragilità di applicazioni di tipo Android scritte in Java e Kotlin. Inoltre la correlazione parziale è stata evidenziata attraverso la regressione lineare semplice. Infine è possibile applicare delle migliorie a questo lavoro soprattutto utilizzando regressioni più raffinate, come la *regressione lineare multipla* e la *stepwise regression*.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione e lavori correlati

1.1 Introduzione al testing

Il test del software è una fase importante nel ciclo di vita dello sviluppo perché aiuta a identificare i bug presenti in un sistema software prima che venga messo a disposizione dei suoi utenti finali (Cruz et al., 2019).

Durante questi anni si è discusso molto su quale fosse il modo migliore di testare i software. Nella letteratura sono presenti due tipologie differenti per la fase di testing: il test manuale e il test automatizzato. La differenza tra i due approcci è sostanzialmente che nel primo i test vengono eseguiti manualmente dall'uomo, mentre nel secondo i test vengono eseguiti tramite uno script che lancia i test in maniera automatica.

Nonostante gli sviluppatori preferiscano utilizzare test manuali, recenti studi (Mateen et al., 2018) hanno evidenziato come i test automatizzati siano migliori in quanto più affidabili, accurati e flessibili rispetto ai test manuali. Infatti, in questi ultimi l'operatore che svolge i test può commettere degli errori che potrebbero portare a bug o ad altri problemi una volta che il software viene messo in commercio. Inoltre, i test automatizzati possono essere riutilizzati per testare altri prodotti. Infine, in un primo momento i test automatizzati sono più lenti in quanto bisogna scrivere gli script di test prima di poter testare il programma, ma nel conteggio totale del tempo impiegato nel testare il software, i due tempi di esecuzione non sono minimamente comparabili: nei test automatici infatti c'è un risparmio notevole dovuto al fatto che, una volta lanciati, gli sviluppatori non devono far altro che attendere i risultati e possono impiegare il loro tempo migliorando la qualità del

codice sorgente del programma, e inoltre tutte le operazioni di verifica, *logging* e *reporting* sono automatizzate.

1.1.1 Test automatizzati della GUI

Quando si vuole testare un software, i test automatizzati possono essere eseguiti su livelli diversi, alcuni più tipicamente utilizzati nei test delle applicazioni di tipo mobile (Coppola et al., 2019):

- *Unit Testing:* per testare la correttezza e la completezza di un programma visto come singolo modulo.
- *Integration Testing:* per verificare la correttezza funzionale nell'iterazione tra più moduli del software.
- *System Testing:* per testare particolari proprietà globali dell'intero applicativo software. Si può definire *End-to-End Testing* quando il sistema viene testato per come viene offerto agli utenti finali, e nel caso in cui il software possieda una GUI, si parla di *GUI Testing* in quanto i test che vengono effettuati passano attraverso di essa.
- *Regression Testing:* per verificare se ad ogni nuovo aggiornamento di un programma la nuova versione mantiene le stesse funzionalità della precedente.
- *Compatibility Testing:* per testare che l'applicazione funzioni su diversi modelli di palmare e/o versioni del sistema operativo.
- *Performance Testing:* per garantire che i dispositivi non consumino troppe risorse.
- *Security Testing:* per verificare che vengano rispettati gli standard di sicurezza del software.

Inoltre nella letteratura dell'Ingegneria del Software sono state fornite diverse classificazioni per quanto riguarda l'automatizzazione dei test di tipo *GUI Testing* del software.

Per definizione, il test della GUI (*GUI testing*) è il processo utile a garantire la corretta funzionalità della Graphical User Interface per una data applicazione e ad assicurarsi che sia conforme alle sue specifiche scritte. Si tratta di una forma

di *System testing* che esamina l'applicazione finita esplorando la GUI tramite dei comandi di prova.

Una classificazione è stata illustrata da Alégroth et al. (2015) che hanno suddiviso le tecniche di test automatizzate della GUI dell'Application Under Test in base a come vengono identificati gli elementi sullo schermo:

- *Coordinate-based:* identificano gli elementi da testare in base alla loro posizione assoluta sullo schermo.
- *Widget/Component-based:* identificano gli elementi dell'interfaccia in base alla definizione e disposizione dei layout e delle loro proprietà.
- *Image recognition-based:* identificano gli elementi sullo schermo in base al loro aspetto visivo.

Inoltre, Linares-Vásquez et al. (2017) ha illustrato sette categorie di strumenti e diversi tipi di approcci (Linares-Vásquez (2015)) che possono essere utilizzati per i test automatizzati della Graphical User Interface. Queste classificazioni si basano sul livello di conoscenza del codice sorgente e sul modo in cui gli input vengono generati e distribuiti:

- *Automation APIs/Frameworks:* considerati uno degli strumenti di test più semplici e più potenti, forniscono un'interfaccia per ottenere informazioni relative alla GUI, come la gerarchia di widget che esistono su una schermata e per simulare le interazioni dell'utente con un dispositivo. Inoltre, possono essere utilizzati per scrivere manualmente il codice di test con sequenze di operazioni da essere eseguite sull'Application Under Test.
- *Record and Replay Tools:* questi strumenti registrano le operazioni eseguite sulla GUI per generare sequenze di test ripetibili, fornendo così un'alternativa più veloce ed economica alla scrittura manuale di script di test. Si tratta di un approccio più vicino ai test di tipo manuale in quanto richiede la registrazione di nuove tracce/script se la GUI viene modificata. Inoltre, è richiesto uno script per ogni dispositivo in quanto gli script sono accoppiati alle posizioni nello schermo. L'efficacia di questo approccio dipende dalla capacità di rappresentare eventi molto complessi (Esempio pratico: *MonkeyRecorder*[1]).

[1]http://code.google.com/p/android-monkeyrunner-enhanced/

- *Automated Test Input Generation Techniques:* include tutti gli strumenti dove il processo di generazione dell'input viene automatizzato per alleggerire il carico agli sviluppatori. Comprende tre categorie di approcci differenti:

  - *Random-Based Input Generation*, si tratta di test casuali, che non richiedono la conoscenza completa dell'AUT, facili da impostare ma che potrebbero invadere la GUI di azioni irrealizzabili. Uno degli strumenti più utilizzati per questo approccio è *UI Monkey*[2];
  - *Systematic Input Generation*, viene generato un modello (in maniera manuale o automatica) che permette l'esplorazione dell'AUT. Questo descrive i componenti della GUI, le sue schermate e la fattibilità delle azioni su ogni componente. La generazione del modello può avvenire prima dell'esplorazione o iterativamente dopo l'esecuzione di un nuovo evento. Questo approccio ottiene una copertura elevata di tutti i componenti della GUI dell'AUT. Un esempio pratico di questo tipo è *AndroidRipper*[3]
  - *Model-Based Input Generation*, dove si richiede la generazione di un modello (manuale o automatico). Questo metodo si basa sulla generazione di eventi che formano uno scenario di test. Uno dei problemi principali di questo approccio è che alcuni eventi nella sequenza non sono fattibili nel contesto del test. Uno strumento molto utilizzato per questo metodo è *SwiftHand*[4].

- *Bug and Error Reporting:* strumenti che aiutano gli sviluppatori nella scrittura del codice di test identificando e segnalando i bug.

- *Monitoring Tools:* strumenti per il monitoraggio degli arresti anomali e del consumo di risorse in fase di esecuzione del programma.

- *Mobile Testing Services:* questi strumenti forniscono tipicamente quattro tipi di servizi di test differenti: *Crowd-Sourced Functional Testing*, utilizzato per inviare segnalazioni di bug nelle applicazioni; *Usability Testing* che misura il

[2]https://developer.android.com/studio/test/monkey.html

[3]https://github.com/reverse-unina/AndroidRipper

[4]https://github.com/wtchoi/SwiftHand

design grafico di un'applicazione focalizzata sulla facilità d'uso e sull'intuitività; *Security Testing*, usato per individuare e segnalare eventuali problemi nel campo della sicurezza delle applicazioni; *Localization testing* che garantisce il giusto utilizzo dell'applicazione in diverse regioni geografiche e con lingue diverse in tutto il mondo.

- *Device Streaming Tools:* questi strumenti aiutano gli sviluppatori ad eseguire un check-up di un dispositivo connesso sul proprio PC personale o accedere ai dispositivi in remoto tramite Internet.

1.1.2 La fragilità dei test

Nell'Ingegneria del Software la fragilità dei test è un argomento ampiamente esplorato in letteratura in quanto rappresenta un grave problema per i software sia di tipo mobile che desktop. Diversi studi si sono interrogati sulla definizione di questo problema che riguarda in generale le classi di test, sia di tipo manuale che automatizzate.

Un'importante definizione è presente nell'articolo di Garousi and Felderer (2016): un test viene definito *fragile* quando esso non compila o fallisce in seguito a modifiche sul System Under Test (SUT) che non influenzano la parte che il test sta esaminando.

Nel campo mobile un'altra definizione è stata descritta nell'articolo di Coppola et al. (2019): un test case della GUI è *fragile* se richiede uno o più interventi quando l'applicazione si evolve in versioni successive a causa di qualsiasi modifica effettuata sull'Application Under Test (AUT).

Entrambe le definizioni portano a capire come i test cosiddetti fragili creano degli intoppi ai programmatori e allo stesso tempo portano i costi di manutenzione del software ad aumentare, costringendo gli sviluppatori a perdere molto tempo nel modificare codici di test fragili ogni qual volta vengano modificate le funzionalità del programma (Garousi & Felderer, 2016).

Molti autori hanno cercato di risolvere il problema illustrando approcci e strumenti che potessero permettere di correggere automaticamente i test che falliscono o non compilano specialmente riguardanti la GUI del software.

Un paio di esempi pratici utili per la localizzazione e la correzione di errori dovuti alla fragilità dei test sono:

- **ReAssert with Symbolic test repair:** Brett et al. (2010) propongono uno strumento che genera suggerimenti di riparazione in maniera automatica per il codice di test, consentendo all'utente di accettarli, quindi riparare immediatamente il caso di test. La tecnica iniziale di *ReAssert* prevede la combinazione dell'analisi dell'esecuzione dinamica dei test con l'analisi e la trasformazione della struttura statica degli stessi test. Nella variante migliorata con *Symbolic test repair* lo strumento effettua le riparazioni attraverso la *Symbolic execution*, che non è altro che una tecnica di analisi del programma che esegue calcoli su valori simbolici piuttosto che valori concreti, incrementando le prestazioni già ottenute con ReAssert in forma standard.

- **ScrIpT repAireR (SITAR)**, proposto da Gao et al. (2016), è un tool che utilizza tecniche di reverse engineering per riparare automaticamente le suite di test modellando i casi di test con l'aiuto di Event-Flow Graphs (EFG). Oltre a riparare il codice di test, SITAR ripara anche il riferimento agli oggetti della Graphical User Interface producendo uno script di test finale che può essere eseguito automaticamente per convalidare il software.

- **VISTA** è uno strumento di Stocco et al. (2018) che permette di riparare i test di un software di tipo web attraverso una meccanismo basato su una pipeline di elaborazione di immagini. La novità di questo strumento è che, a differenza degli altri presenti fin'ora in commercio, VISTA non si basa sul codice sorgente del programma ma focalizza la sua attenzione direttamente nei compomenti della Graphical User Interface del software.

1.1.3 Cause di modifica nelle suite di test

La fragilità delle suite di test è strettamente collegata agli interventi che vengono effettuati nel codice sorgente del programma. Questi non sono altro che le modifiche effettuate dagli sviluppatori software durante l'evoluzione del programma stesso. Yusifoglu et al. (2015) evidenziano come il codice di test necessiti di mantenimento durante le varie versioni rilasciate del software. Il mantenimento può essere di quattro tipi differenti:

- **Perfective maintenance of test code:** miglioramento della qualità del codice di test, ad esempio, modifiche di refactoring con l'uso di pattern di codice di test.

- **Adaptive maintenance of test code:** manutenzione riflessa dal codice sorgente che viene a sua volta migliorato.
- **Preventive maintenance of test code:** valutazione della qualità del codice di test, ad esempio rilevamento di *smell code* o di ridondanze.
- **Corrective maintenance of test code:** individuazione e correzione di bug o errori nel codice di test.

Per quanto riguarda le cause di modifica nei metodi di test, una classificazione ben delineata è stata rilasciata da Coppola et al. (2019). Le singole cause sono state divise in nove macrocategorie in base alla loro correlazione alla GUI dell'Application Under Test: la categoria *Test Code Change* non è collegata alla GUI dell'AUT; *Application Code Change*, *Execution Time Variability* e *Compatibility Adaptations* sono collegate all'AUT ma non nello specifico alla GUI corrispondente; le altre macrocategorie sono correlate alla GUI dell'AUT, ai suoi elementi e ai suoi widget.

Seguendo la classificazione di Yusifoglu et al. (2015), la categoria *Test Code Change* è l'unica che riguarda le modifiche relative alle manutenzioni di tipo *Perfective*, *Preventive* e *Corrective*, mentre tutte le altre macrocategorie coprono le modifiche che vengono operate sul codice di test per eseguire la manutenzione di tipo *Adaptive*.

- **Test Code Change:** include le modifiche nel codice di test che non sono dovute a modifiche dell'interfaccia grafica, a cambiamenti nel comportamento e nelle funzionalità effettive dell'applicazione, ma solo al modo in cui i test vengono effettivamente impostati ed eseguiti. Inoltre, queste modifiche riguardano solo le classi di test e non c'è correlazione con il codice sorgente del programma. All'interno di questa macrocategoria sono presenti: *Test Logic Change* (modifiche di tipo logico, come API o deprecazioni), *Changed Assertions* (modifiche nelle asserzioni, come sul caso d'utilizzo), *Test refactoring* (modifiche di refactoring), *Logging* (modifiche di registrazione), *Screenshots* (modifiche nelle acquisizioni delle schermate che vengono utilizzate per testare l'applicazione), *Test Syntax Corrections and Comments* (modifiche di tipo sintattico o commenti).
- **Application Code Change:** include le modifiche dovute a cambiamenti nel codice sorgente del software che non sono correlati all'aspetto grafico

dell'applicazione, cioè che riguardano solamente le funzionalità dell'AUT. Le varie categorie incluse sono: *Application Functionalities Change* (modifiche di funzionalità fornite all'applicazione), *Application Startup/Intents* (modifiche nelle definizioni e nei parametri delle attività dell'applicazione), *Application Data Change* (modifiche ai modelli dei dati), *Application Code Refactoring* (modifiche di refactoring nell'AUT).

- **Execution Time Variability:** questa macrocategoria comprende le modifiche che riguardano il tempo di attesa per una completa visualizzazione dell'utente. Nel codice di test questo genere di modifiche si riflettono con la necessità di aggiungere, rimuovere o modificare istruzioni di sospensione (come le istruzioni di *sleep*) tra operazioni e controlli sulle schermate.

- **Compatibility Adaptations:** include tutti gli adattamenti che garantiscono la compatibilità con le diverse versioni del sistema operativo. I problemi di compatibilità, nel caso di applicazioni Android, possono riguardare anche l'aspetto grafico dell'AUT e generalmente si riflettono in creazioni di nuove classi per lo stesso widget per via di deprecazioni varie.

- **GUI Interaction Change:** comprende le modifiche a causa di cambiamenti nelle operazioni eseguite su view e widget esistenti che compongono l'interfaccia utente dell'applicazione. Le sottocategorie incluse sono: *Navigation Change* (modifiche all'ordine di interazione con i widget), *Changed Operations Performed on Views* (modifiche ai modi in cui vengono eseguite determinate operazioni nelle view), *Changed Keyboards/Input Methods* (modifiche riguardanti l'accessibilità della tastiera del software), *Changed Checked Properties* (modifiche alle proprietà verificate nei widget testati), *Changed Way of Accessing Widgets* (modifiche al modo di accesso ai vari widget testati).

- **GUI Views Arrangement:** include le modifiche nel numero e nel tipo di elementi della GUI dell'AUT. Comprende: *View Addition* (aggiunta di view), *View Substitution* (sostituzione di view), *View Removal* (rimozione di view), *Screen Orientation Change* (cambio di orientamento dello schermo in determinate operazioni), *Hierarchy Change* (modifiche nella definizione dei layout delle attività e nella disposizione dei widget).

- **View Identification:** quando tra due versioni successive l'identificativo o il modo di identificare una singola view subisce delle modifiche nel codice di test

si potrebbero invalidare metodi di test funzionanti. Questa macrocategoria comprende tutte questo tipo di modifiche ed include: *Change ID* (modifiche degli ID degli elementi della GUI), *Text Change* (modifiche di elementi di testo che vengono identificati con esso), *Changed Way of Identifying Elements* (cambio di modo di identificare gli widget).

- **Access to Resources:** modifiche dovute al fatto che le risorse vengono utilizzate dai test per effettuare delle verifiche. Il loro identificativo o il luogo dove esse risiedono potrebbero cambiare e questo porterebbe a delle modifiche al codice di test tra versioni successive del software. Questa macrocategoria include: *Changed Retrieval of Text Resources* (modifiche nel modo in cui vengono definite le risorse testuali), *Changed Retrieval of Other Resources* (modifiche nel modo in cui vengono definite le risorse grafiche).

- **Graphic Changes:** comprende tutte le modifiche che riguardano l'aspetto grafico, come ad esempio animazioni e coordinate assolute, dei vari widget della GUI dell'applicazione.

1.2 Qualità del software

L'Ingegneria del Software si è da sempre dedicata al tema della qualità del programma che, per definizione, viene vista come la misura in cui un prodotto soddisfa un certo numero di aspettative sia rispetto al suo funzionamento che alla sua struttura interna.

Una definizione più precisa per quanto riguarda la qualità del codice è stata illustrata da Kothapalli et al. (2011): « La capacità del codice sorgente di soddisfare le esigenze dichiarate e implicite per un dato progetto software ».

In letteratura si sono stabiliti dei parametri rispetto a cui si può misurare o definire la qualità del software. Questi si dividono in due grandi categorie: parametri *esterni*, che si riferiscono a come il programma viene percepito dagli utenti finali, e parametri *interni*, che si riferiscono a come gli sviluppatori percepiscono la qualità del software. In questo lavoro l'interesse è focalizzato sui secondi, in quanto lo scopo finale è strettamente collegato al punto di vista degli sviluppatori e non degli utenti.

I parametri interni possono essere classificati come segue:

- *Verificabilità (Testability):* un software si definisce verificabile se le proprietà di correttezza e affidabilità sono facilmente verificabili, cioè se rivela facilmente i suoi guasti.
- *Manutenibilità (Maintainability):* capacità di un programma di essere modificato. Queste modifiche includono correzioni o adattamenti del sistema a cambiamenti nei requisiti, negli ambienti e nelle specifiche. Include le proprietà di *Riparabilità*, facilità di eliminare difetti e *Evolvibilità*, facilità di modificare il programma in modo da adattarlo a un nuovo ambiente o per migliorarne la qualità.
- *Riusabilità (Reusability):* possibilità di riutilizzare un software nella creazione di un altro programma, in caso con piccole modifiche.
- *Portabilità (Portability):* capacità del sistema di funzionare su piattaforme hardware e software differenti. Si tratta di un parametro che viene facilitato dalla progettazione modulare.
- *Leggibilità (Readability):* un software si definisce leggibile se vi è facilità nella comprensione della lettura del codice e della sua organizzazione ed implementazione.
- *Modularità (Modularity):* utile per misurare da quanti moduli è composto un software. I moduli sono porzioni di codice sorgente contenenti istruzioni scritte per essere riutilizzate più volte nello stesso programma.

In questo studio l'attenzione sarà riservata soprattutto all'attributo *Manutenibilità* in quanto si vuole trovare una relazione tra questa caratteristica del codice e gli effetti che ha nella fragilità dei test.

1.2.1 Manutenibilità del codice

Lo *IEEE Standard Glossary of Software Engineering Terminology* definisce la manutenibilità del software come la facilità con cui un sistema o un componente software può essere modificato per correggere errori, migliorare le prestazioni o altri attributi o adattarsi a un ambiente modificato.

Inoltre, secondo lo standard ISO\IEC 9126, *il software segue l'evoluzione dell'organizzazione*, nel senso che il programma deve adattarsi a tutte le caratteristiche di contorno presenti nello suo sviluppo (ambiente, requisiti, funzionalità).

Sempre secondo lo standard ISO\IEC 9126, la manutenibilità del codice presenta alcuni attributi che ne permettono la descrizione in maniera più completa:

- *Analizzabilità (Analysability):* capacità di poter effettuare la diagnosi sul software ed individuare le cause di errori e di malfunzionamenti.
- *Modificabilità (Changeability):* capacità di consentire lo sviluppo di modifiche al software originale. L'implementazione include modifiche al codice, alla progettazione e alla documentazione.
- *Stabilità (Stability):* capacità di evitare effetti non desiderati a seguito di modifiche al software.
- *Provabilità (Testability):* capacità di consentire la verifica e la validazione del software modificato, in altre parole di eseguire i test.
- *Aderenza alla manutenibilità (Maintainability compliance):* capacità di aderire agli standard e alle convenzioni relative alla manutenibilità.

1.2.2 Modelli per misurare la manutenibilità

L'Ingegneria del Software si è occupata in maniera molto amplia di trovare dei modelli utili per poter misurare la manutenibilità del codice sorgente del software. Lo scopo di questi modelli è di riuscire ad effetturare questa misurazione in base alle modifiche nel codice nelle diverse versioni del programma, così da migliorarne la qualità.

Nel corso degli anni è stato dimostrato come la misurazione e il miglioramento della manutebilità del codice sono molto utili per gestire il *technical debt*, definizione usata per descrivere tutto ciò che riguarda le complicazioni subentranti durante lo sviluppo di un progetto di tipo software (Cunningham, 1992). Inoltre, un altro studio più recente ha evidenziato come l'analisi e la misurazione della manutenibilità del codice sorgente sono tuttora i principali metodi utilizzati per la gestione del *technical debt* (Ernst et al., 2015).

In letteratura sono infatti presenti numerosi *framework* utili per la misurazione della manutenibilità del codice sorgente:

- Aggarwal et al. (2002) hanno proposto un modello basato su tre caratteristiche principali: la leggibilità del codice (RSC), la qualità della documentazione (DOQ) e la comprensibilità del software (UOS). Le misure che vengono

calcolate sono trasformate in valori di tipo *fuzzy*, che verranno elaborati e ritrasformati dagli esperti di dominio.

- Antonellis et al. (2007) è partito dalle caratteristiche dello standard ISO\IEC 9126 per proporre un modello per mappare le metriche orientate all'*object-oriented* in modo da valutare e misurare la manutenibilità di un sistema software. Questo metodo è stato applicato ad un software di tipo OSS dimostrando la possibilità di misurare la manutenibilità del codice attraverso un processo sistematico.

- *SIG Maintainability Model (SIG-MM):* questo modello prevede il collegamento delle caratteristiche di manutenibilità a livello di sistema con le misure a livello di codice in due passaggi. Nel primo passaggio vengono mappate le caratteristiche a livello di sistema su proprietà a livello di codice sorgente. Nel secondo sono determinate una o più misure del codice sorgente per ogni proprietà (Heitlager et al., 2007).

- Un approccio di tipo probabilistico è stato adottato da Bakota et al. (2011) per il calcolo di caratteristiche di alto livello integrando conoscenze specifiche e allo stesso tempo affrontando l'ambiguità. Il valore della manutenibilità del codice viene visto come una distribuzione di probabilità.

- *SQUALE:* questo metodo è basato su degli *Indices*, che rappresentano i costi, per valutare vari aspetti della qualità del codice sorgente. Nel metodo sono presenti due modelli differenti: il *Quality Model* utilizzato per formulare e organizzare i requisiti non funzionali relativi alla qualità del codice e l'*Analysis Model* che contiene sia le regole che vengono utilizzate per normalizzare le misure e le violazioni relative al codice, sia le regole per l'aggregazione dei valori normalizzati (Letouzey, 2012).

- *QUAMOCO:* questo approccio prevede lo sviluppo di un meta modello per la qualità del software che parte dalla strutturazione di concetti legati alla qualità fino alla definizione delle modalità operative per valutarne l'adempimento in un ambiente specifico. Inoltre viene fornito un metodo di valutazione da integrare col meta modello precedente. Questo approccio viene utilizzato per integrare le definizioni di qualità astratte fornite nelle tassonomie di qualità con le tecniche di valutazione e misurazione della qualità concreta del software (Wagner et al., 2012).

- Bauer et al. (2012) ha proposto un approccio alternativo rispetto agli altri analizzati finora, che prevede l'utilizzo di un framework che si adatta alle esigenze di qualità incrementale e di controlli di manutenibilità sul codice sorgente. Questo permette il calcolo incrementale e distribuito di metriche di qualità utili per la valutazione e la misurazione della qualità del software, riuscendo ad includere nei calcoli sia metriche di tipo locale che metriche di tipo globale.

- *Delta Maintainability Model (DMM):* questo modello misura la manutenibilità di una modifica del codice come rapporto tra codice a basso rischio e codice complessivo modificato. Inoltre, identifica i fattori di rischio del codice sorgente riutilizzando le metriche del software e i profili di rischio del SIG-MM, applicando nuove aggregazioni e punteggio per le metriche delta del software a livello di modifiche del codice a grana fine, come commit o pull request, invece di aggregare a livello di sistema (di Biase et al., 2019).

1.3 Strumenti per il miglioramento della qualità del software

La gestione della qualità del software sta diventando un tema di assoluta necessità in quanto i sistemi col passare degli anni si stanno evolvendo in complessità e grandezza. L'utilizzo di programmi o strumenti efficaci per il loro mantenimento risulta fondamentale per gli sviluppatori durante il ciclo di vita del software.

In letteratura sono presenti diversi tipi di strumenti che possono essere utilizzati per migliorare la qualità del software (Krishnan et al., 2007):

- *Static Analysis Tools:* sono utili per esaminare i problemi basati sull'analisi del codice, come per esempio l'utilizzo di variabili non inizializzate, possibilità di perdita di memoria, deferenziazione di puntatori a null.

- *UT Tools:* consentono di eseguire lo *Unit Testing* del codice.

- *Memory Leak Detection Tools:* rilevano possibili perdite di memoria in fase di esecuzione.

- *Code Browsing/Reverse Engineering Tools:* aiutano la comprensione del codice, in modo che miglioramenti e risoluzione dei problemi possano essere applicati in modo appropriato.

- *Profiling Tools:* aiutano a capire e monitorare gli aspetti prestazionali del codice.

- *Coverage Tools:* evidenziano quali parti del codice è coperto dai casi di test eseguiti per aiutare a garantire la qualità dei test.

1.3.1 Esempi di strumenti per il monitoraggio della qualità

Di seguito vengono presentati alcuni strumenti di esempio utili per migliorare e monitorare in maniera costante la qualità del software durante il suo sviluppo:

- *Scalable and Flexible Error Detection (SAFE)*: è uno strumento di tipo *Static Analysis Tools* che viene utilizzato per la verifica e la ricerca di bug in piccoli software scritti in linguaggio Java (Geay et al., 2006).

  Lo strumento, sviluppato da IBM, utilizza due tipi di analisi: un semplice controllo strutturale basato sul *pattern-matching* e un risolutore di flussi di dati sensibili al flusso interprocedurale che integra il controllo dei tipi e l'analisi degli alias.

  SAFE è progettato per funzionare in modalità *batch*, cioè come parte integrante di un loop continuo di tipo *build|integration* o come componente interattivo in un ambiente di sviluppo Eclipse.

  Lo strumento si basa su due componenti di analisi principali che permettono all'utente di personalizzare regole e proprietà in base a costrutti specifici del software:

  - *Structural Engine:* si basa principalmente sulla corrispondenza di pattern intraprocedurali. Questo componente cerca di associare i pattern del codice riconosciuti.
  - *Typestate Checking Engine:* esegue il controllo interprocedurale dello stato del tipo con diversi gradi di costo e precisione, a seconda del modo in cui viene gestito l'aliasing.

SAFE è stato progettato per permettere agli utenti di eseguire analisi relativamente costose interattivamente e può essere utilizzato in tanti contesti diversi:

  - All'interno di CruiseControl, SAFE si adatta benissimo trattandosi di un ambiente di tipo *continuous integration*. Lo strumento funziona come parte normale del loop di costruzione CruiseControl e pubblica i risultati in una scheda HTML all'interno del framework. SAFE può essere installato all'interno di CruiseControl senza creare sovraccarico di compilazione per i singoli sviluppatori.
  - Come plugin di Eclipse per fornire un'analisi statica all'interno dell'IDE. In questo modo lo sviluppatore può analizzare il codice sorgente per trovare problemi prima di effettuare commit nella repository del suo software. Inoltre, Eclipse fornisce una più ricca ed interattiva interfaccia grafica per l'utente.
  - Come task di ANT, dove l'utente fornisce lo scopo dell'analisi e le opzioni come file XML di configurazione e SAFE produce i suoi risultati in un file XML di output. L'esplorazione dei risultati può essere effettuata attraverso un'interfaccia utente web fornita, utilizzando Extended Style Sheets e script Javascript.
  - Come strumento stand-alone da linea di comando che riceverà le opzioni ed eventuali altri parametri da linea di comando da parte dell'utente. Anche in questo contesto i risultati possono essere esplorati attraverso l'interfaccia utente fornita dal programma.
  - Come plugin di Eclipse ma senza l'utilizzo dell'IDE stesso, SAFE può essere utilizzato come un'applicazione di tipo Eclipse Rich Client Platform (RCP), senza interfaccia utente, ma con il modello di dati Eclipse.

- *CodeArena*: CodeArena[5] è un'estensione del gioco Minecraft proposto da Baars and Meester (2019) e utilizzato per rendere intuitivi il campo della qualità del codice e della manutenzione per gli sviluppatori software. Infatti

[5]https://github.com/SimonBaars/CodeArena

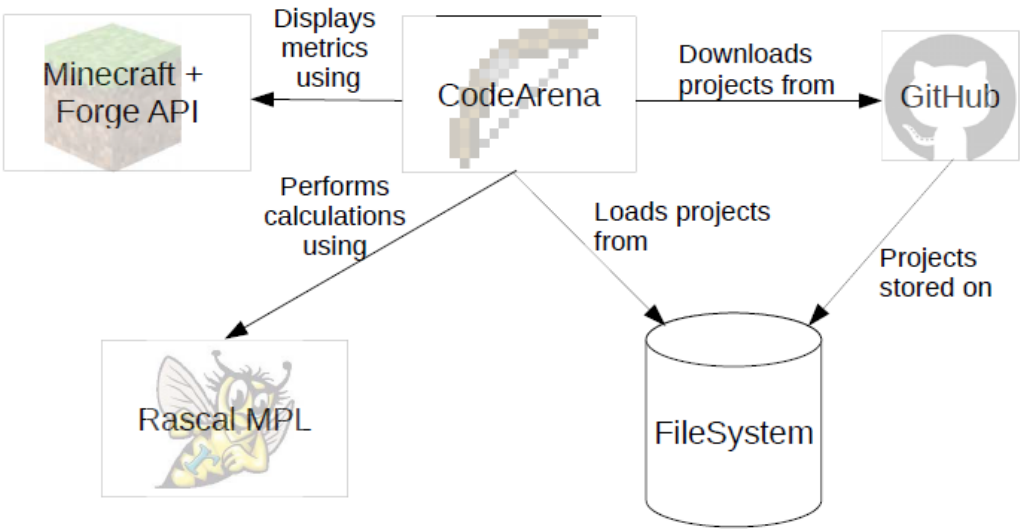


Figura 1.1. Architettura di CodeArena.

questo strumento ha come scopo ridurre al minimo il *technical debt* e aumentare la manutenibilità del software, ottenendo una visione progressiva delle cause di difficoltà nella manutenibilità del codice.

CodeArena converte i modelli nel codice sorgente che vengono considerati pericolosi o dannosi in mostri di Minecraft. La creazione di questi mostri avviene soprattutto per violazioni delle linee guida sulla qualità trovate nel codice sorgente. Questi pericoli possono essere combattuti in maniera graduale permettendo all'utente di ricondursi alla locazione del problema nel codice. Se lo sviluppatore riesce a risolvere il problema, il mostro morirà e lo sviluppatore verrà ricompensato nel gioco. La presenza di diversi tipi di mostri permette all'utente di capire anche quale tipo di problema si presenta, per esempio in caso di presenza di ragni si ha un evidente problema di refactoring.

L'architettura di CodeArena è illustrata nella Figura 1.1. Lo strumento è stato realizzato utilizzando un'API che permette di estendere il gioco attraverso codice Java, Minecraft Forge. Una volta che l'utente sceglie il progetto, verrà creato un *Abstract Syntax Tree (AST)* basato sul codice sorgente e sul quale verranno effettuati i calcoli necessari all'analisi. I calcoli vengono eseguiti utilizzando Rascal, un meta-linguaggio di programmazione molto propenso

all'analisi e alla trasformazione del codice sorgente. Gli sviluppatori di CodeArena hanno deciso di utilizzare un thread separato per l'algoritmo per i vari calcoli, in modo tale da far mantenere all'utente l'utilizzo del gioco e per fornire un feedback diretto all'utente durante la scansione dei possibili problemi. Quando viene corretto un bug viene immediatamente rieseguita la scansione del codice. I punti che si guadagnano correggendo il proprio codice vengono aggiunti ad un tabellone segnapunti nel display dell'utente. Questo strumento è un esempio di come si potrebbero educare gli sviluppatori software utilizzando la *gamification*. Si tratta di un metodo nuovo che permette di acquisire familiarità con il campo del *technical debt* e dei concetti correlati e aumenta la comprensione immergendo lo sviluppatore nei problemi riscontrati nel codice sorgente del programma.

- *CompareCFG*: CompareCFG è uno strumento proposto da Jiang et al. (2020) che fornisce un feedback basato sul grafico del flusso di controllo (CFG) generando il confronto tra il CFG del codice sorgente e altri CFG meno complessi, in modo da favorire la riduzione della complessità del codice. Inoltre, CompareCFG identifica gli indicatori di stile semantico fornendo degli esempi specifici in cui il codice può essere migliorato. Questo strumento è stato utilizzato per aiutare gli studenti a produrre codice sorgente di più alta qualità possibile, attraverso l'utilizzo di feedback visivi automatizzati.

  Il grafico del flusso di controllo (CFG) è un grafo diretto dove i vertici rappresentano gli stati. Un collegamento diretto tra due vertici esiste se l'esecuzione può procedere dagli stati rappresentati dal vertice della coda alle affermazioni rappresentate dal vertice della testa. Inoltre, un CFG può essere rappresentato in diverse maniere, come per esempio un vertice può rappresentare una singola istruzione di codice o direttamente blocchi base di istruzioni. Un esempio di trasformazione di un blocco di codice in un CFG è illustrato nella figura 1.2. Per questo lavoro la scelta è ricaduta sugli CFG in quanto forniscono un'interfaccia intuitiva e una visualizzazione grafica accattivante del codice sorgente. Inoltre, la differenza nella complessità di due implementazioni con la stessa funzionalità spesso può essere determinata semplicemente visualizzando i rispettivi grafici del flusso di controllo *side-by-side*, funzione principale di CompareCFG.

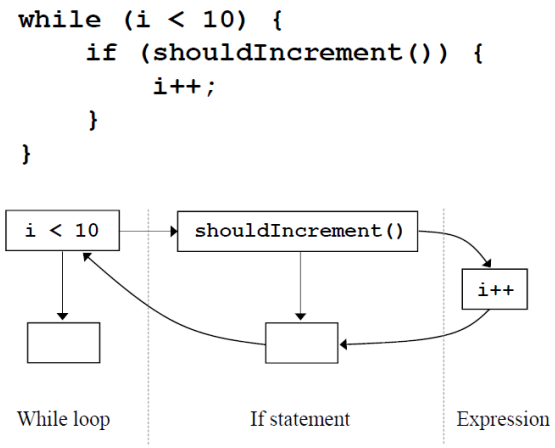


Figura 1.2. Un frammento di codice e il suo CFG corrispondente.

Un'altra caratteristica di questo strumento riguarda gli indicatori di stile semantico: si tratta di caratteristiche a livello di istruzione che possono essere più o meno ideali in alcune circostanze. Sono stati identificati 16 indicatori diversi che si verificavano più spesso nei codici sorgenti. Inolte, è stato notato come molti degli indicatori proposti corrispondono ad un CFG confusionario o eccessivamente complesso. Un'altra funzione di CompareCFG è appunto quella di fornire dei feedback anche in base a questi indicatori di stile semantico.

Lo strumento include una semplice interfaccia grafica basata sul web utile per visualizzare il feedback generato. L'analisi del codice viene eseguita dal framework JavaParser, di conseguenza CompareCFG attualmente supporta solo codice sorgente scritto in Java. Il framework viene utilizzato per la creazione di un Abstract Syntax Tree (AST) che viene attraversato per la creazione del grafico del flusso di controllo. Inoltre sono state prese delle decisioni per quanto riguarda la mappatura, in quanto non esistono convenzioni standard per mappare codice Java: la mappatura deve essere fatta in modo tale da garantire che il CFG risulti riconoscibile al codice sorgente pur mantenendo la visibilità dei problemi presenti.

1.4 Obiettivi

Lo scopo di questo lavoro di tesi è trovare una possibile correlazione tra la fragilità delle suite di test e la qualità del codice sorgente, interessandosi soprattutto sulla manutenibilità di quest'ultimo.

L'analisi si baserà sull'utilizzo di metriche statiche, cioè che non prevedono che venga eseguito il software per il loro calcolo, che permettono di misurare sia la fragilità dei test che la manutenibilità del software. Queste metriche permettono di avere un quadro generale sull'evoluzione del programma da parte degli sviluppatori in modo da tracciare un'analisi di correlazione valida lungo le diverse versioni del codice.

Il lavoro verrà svolto studiando il codice sorgente di un software in tutte le sue versioni rilasciate, le metriche verranno calcolate per ogni versione e verrà calcolata una correlazione tra i due fattori di studio in modo tale da determinare una legge che permetta di legare la qualità del software alla fragilità dei test.

La parte restante di questo lavoro di tesi è strutturata come segue:

- Il Capitolo 2 tratta il processo di sviluppo dello script che analizza le metriche di fragilità dei test e di qualità del codice in modo da verificarne una legge di correlazione;
- Il Capitolo 3 presenta la descrizione dell'esperimento di questo lavoro di tesi e i risultati dell'utilizzo dello strumento in un set di applicazioni mobile di tipo Android;
- Il Capitolo 4 illustra le conclusioni e i suggerimenti per lavori futuri correlati.

Capitolo 2

Architettura e design

Lo script sviluppato per questo lavoro di tesi permette il calcolo di due tipologie differenti di metriche di tipo software per applicazioni di tipo Android scritte in linguaggio Java o in Kotlin. Lo strumento è scritto in Python e prevede lo sviluppo finale di una legge di correlazione tra:

- le metriche di fragilità delle suite di test che permettono di individuare quali classi e quali metodi risultano più fragili e di conseguenza devono essere modificati più volte durante lo sviluppo del software in tutte le sue versioni rilasciate.
- le metriche di qualità del software, più precisamente quelle di manutenibilità, che permettono agli sviluppatori di controllare e mantenere continuamente la qualità e la correttezza del proprio programma.

La Figura 2.1 illustra la struttura modulare dello script che riceve come input:

- un file, *sonar-qube.properties*, indispensabile per gestire la configurazione e l'analisi del progetto mediante il software SonarQube, fondamentale nel calcolo delle metriche di qualità come verrà illustrato nel Capitolo 2.1.4.
- l'indirizzo url di GitHub del progetto da analizzare che verrà copiato in locale per effettuare l'analisi delle metriche.

Inoltre lo script può essere suddiviso in tre blocchi principali:

1. Il primo blocco riguarda la copia del progetto GitHub in locale e l'aggiornamento di essa ad ogni iterazione dello script in avanzamento con le varie

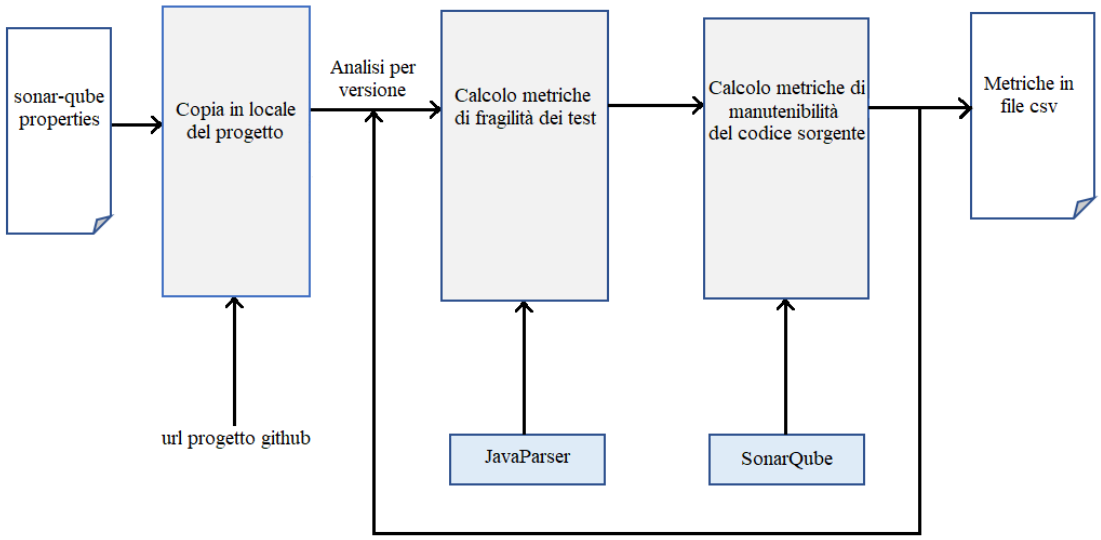


Figura 2.1. Rappresentazione modulare dello script sviluppato.

versioni rilasciate del software. Alla prima iterazione infatti avviene la clonazione dell'intero progetto nelle prime due versioni. Nelle successive iterazioni avviene invece l'aggiornamento delle repository alle due versioni successive. La necessità di avere a disposizione in locale due versioni consecutive del progetto, è dovuta al fatto che diverse metriche vengono calcolate da fattori che si basano sulle differenze tra due versioni consecutive del software e della sua suite di test.

2. Il secondo blocco si occupa del calcolo delle metriche di fragilità della suite di test del programma. Vengono dunque analizzati più nello specifico tutte le classi di test, i loro metodi e le modifiche che vengono effettuate nelle versioni in esame. In questa parte dello script è stato necessario utilizzare *JavaParser*, soprattutto per l'analisi mirata sui metodi delle classi di test scritte in Java.

3. Il terzo blocco infine si occupa del calcolo delle metriche di qualità del codice sorgente del programma. Queste riguardano la dimensione, la complessità e la manutenibilità del progetto software che deve essere analizzato. Per questo scopo è stato utilizzato il software SonarQube che permette facilmente il calcolo di diverse metriche di questa parte dello script.

In conclusione, lo strumento rilascia come output due file di tipo CSV con le diverse metriche calcolate per ogni versione rilasciata del software.

2.1 Strumenti e librerie utilizzate

All'interno dello script è stato necessario utilizzare degli strumenti che hanno permesso di calcolare le metriche, secondo la loro definizione, in maniera più efficiente e veloce. Si tratta di tool di tipo Open-Source, facilmente scaricabili e con un'ampia e consultabile documentazione online.

Sebbene la diffusione di strumenti per le applicazioni Android scritte in Java sia notevole, ciò non si ripete per il più nuovo linguaggio Kotlin. Per esempio, per questo linguaggio al momento non esiste un tool di analisi sintattica, più precisamente chiamato *parser*, che invece è facilmente identificabile in JavaParser per Java.

Inoltre per il calcolo delle metriche di qualità e manutenibilità del codice sorgente, gli unici software presenti sul mercato che supportano il linguaggio Kotlin sono di tipo Closed-Source. Le uniche due eccezioni sono: SonarQube un programma che cura costantemente la qualità e la sicurezza del codice sorgente basato su una piattaforma di tipo OSS scaricabile gratuitamente dal sito ufficiale; e CBR Insight, uno strumento Open-Source utile per il calcolo di diverse metriche ma costruito sulla base di Understand e ne necessita una licenza senza la quale non è possibile utilizzare il software di tipo OSS.

Questa importante limitazione è dovuta al fatto che Kotlin è un linguaggio di programmazione molto recente (2011) e a breve gli strumenti e gli sviluppatori software supporteranno più adeguatamente anche questo nuovo linguaggio. Infatti soltanto a partire da Maggio del 2019, Google indica Kotlin come il linguaggio consigliato per lo sviluppo di un'applicazione di tipo Android.

In questo studio come software per il calcolo di metriche è stato deciso di utilizzare solamente stumenti Open-Source, di conseguenza l'unica possibilità è stata quella di utilizzare SonarQube per il calcolo di metriche di manutenibilità e sicurezza del codice sorgente.

2.1.1 GitPython

GitPython[1] è una libreria Python di alto livello che viene utilizzata per interagire con le repository git. Questo tipo di interazioni possono avvenire sia in locale, dopo aver clonato un progetto, che in remoto, direttamente sul server GitHub. In questo lavoro di tesi, questa libreria è stata utilizzata nello script ogni qual volta è stato necessario consultare GitHub per recepire informazioni sul progetto in analisi.

GitPython mette a disposizione diverse Application Programming Interface (API), librerie che permettono una facile interazione tra software differenti, utilizzabili direttamente all'interno del codice scritto in Python. Infatti, qualsiasi operazione disponibile in git è riproducibile attraverso le API con una semplice linea di codice e inoltre con una notevole facilità di analisi dei dati ottenuti.

Le API usate nello specifico riguardano la clonazione iniziale o l'aggiornamento del progetto ad ogni iterazione dello script attraverso la *git.repo.clone_from()* e la *git.repo.git.checkout()*, l'elenco delle versioni rilasciate nel progetto con la *git.repo.git.tag()* e diverse altre operazioni come quelle di differenza, *git.repo.git.diff()* o di ricerca, *git.repo.git.grep()*.

2.1.2 Cloc

Cloc[2] è un software di tipo Open-Source che permette il calcolo di righe vuote, righe di commento e righe fisiche del codice sorgente in molti linguaggi di programmazione.

Cloc è interamente scritto in Perl e il suo codice è completamente consultabile su GitHub. Il programma ha diverse caratteristiche che lo rendono completo, facile da usare e portabile:

- è facilmente scaricabile ed eseguibile su qualsiasi sistema operativo
- può lavorare su linguaggi di programmazione non ancora del tutto sviluppati in quanto può leggere le definizioni dei commenti da file
- selezionando una cartella consente la somma di più file di progetto anche in diversi linguaggi di programmazione e anche se contenuti all'interno di archivi compressi

[1] https://gitpython.readthedocs.io/en/stable/

[2] https://github.com/AlDanial/cloc

- i risultati possono essere prodotti in diversi formati, tra cui SQL, JSON, XML e YAML

All'interno dello script è stato utilizzato per il conteggio delle linee fisiche e delle linee di commento del codice sorgente per ogni versione del software in esame.

2.1.3 JavaParser

JavaParser[3] è un insieme di strumenti utile per analizzare, trasformare e generare codice Java. Inoltre, consente di interagire con il codice sorgente Java come rappresentazione di un oggetto in un ambiente Java.

Questo strumento rientra nella categoria dei *parser*: si tratta di un programma che esegue un processo analizzatore di un flusso continuo di dati in input in modo da determinare la correttezza della sua struttura sintattica grazie ad una data grammatica formale.

Inoltre, per la rappresentazione di un oggetto Java viene costruito un Abstract Syntax Tree (AST): si tratta di una rappresentazione ad albero della struttura sintattica astratta del codice sorgente, dove ogni nodo dell'albero denota un costrutto che si verifica nel codice. Si parla di sintassi di tipo astratta in quanto non rappresenta ogni dettaglio che appare nella sintassi reale, ma solo i dettagli strutturali o relativi al contenuto.

JavaParser include anche una libreria chiamata *JavaSymbolSolver* che viene utilizzata per risolvere i simboli in AST per recuperare il loro tipo e la loro dichiarazione a partire da un dato riferimento, oppure per calcolare il tipo di risultato di un'espressione e per molte altre operazioni correlate all'AST. Tra le caratteristiche principali troviamo:

- **Supporto per la generazione di codice:** JavaParser può essere utilizzato per evitare di scrivere codice ripetuto molte volte all'interno del programma, per creare *source-to-source compiler*, DSL e tanto altro. Questo viene fatto molto facilmente grazie ad alcuni metodi specifici che possono essere applicati ad un oggetto di tipo *CompilationUnit* che potrà essere successivamente trasformato in codice attraverso il metodo *toString()*.

[3]https://javaparser.org/

- **Supporto per l'analisi di codice:** più nello specisico JavaParser fornisce supporto per controllare la qualità attraverso alcuni standard, per il calcolo di metriche di codice e infine per eseguire alcune semplici *query* per prendere più familiarità col codice stesso.

- **Supporto per il refactoring del codice:** funzionalità molto utile quando si deve modernizzare una grande quantità di codice sorgente, o per aggiornare le dipendenze o per modificare i pattern di utilizzo.

In questo lavoro di tesi, JavaParser è stato utilizzato per l'analisi e i conteggi vari sui metodi presenti nei file di codice sorgente in Java, al fine di calcolare alcune metriche di fragilità delle suite di test.

2.1.4 SonarQube

SonarQube[4] è una piattaforma di tipo Open-Source sviluppata da SonarSource per l'ispezione continua della qualità del codice e per eseguire revisioni automatiche con analisi statica del codice per il rilevamento di bug, *code smells* e vulnerabilità di sicurezza, su oltre 20 linguaggi di programmazione.

SonarQube inoltre permette la registrazione della cronologia delle metriche statiche calcolate e fornisce dei grafici di evoluzione del progetto nelle varie analisi che vengono effettuate.

Il software è distribuito in diverse edizioni:

- *Community Edition*: il punto di partenza per analizzare in maniera continua l'analisi del codice sorgente del proprio software. Questa versione comprende l'analisi statica per 15 linguaggi di programmazione, Python, Kotlin e Java compresi, il rilevamento di bug e violazioni della sicurezza nel codice e il calcolo e la relativa storia di metriche di manutenibilità e sicurezza del software. Si tratta dell'unica edizione Open-Source ed è quella che è stata utilizzata nello script di questa tesi.
- *Developer Edition*: rispetto alla precedente edizione, viene garantita una sicurezza più elevata del proprio software. Include il supporto ad altri 7 linguaggi di programmazione aggiuntivi, tra cui i più comuni C, C++. Inoltre, comprende il rilevamento di tentativi di *code injection* in diversi linguaggi di

[4]https://www.sonarqube.org/

programmazione e la manutenzione e l'analisi dei vari branches del proprio progetto. Supporta anche interazione con GitHub, BitBucket, Azure DevOps e GitLab.

- *Enterprise Edition*: oltre a tutte le caratteristiche delle precedenti versioni, l'Enterprise Edition abilita l'analisi della qualità e della sicurezza del codice sorgente a livello aziendale, con il supporto ad altri 5 linguaggi di programmazione come COBOL, Apex e RPG.
- *Data Center Edition*: l'ultima versione di SonarQube comprende una più alta disponibilità, adatta per distribuzioni di tipo globale. Come caratteristiche aggiuntive si hanno la ridondanza dei componenti, la resilienza dei dati e la scalabilità di tipo orizzontale.

2.2 Metriche di fragilità delle suite di test

Tabella 2.1. Definizione delle metriche di fragilità dei test.

Gruppo	Nome	Spiegazione
Diffusion and Size	NTR	Number of Tagged Releases
	NTC	Number of Test Classes
	TTL	Total Test LOCs
	TLR	Total LOCs Ratio
Evolution	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
	MRR	Modified Releases Ratio
	TSV	Test Suite Volatility

Per quanto riguarda la fragilità dei test, in questo lavoro di tesi si è deciso di scegliere delle metriche che sono state già fornite e menzionate in diversi lavori nella letteratura dell'Ingegneria del Software.

Per questa parte è stato scelto un insieme di 11 metriche, riprese dall'articolo di Coppola et al. (2019). La Tabella 2.1 illustra le metriche selezionate e le relative

descrizioni. Si tratta di metriche normalizzate che permettono il confronto tra progetti di dimensioni differenti. Possono essere divise in due gruppi principali:

- *Diffusion and Size metrics*, che caratterizzano la quantità di codice di test e la dimensione dei progetti analizzati.
- *Evolution metrics*, che forniscono informazioni sull'evoluzione delle suite di test durante il ciclo di vita del progetto.

2.2.1 Diffusion and Size

Questo gruppo di metriche permette di stimare caratteristiche del progetto analizzato che riguardano la quantità e la dimensione delle suite di test. Sono state definite 4 metriche:

- **Number of Tagged Releases (NTR):** si tratta del numero di versioni rilasciate di un progetto Android, che solitamente vengono elencate utilizzando il comando *git tag* nella repository git. In questo studio per ricavare questa metrica è stata utilizzata l'equivalente API di GitPython *git.repo.git.tag()*. Questa metrica può essere utilizzata per capire qual è la natura delle applicazioni che vengono più probabilmente testate col testing di tipo automatico.

- **Number of Test Classes (NTC):** è il numero di classi riguardante il codice di test presenti in una versione di un progetto di tipo Android.

- **Total Test LOCs (TTL):** è il numero di linee fisiche di codice che possono essere attribuite alle classi di test in una versione di un progetto Android.

- **Total LOCs Ratio (TLR):** è definita come

$$TLR_i = \frac{TTL_i}{Plocsi_i}, \tag{2.1}$$

  dove *Plocsi* è la quantità totale di LOC di un progetto Android nella versione *i*. Nel *Plocsi* vengono incluse le linee di codice riguardanti il software e le suite di test. Questa metrica si presenta con un valore compreso nell'intervallo tra [0-1] e consente di quantificare la rilevanza del codice di test rispetto all'intero quantitativo di linee di codice dell'intero progetto.

2.2.2 Evolution

Il secondo gruppo di metriche permettono di descrivere l'evoluzione dei progetti Android e delle loro relative suite di test. Queste metriche vengono calcolate per ogni coppia di versioni rilasciate in maniera consecutiva. Sono state definite 7 metriche:

- **Modified Test LOCs Ratio (MTLR):** è definita come

$$MTLR_i = \frac{Tdiffi_i}{TTL_{i-1}}, \tag{2.2}$$

  dove $Tdiffi_i$ è la quantita di LOC modificati, aggiunti o cancellati associati alle classi di test, nella transizione tra le versioni *i - 1* e *i* e TTL_{i-1} è la quantità totale di LOC riguardanti le classi di test nella versione *i - 1*. Questa metrica viene definita solo quando $TTL_{i-1} > 0$, cioè solo la versione precedente è fornita di codice di test. Inoltre, essa quantifica la quantità di modifiche eseguite sui LOC del codice di test esistente per una versione specifica di un progetto.

- **Modified Relative Test LOCs (MRTL):** è definita come

$$MRTL_i = \frac{Tdiffi_i}{Pdiffi_i}, \tag{2.3}$$

  dove $Tdiffi_i$ è la quantita di LOC modificati, aggiunti o cancellati associati alle classi di test, nella transizione tra le versioni *i - 1* e *i* e $Pdiffi_i$ è la quantita di LOC modificati, aggiunti o cancellati riguardante l'intero progetto, classi di test comprese, nella transizione tra le versioni *i - 1* e *i*. Questa metrica viene definita quando $Pdiffi_i > 0$ e viene calcolata solo nel caso in cui $TRL_i > 0$. Si presenta con un valore compreso nell'intervallo tra [0-1] e valori vicini a 1 implicano che una parte significativa del lavoro totale nella realizzazione dell'evoluzione dell'applicazione è necessaria per mantenere aggiornati i test nelle varie versioni.

- **Modified test Classes Ratio (MCR):** è definita come

$$MCR_i = \frac{MC_i}{NTC_{i-1}}, \tag{2.4}$$

dove MC_i è il numero di classi di test che vengono modificate nella transizione tra la versione *i - 1* e *i* e NTC_{i-1} è il numero di classi di test relativo alla versione *i - 1*. Questa metrica non è definita quando $NTC_{i-1} = 0$ e si presenta con un valore compreso nell'intervallo tra [0-1]. Inoltre, maggiori sono i valori di *MCR*, meno le classi di test sono stabili durante l'evoluzione dell'applicazione Android.

- **Modified test Methods Ratio (MMR):** è definita come

$$MMR_i = \frac{MM_i}{TM_{i-1}}, \tag{2.5}$$

dove MM_i il numero di metodi delle classi di test che vengono modificati tra le versioni *i - 1* e *i* e TM_{i-1} è il numero totale di metodi delle classi di test associati alla versione *i - 1*. Questa metrica non è definita quando $TM_{i-1} = 0$ e si presenta con un valore compreso nell'intervallo tra [0-1]. Infine, maggiori sono i valori di *MMR*, meno i metodi delle classi di test sono stabili durante l'evoluzione dell'applicazione che testano.

- **Modified Classes with Modified Methods Ratio (MCMMR):** è definita come

$$MCMMR_i = \frac{MCMM_i}{NTC_{i-1}}, \tag{2.6}$$

dove $MCMM_i$ è il numero di classi di test che vengono modificate e che includono almeno un metodo modificato tra le versioni *i - 1* e *i* e NTC_{i-1} è il numero di classi di test relativo alla versione *i - 1*. Questa metrica non è definita quando $NTC_{i-1} = 0$ ed è limitata superiormente da *MCR*, poiché per definizione $MCR_i = MC_i/TC_i$, e di conseguenza $MCMM_i \leq MC_i$.

- **Modified Releases Ratio (MRR):** è definita come

$$MRR = \frac{MTCTR}{NTR}, \tag{2.7}$$

dove *MTCTR* è il numero di versioni rilasciate in cui è stata modificata almeno una classe di test e *NTR* è il numero di versioni rilasciate dell'applicazione. Questa metrica si presenta con un valore compreso nell'intervallo tra [0-1] e valori maggiori di *MRR* indicano un valore minore dell'adattabilità della suite di test relativa ai cambiamenti nell'applicazione Android.

- **Test Suite Volatility (TSV):** è definita per ogni progetto come il rapporto tra il numero di classi di test che vengono modificate almeno una volta nella loro esistenza e il numero totale di classi di test della storia del progetto.

2.3 Metriche di manutenibilità del codice sorgente

Tabella 2.2. Definizione delle metriche di manutenibilità del codice sorgente.

Gruppo	Nome	Spiegazione
Complexity	CC	McCabe's Cyclomatic Complexity
Size	CLOC	Comment Lines of Code
	LOC	Lines of Code
	NOC	Number of Classes
	NOF	Number of Files
	NOM	Number of Methods
	STAT	Number of Statements
Stability	CHANGE	Number of Lines Changed in the Class
	CODE SMELLS	Number of Code Smells
	TD	Technical Debt
	TDR	Technical Debt Ratio

Anche per misurare la qualità del codice sorgente di un progetto GitHub la scelta è ricaduta su metriche già definite e menzionate da diversi autori nella letteratura dell'Ingegneria del Software.

Alcune metriche sono state riprese dall'articolo di Ardito, Coppola, Barbato, and Verga (2020) che nello studio citato sono state identificate tra le più menzionate negli articoli in letteratura riguardanti questo argomento, altre invece sono state riprese direttamente dalla documentazione di *SonarQube*. La Tabella 2.2 espone le 11 metriche selezionate e le relative descrizioni, divise in base al gruppo di appartenenza. Si possono facilmente individuare infatti, tre gruppi di appartenenza differenti:

- *Complexity metrics*, che indicano alcune caratteristiche sulla complessità del software sotto esame.

- *Size metrics*, che si basano sulla dimensione del progetto e sulla quantità di codice sorgente del software analizzato.
- *Stability metrics*, che forniscono informazioni su quanto il progetto analizzato risulta essere stabile.

2.3.1 Complexity

Questa tipologia di metriche è molto utile agli sviluppatori in quanto indica il livello di complessità del codice sorgente del software. La complessità è strettamente legata alla manutenibilità del programma, in quanto per definizione un codice più complesso è meno mantenibile e meno stabile. In questo studio, l'unica metrica di complessità scelta è la seguente:

- **McCabe's Cyclomatic Complexity (CC):** è una metrica sviluppata da McCabe nel 1976 che in principio intendeva calcolare la complessità del codice esaminando il grafico del flusso di controllo del programma, ovvero contando i suoi percorsi di esecuzione indipendenti basati su il diagramma di flusso. Il presupposto è che la complessità del codice è correlata al numero dei percorsi di esecuzione del suo diagramma di flusso. È anche dimostrato che esiste una correlazione lineare tra le metriche CC e LOC. Tale relazione è inoltre indipendente dal linguaggio di programmazione o da paradigmi di codice che vengono utilizzati nella stesura del programma. Ogni nodo nel grafico di flusso corrisponde a un blocco di codice nel programma in cui il flusso è sequenziale e gli archi corrispondono a rami che possono essere presi dal flusso di controllo durante l'esecuzione del programma. Sulla base di questi elementi, la CC di un codice sorgente è definito come $M = en + 2p$ dove e è il numero di bordi del grafico, n è il numero di nodi del grafo e p è il numero di componenti collegati, cioè il numero di uscite dalla logica di programma.

2.3.2 Size

L'ultima tipologia di metriche di qualità del codice riguarda la dimensione del progetto e la quantità di codice che viene scritto durante l'evoluzione del software. Sono state definite 6 metriche:

- **Comment Lines of Code (CLOC):** si tratta della metrica che fornisce il numero di righe di codice contenenti commenti testuali. Righe vuote di

commenti non vengono conteggiate. In contrasto con il LOC, maggiore è il valore restituito da CLOC, più commenti ci sono nel codice analizzato; pertanto, il codice dovrebbe essere più facile da capire e più mantenibile. In letteratura è stata anche proposta una metrica che mette in relazione CLOC e LOC, e si chiama *Code-to-Comment Ratio*.

- **Lines of Code (LOC):** si tratta di una metrica ampiamente utilizzata che viene spesso utilizzata per la sua semplicità. Infatti essa dà una misura immediata della dimensione del codice sorgente. Tra le metriche più popolari, la metrica LOC è stata l'unica che non sembra avere una definizione singola e universalmente adottata di come questa metrica viene calcolata. Alcuni lavori considerano il conteggio di tutte le righe in un file e altre (la maggior parte) rimuovono le righe vuote da tale calcolo; se c'è più di un'istruzione in una singola riga oppure una singola istruzione è divisa in differenti righe, vi è ambiguità nel considerare il numero di linee (righe fisiche) o l'effettivo numero di istruzioni coinvolte (linee logiche). Così, è della massima importanza che gli strumenti per calcolare le metriche specifichino esattamente come calcolare i valori che restituiscono (o che sono Open-Source, consentendo quindi un'analisi dello strumento codice sorgente per ricavare tali informazioni). Sebbene la metrica LOC sembra essere scarsamente correlata al lavoro di manutenzione e c'è più di uno modo per calcolarla, questa metrica viene utilizzata all'interno dell'indice di manutenibilità e sembra essere correlata con molte misure metriche differenti. L'ipotesi è che maggiore è la metrica LOC, il meno mantenibile è il codice analizzato.

- **Number of Classes (NOC):** questa metrica è definita come il numero di classi presenti nell'intero progetto. Sono incluse eventuali classi annidate, interfacce, enumerazioni e annotazioni.

- **Number of Files (NOF):** si tratta di una metrica che illustra il numero di file presenti all'interno del progetto durante il ciclo di vita del software.

- **Number of Methods (NOM):** questa metrica illustra il numero di metodi in una data classe/file sorgente, con l'ipotesi che maggiore è il numero di metodi, minore è la manutenibilità del codice.

- **Number of Statements (STAT):** questa metrica conta il numero di istruzioni in un metodo. Diverse variazioni della metrica sono state proposte nella

letteratura, che differiscono sulla decisione di conteggio istruzioni anche in inner class nominate, interfacce, e inner class anonime. Ad esempio, Kaur et al., nel loro studio sulla manutenibilità del software previsione, conta il numero di dichiarazioni solo in classi interne anonime.

2.3.3 Stability

La stabilità è una capacità fondamentale di un software. Questa tipologia di metriche serve per individuare quanto il programma risulti stabile e di conseguenza mantenibile, cioè adatto a subire modifiche o miglioramenti a seguito di cambiamenti dell'ambiente, dei requisiti o delle specifiche funzionali. Di seguito vengono definite 4 metriche di stabilità del codice:

- **Number of Lines Changed in the Class (CHANGE):** si tratta di una metrica di cambiamento, che misura quante righe di codice vengono modificate tra due versioni della stessa classe di codice. Questa metrica quindi non è definita su un'unica versione del progetto software, ma è su misura per analizzare l'evoluzione del codice sorgente per ogni classe del progetto in esame. Il presupposto tra l'utilizzo di questa metrica è che se una classe viene modificata continuamente, può essere un segno che è difficilmente mantenibile. In genere, è possibile apportare tre tipi di modifiche ad una riga di codice: aggiunte, eliminazioni o modifiche. In letteratura, c'è tipicamente accordo su come contare le operazioni di modifica, che in genere vengono contate doppie, in quanto la modifica viene considerata come una cancellazione seguita da un'aggiunta. La maggior parte dei tempi, commenti e spazi vuoti non sono considerati nel calcolo delle LOC modificate durante l'evoluzione del codice sorgente di un software.

- **Number of Code Smells (CODE SMELLS):** si tratta di una metrica che illustra il numero di *code smells* presenti nel codice sorgente dell'intero progetto software. L'espressione *code smell* viene utilizzata nel campo dell'informatica per identificare una serie di caratteristiche che possono essere presenti nel codice sorgente che sono solitamente sono riconosciute come probabili indicazioni di un difetto di programmazione. Questa definizione non è da confondere con la descrizione di *bug*, infatti non si tratta di veri e proprio bachi, bensì di debolezze di progettazione che riducono la qualità del software.

- **Technical Debt (TD):** si tratta di una metrica composta, proposta come metodo per valutare la stabilità e la manutenibilità di un sistema software. Ci sono diverse definizioni di questa metrica, che è stata introdotta per la prima volta nell'articolo di Cunningham (1992). Strecansky et al. (2020) illustrano tre diverse vie per il calcolo del Technical Debt: la prima utilizza il MI (Maintainability Index), una metrica composta che prevede l'utilizzo a sua volta di tre metriche, Halstead Volume (HV), cyclomatic complexity (CC) e il numero di righe di codice (LOC) e che ha ricevuto pareri contrastanti in letteratura sulla sua efficacia (Ostberg and Wagner (2014) e Sarwar et al. (2008)); la seconda calcola il TD attraverso i modelli SIG-TD che si basano su cinque proprietà del codice sorgente del software (Volume, Complexity, Duplication, Unit Size, Unit Testing); la terza infine arriva alla definizione di Technical Debt tramite l'analisi SQALE (Software QuALity Enhancement) basata sul *Remediation Cost*, che rappresenta la quantità di lavoro, in ore e minuti, necessario per correggere eventuali *bug*, *code smell* o altri problemi individuati nel codice sorgente di un software. SonarQube utilizza proprio quest'ultima definizione calcolata in base al *Remediation Cost*. Inoltre in base al valore di questa metrica è possibile identificare quanto il codice sorgente sia mantenibile: più alto è il suo valore, meno il codice è mantenibile.

- **Technical Debt Ratio (TDR):** è una metrica che si calcola come il rapporto tra il costo per sviluppare il software e il costo per risolvere eventuali problemi che si verificano durante lo sviluppo. Inoltre per il calcolo di questa metrica, il valore del costo per sviluppare una singola riga di codice è di 0,06 giorni.

2.4 Descrizione dell'output

Per quanto riguarda l'output dello script, la scelta è ricaduta sul formato CSV (Comma-Separated Values), letteralmente *valori separati da virgole*, un formato di file basato su file di testo utilizzato per l'importazione ed esportazione di una tabella di dati.

Il motivo principale per il quale è stato scelto questo formato è che i dati raccolti nei file CSV sono facili da importare in qualsiasi foglio di calcolo o in qualsiasi database di archiviazione e inoltre sono facilmente estraibili con qualche riga di codice in diversi linguaggi di programmazione tra cui Python e R.

Quest'ultimo fattore sarà utilissimo nella parte successiva di questo lavoro di tesi in quanto sarà necessario estrarre le diverse metriche calcolate dallo script per poi analizzarne l'andamento e determinare una correlazione tra le due tipologie di metriche prese in esame.

I due file di output più nello specifico riguardano le metriche di fragilità delle suite di test e della qualità del codice sorgente, rispettivamente per le metriche calcolabili per ogni versione e per quelle calcolabili per l'intero progetto.

Il primo file infatti è composto da:

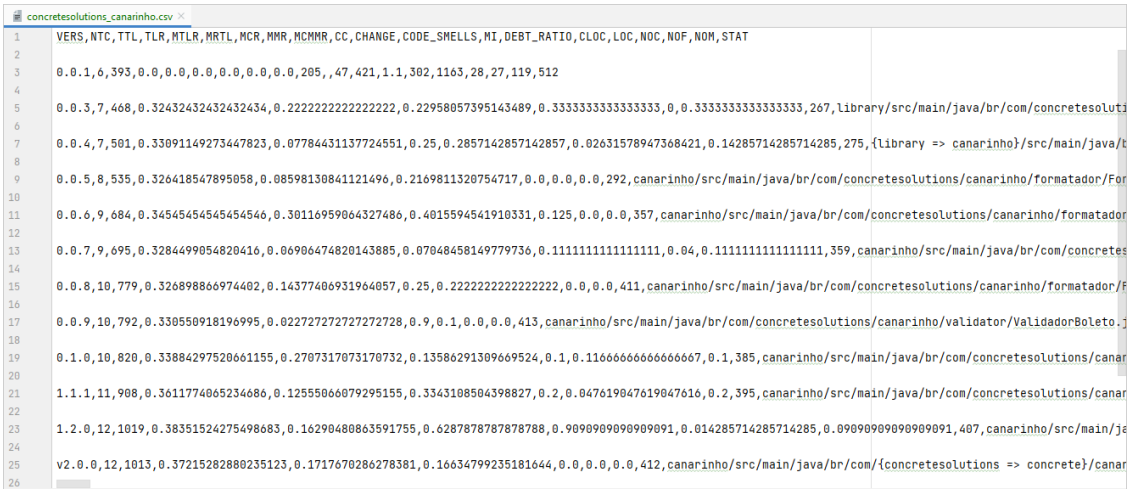
- una prima riga contenente tutti i codici identificativi delle metriche che sono state illustrate in questo capitolo, riguardanti le singole versioni, suddivise per gruppo e in ordine alfabetico;
- una riga per ogni versione rilasciata del progetto che contiene l'identificativo *tag* della versione e i valori delle metriche di fragilità *NTC*, *TTL*, *TLR*, *MTLR*, *MRTL*, *MCR*, *MMR* e *MCMMR*, e delle metriche di qualità *CC*, *CHANGE*, *CODE SMELLS*, *TD*, *TDR*, *CLOC*, *LOC*, *NOC*, *NOF*, *NOM* e *STAT*. Solo nel caso della prima versione i valori non calcolabili, in quanto ancora non presente una versione precedente con cui effettuare un confronto, sono indicati con il valore `"-"`.

Invece il secondo file, che riguarda le metriche dell'intero progetto, contiene:

- una prima riga contenente i codici identificativi delle metriche che sono state illustrate in questo capitolo, riguardanti l'intero progetto, suddivise per gruppo e in ordine alfabetico;
- infine un'ultima riga contenente i valori delle metriche di fragilità dei test *NTR*, *MRR* e *TSV*.

Per quanto riguarda il formato delle varie metriche, per quasi tutte le metriche che vengono calcolate si tratta di valori numerici, interi o con virgola, ad eccezione della metrica *CHANGE* che invece contiene il nome delle varie classi che vengono modificate tra due versioni consecutive e il numero di righe di codice modificate per ogni classe.

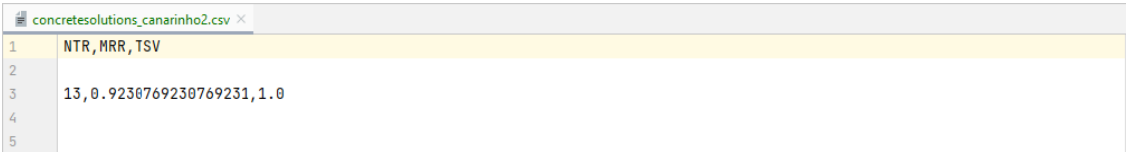
Nelle figure 2.2 e 2.3 sono illustrate due esempi di file CSV di output dello script per quanto riguarda uno dei progetti che verranno presi in esame durante l'esperimento di questo lavoro di tesi.



concretesolutions_canarinho.csv

```
VERS,NTC,TTL,TLR,MTLR,MRTL,MCR,MMR,MCMMR,CC,CHANGE,CODE_SMELLS,MI,DEBT_RATIO,CLOC,LOC,NOC,NOF,NOM,STAT

0.0.1,6,393,0.0,0.0,0.0,0.0,0.0,0.0,205,,47,421,1.1,302,1163,28,27,119,512

0.0.3,7,468,0.32432432432432434,0.2222222222222222,0.22958057395143489,0.3333333333333333,0,0.3333333333333333,267,library/src/main/java/br/com/concretesoluti

0.0.4,7,501,0.33091149273447823,0.07784431137724551,0.25,0.2857142857142857,0.02631578947368421,0.14285714285714285,275,{library => canarinho}/src/main/java/b

0.0.5,8,535,0.326418547895058,0.08598130841121496,0.2169811320754717,0.0,0.0,0.0,292,canarinho/src/main/java/br/com/concretesolutions/canarinho/formatador/For

0.0.6,9,684,0.34545454545454546,0.30116959064327486,0.4015594541910331,0.125,0.0,0.0,357,canarinho/src/main/java/br/com/concretesolutions/canarinho/formatador

0.0.7,9,695,0.3284499054820416,0.06906474820143885,0.0704845814977936,0.1111111111111111,0.04,0.1111111111111111,359,canarinho/src/main/java/br/com/concretes

0.0.8,10,779,0.326898866974402,0.14377406931964057,0.25,0.2222222222222222,0.0,0.0,411,canarinho/src/main/java/br/com/concretesolutions/canarinho/formatador/F

0.0.9,10,792,0.330550918196995,0.022727272727272728,0.9,0.1,0.0,0.0,413,canarinho/src/main/java/br/com/concretesolutions/canarinho/validator/ValidadorBoleto.j

0.1.0,10,820,0.33884297520661155,0.2707317073170732,0.13586291309669524,0.1,0.11666666666666667,0.1,385,canarinho/src/main/java/br/com/concretesolutions/canar

1.1.1,11,908,0.3611774065234686,0.12555066079295155,0.3343108504398827,0.2,0.0476190476190476,0.2,395,canarinho/src/main/java/br/com/concretesolutions/canar

1.2.0,12,1019,0.38351524275498683,0.16290480863591755,0.6287878787878788,0.9090909090909091,0.014285714285714285,0.09090909090909091,407,canarinho/src/main/ja

v2.0.0,12,1013,0.37215282880235123,0.1717670286278381,0.16634799235181644,0.0,0.0,0.0,412,canarinho/src/main/java/br/com/{concretesolutions => concrete}/canar
```

Figura 2.2. Un esempio di primo file di output CSV (Comma-Separated Values) rilasciato dallo script.



concretesolutions_canarinho2.csv

```
NTR,MRR,TSV

13,0.9230769230769231,1.0
```

Figura 2.3. Un esempio di secondo file di output CSV (Comma-Separated Values) rilasciato dallo script.

Capitolo 3

Esperimento e risultati

L'obiettivo principale di questo lavoro di tesi è trovare una correlazione tra la qualità del codice sorgente di un software e la fragilità delle suite di test dello stesso programma.

Il punto di partenza è stato sviluppare uno script che fosse in grado di calcolare due set di metriche utili per stabilire quanto un software fosse mantenibile e quanto la sua suite di test fosse fragile.

3.1 Definizione delle Research Questions

Per raggiungere gli obiettivi sovra indicati, è stato necessario introdurre le seguenti *Research Questions*:

RQ1 Fragilità dei test: in base alle metriche calcolate quanto le suite di test dei software mobile di tipo Android sono fragili?

RQ2 Qualità del codice: in base alle metriche calcolate quanto il codice sorgente dei programmi mobile di tipo Android è di qualità, e più nello specifico è mantenibile?

RQ3 Correlazione tra Fragilità e Qualità: esiste una correlazione tra la qualità del codice sorgente e la fragilità delle sue suite di test di un software mobile di tipo Android?

3.2 Definizione della metodologia

L'approccio adottato per la selezione dell'insieme dei progetti da utilizzare per rispondere alle *Research Questions* e per raggiungere gli obiettivi di questo lavoro di tesi è una sequenza di diversi passaggi.

Il primo passaggio è stato fare una ricerca tramite il *Repository Search API*[1] di GitHub per trovare un set di progetti mobile di tipo Android per poterne analizzare l'evoluzione della qualità del codice e della fragilità delle suite di test nelle varie versioni rilasciate. Questa ricerca si è basata sostanzialmente su due parole chiave:

- la parola *Android* nei nomi dei progetti, nelle loro descrizioni e nei loro file *readme*, fondamentale per scovare tutti i progetti di tipo Android presenti nella piattaforma GitHub.

- la parola *test* nel path assoluto di almeno una classe .java, per garantire la presenza di una suite di test valida a stabilire la fragilità dei test del progetto stesso.

Questa prima ricerca ha portato alla selezione di 5223 progetti GitHub di tipo Android con almeno una classe di test da poter analizzare. Nell'appendice digitale Pirrigheddu (2021) è possibile consultare l'elenco dei progetti selezionati con le tre qualità specificate.

Il secondo passo è stato fare un'analisi approfondita per questo insieme di progetti in modo tale da ottenere un numero adeguato di applicazioni mobile da analizzare con lo script principale. Questo è stato fatto attraverso un altro script in Python che ha permesso di calcolare tre caratteristiche principali per ogni progetto per poi permettere di filtrare, attraverso l'utilizzo di adeguate *thresold*, il primo set di progetti.

I vari passaggi di filtraggio per arrivare al set di progetti GitHub per l'esperimento di questo lavoro di tesi sono ricapitolati nel diagramma di flusso della Figura 3.1.

Le tre qualità analizzate sono:

- **NTR (Number of Tagged Releases)**, il numero di versioni rilasciate. Questa caratteristica è necessaria in quanto lo scopo di questa tesi è strettamente collegato all'evoluzione dei progetti nelle loro versioni.

[1] https://docs.github.com/en/rest/reference/search

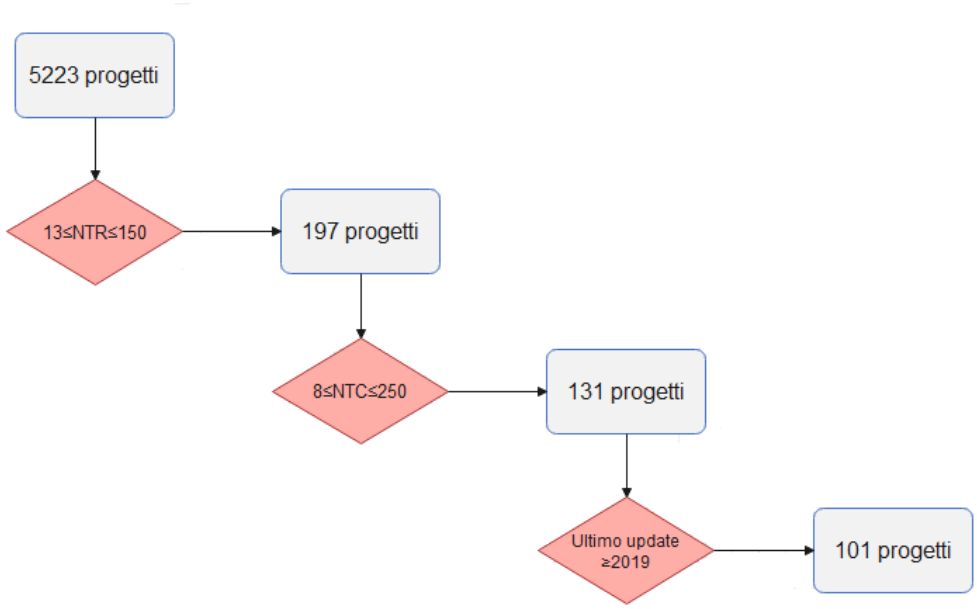


Figura 3.1. Le varie fasi di filtraggio per la selezione dei progetti GitHub valida per l'esperimento.

- **NTC (Number of Test Classes)**, il numero di classi di test dell'ultima versione del progetto è la seconda qualità inclusa in questo ultimo filtraggio. Si tratta di una caratteristica anche questa necessaria in quanto l'obiettivo della tesi riguarda l'evoluzione e la fragilità delle suite di test dei progetti analizzati.

- **Data dell'ultimo aggiornamento:** come ultima qualità si è deciso di introdurre la data dell'ultimo aggiornamento su GitHub per evitare di analizzare progetti datati e ormai archiviati e per riuscire ad includere progetti scritti sia in Java sia in Kotlin, linguaggio nuovo presente solo da pochi anni negli standard Android.

L'ultimo passaggio è stato quello di scegliere delle opportune *thresold* per le caratteristiche analizzate in modo da arrivare alla selezione di un set di 100 progetti circa:

- $13 \leq NTR \leq 150$: la soglia minima di 13 unità per il numero di versioni è dovuta alla necessità di analizzare i progetti in tutta la loro evoluzione per la buona riuscita dell'esperimento. La soglia massima di 150 invece è

semplicemente una scelta dovuta alle tempistiche, in quanto lo script che analizza la qualità del codice e della fragilità dei test impiega quasi più di un giorno per progetti con più di 150 versioni rilasciate. Questo non crea problemi per la riuscita dell'esperimento in quanto il numero di versioni è altamente sufficiente per calcolare l'evoluzione del progetto GitHub.

- $8 \leq NTC \leq 250$: la soglia minima di 8 classi di test è necessaria in quanto parte dell'esperimento riguarda la misurazione della fragilità delle suite di test. Invece la soglia massima di 250 classi di test è stata inserita per motivi di praticabilità in quanto è ritenuta sufficiente per la buona riuscita dell'esperimento ed è stato constatato che progetti con più di 250 classi di test portavano lo script a richiedere un tempo notevolmente maggiore rispetto a progetti con meno classi di test.
- *Data ultimo aggiornamento* $\geq$ 01/01/2019: per quest'ultima qualità, la soglia unica di un aggiornamento dopo l'inizio del 2019 è ritenuta necessaria per includere progetti in continua evoluzione e progetti che presentano le novità che si stanno sviluppando in questi ultimi anni, tra cui il linguaggio Kotlin.

Inoltre, nell'Appendice A è presente la Tabella A.1 che illustra il set completo dei progetti GitHub con le 3 caratteristiche analizzate (NTR, NTC e Data ultimo aggiornamento).

3.3 Strumenti utilizzati per l'analisi dei risultati

Per quanto riguarda l'analisi dei risultati, una volta analizzati i 101 progetti GitHub selezionati, sono stati ricavati i relativi file CSV con tutte le metriche di fragilità e di qualità versione per versione.

Per facilitare la visualizzazione dei risultati, per ogni singolo progetto si è calcolata la media per ogni singola metrica, soltanto dove questa è definita, in modo tale da avere a disposizione un unico valore per progetto per ogni metrica analizzata.

Per la rappresentazione dei dati si è deciso di utilizzare i *violin plot*. Questo tipo di grafici sono molto simili ai *box plot* ma contengono un'informazione aggiuntiva: la densità di probabilità del kernel dei dati. Inoltre in questo studio i *violin plot* includono un marker per la media ed un altro per la mediana dei dati riguardanti ogni singola metrica. Il primo marker è identificato da un piccolo rombo all'interno del grafico, mentre per il secondo si tratta di un puntino rosso.

In questo studio la presenza di diversi *violin plot* è dovuta dal fatto che si è ritenuto necessario suddividere i grafici in base a due fattori:

- Tipologia e gruppo di appartenenza delle metriche.
- Ordine di grandezza della metrica, per evitare che alcuni grafici divenissero illeggibili.

Inoltre per la dimostrazione della presenza di una correlazione tra le metriche di qualità del codice sorgente e della fragilità dei test, sono stati utilizzati gli *scatter plot*: si tratta di un grafico a dispersione in cui due variabili di un set di dati sono riportate su uno spazio cartesiano. In questo tipo di grafici è anche possibile tracciare una linea retta utile a dimostrare la dipendenza lineare tra le due variabili.

Infine le medie e le mediane delle metriche analizzate per tutti i 101 progetti sono riportate nei paragrafi successivi all'interno di due tabelle, rispettivamente per le metriche di fragilità delle suite di test (3.1), e per le metriche di qualità del codice sorgente (3.2).

3.3.1 RStudio

Per analizzare i risultati e le metriche dei 101 progetti GitHub derivanti dallo script principale, è stato utilizzato un software specifico per il calcolo statistico, RStudio[2].

RStudio è un ambiente di sviluppo integrato (IDE), di tipo Open-Source, che si basa sull'utilizzo del linguaggio di programmazione R. Questo software è presente sia in versione Desktop che in versione Server, in base alla macchina dove viene eseguito.

La scelta dell'utilizzo del linguaggio R è dovuta dal fatto che solitamente viene utilizzato per l'analisi statistica di dati e che permette in maniera molto semplice l'importazione e la manipolazione di Dataset da file CSV.

3.4 Fragilità delle suite di test (RQ1)

Nella Tabella 3.1 sono presenti tutte le medie e le mediane per quanto riguarda le metriche di fragilità delle suite di test dei 101 progetti GitHub analizzati.

[2]https://www.rstudio.com/

Tabella 3.1. Valori medi e mediane per le metriche calcolate di fragilità delle suite di test.

Gruppo	**Metrica**	**Valore Medio**	**Mediana**
Diffusion and Size	NTR	49	41
	NTC	33	16
	TTL	3212	1150
	TLR	0.20	0.15
Evolution	MTLR	1.15	0.29
	MRTL	0.25	0.23
	MCR	0.30	0.26
	MMR	0.05	0.04
	MCMMR	0.13	0.11
	MRR	0.45	0.45
	TSV	0.62	0.62

Quello che si può notare è che il valore medio di quasi tutte le metriche di fragilità dei test si avvicina alla mediana, ad eccezione delle metriche NTC, TTL e MTLR. Queste eccezioni sono dovute al fatto che: tra i programmi analizzati, per quanto riguarda la metrica NTC, sono presenti molti progetti con poche classi di test, precisamente con un numero compreso tra 8 e 16, quindi si evince facilmente che in generale i progetti di tipo Android non vengono testati in maniera sistematica con tool automatici. Di conseguenza la mediana risulta essere più bassa della media che si innalza in quanto alcuni progetti superano le 100 classi di test. Questo fattore appena illustrato influenza anche la metrica TTL in quanto quest'ultima mostra quante righe di codice sono incluse nelle classi di test e, di conseguenza, il valore medio e la mediana seguono l'andamento della metrica NTC. Infine, discorso diverso è dato dai valori della metrica MTLR, dove il valore medio risulta essere molto più alto della mediana per via della presenza di un outlier.

Ora saranno analizzati più nello specifico i singoli grafici che riguardano le metriche di fragilità delle suite di test dei 101 progetti GitHub esaminati per questo lavoro di tesi.

Partendo dal gruppo di metriche *Diffusion and Size*, la Figura 3.2 mostra la distribuzione dei dati delle metriche NTR e NTC che sono state calcolate nello script principale in Python attraverso le loro definizioni, cioè rispettivamente il numero di versioni rilasciate con il comando *Repo.git.tag* di GitPython e il numero di classi di test con la ricerca nel path di ogni classe Java o Kotlin della parola *test*. Per la prima metrica la distribuzione dei dati è concentrata soprattutto nell'intervallo

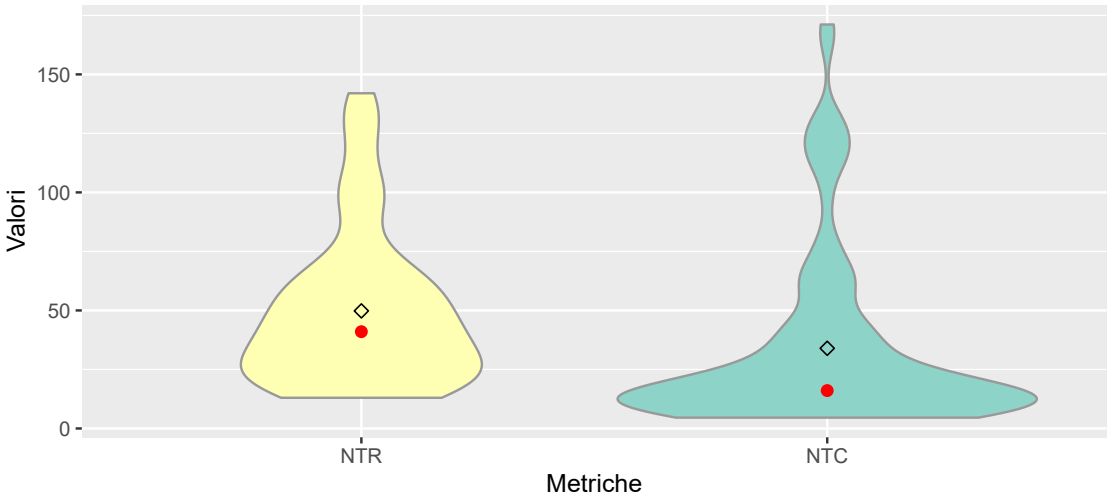


Figura 3.2. Violin plot per le metriche NTR e NTC.

di valori che va da 20 a 70 circa. Invece per la seconda metrica la distribuzione è sviluppata per la maggior parte dei progetti nei valori compresi tra 8 e 30. Inoltre si nota facilmente come per la metrica NTC il valore della media (rombo nero) risulta essere molto più alto della mediana (puntino rosso), cosa che non accade per la metrica NTR.

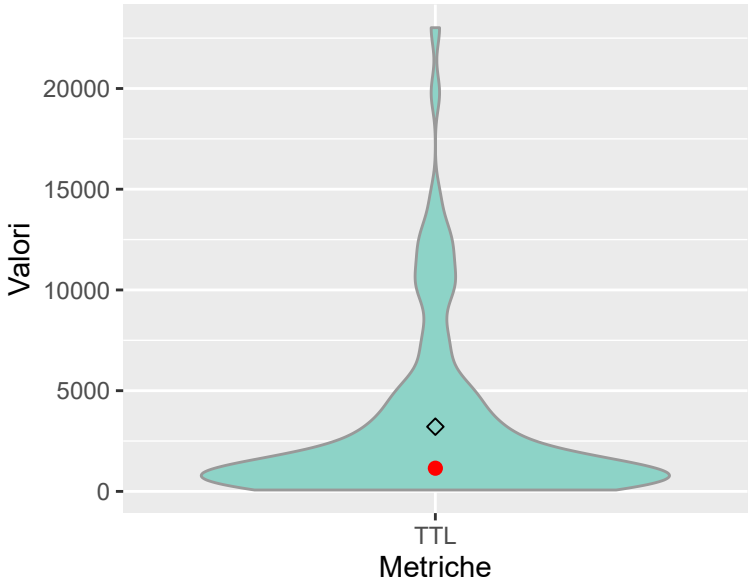


Figura 3.3. Violin plot per la metrica TTL.

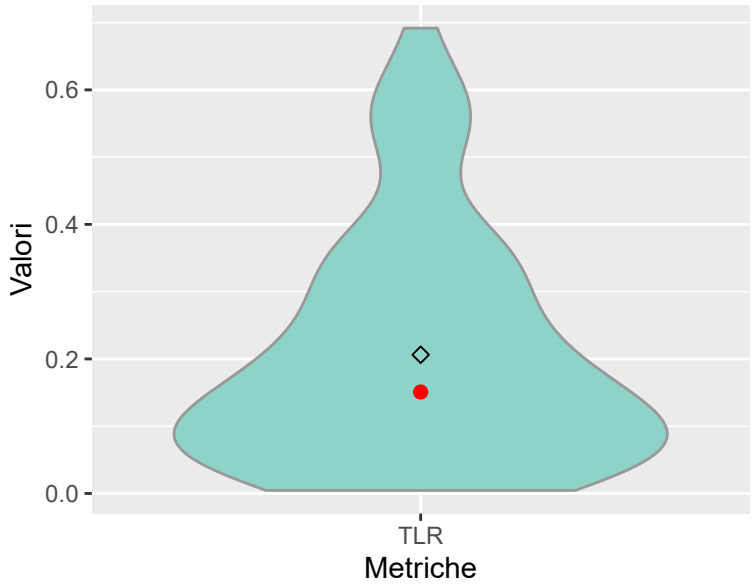


Figura 3.4. Violin plot per le metrica TLR.

La Figura 3.3 illustra la distribuzione dei dati della metrica TTL, cioè del numero totali di righe di codice che riguardano le classi di test dei progetti analizzati, ed è stata calcolata attraverso l'utilizzo del software *Cloc* per ogni classe di test. La distribuzione dei dati di questa metrica è situata, per la maggior parte dei valori, nell'intervallo che va da 0 a 5000 righe di codice. Questo risultato significa che, nella maggior parte dei progetti analizzati il numero totale di righe di codice che riguardando le classi di test è inferiore a 5000 e che sono pochi i progetti che presentano fino a 10000 TTL e che sono ancora meno quelli che presentano sulle 20000 righe di codice di test. Anche nel caso della metrica TTL, così come NTC, il valore della media è molto più alto, quasi il doppio, del valore della mediana.

La Figura 3.4 mostra la distribuzione dei dati della metrica TLR, calcolata nello script principale in Python attraverso la sua definizione (2.1), cioè il rapporto tra le righe di codice di test e le righe di codice dell'intero progetto. La distribuzione di questa metrica è sviluppata soprattutto tra i valori 0.1 e 0.4 per la maggior parte dei progetti, fino ad arrivare ad un valore vicino allo 0.7 per la restante parte. Questo risultato sta a significare che per il maggior numero di progetti analizzati il numero delle righe di codice di test non arriva neanche alla metà del numero di righe dell'intero progetto.

Per quanto riguarda il gruppo di metriche *Evolution*, la Figura 3.5 mostra invece

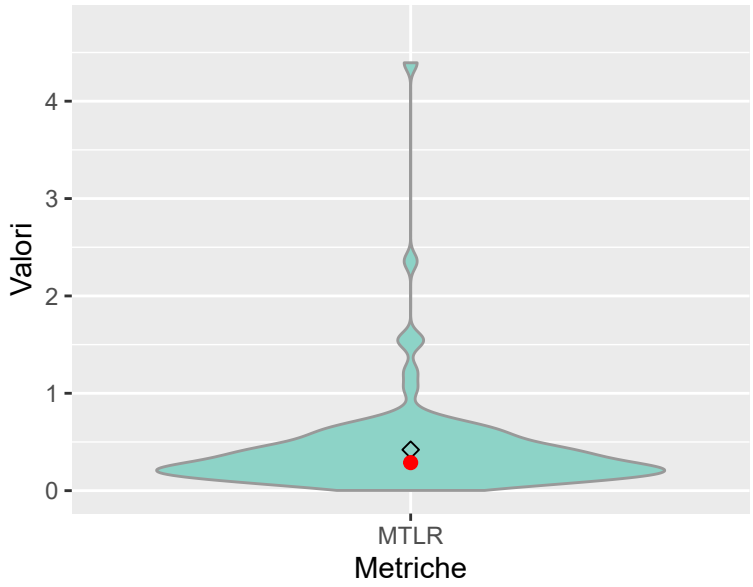


Figura 3.5. Violin plot per la metrica MTLR.

la distribuzione dei dati della metrica MTLR, calcolata nello script principale in Python attraverso la sua definizione (2.2), cioè il rapporto tra le righe di codice di test aggiunte, modificate o cancellate e le righe di codice totali delle classi di test della versione precedente. Per rendere leggibile questo *violin plot* è stato necessario rimuovere un outlier di valore superiore a 60, che avrebbe appiattito in maniera eccessiva il grafico. Ora invece si può benissimo notare come per quasi tutti i 101 progetti analizzati il valore della metrica MTLR risulta essere tra 0 e 1. Inoltre per questa metrica, come si evince dalla tabella, il valore della media risulta essere molto più alto della mediana e anche questo è dovuto alla presenza di un outlier. Infatti, una volta rimosso l'unico valore del progetto outlier, nella Figura 3.5 si può notare che la media e la mediana risultano molto vicine tra loro e si attestano intorno allo 0.5.

La Figura 3.6 mostra le distribuzioni dei dati delle metriche:

- MRTL, definita dalla formula (2.3), viene calcolata come il rapporto tra le righe di codice modificate riguardanti le classi di test e le righe di codice modificate riguardanti l'intero progetto.
- MCR, definita dall'equazione (2.4), viene calcolata come il rapporto tra le classi di test modificate e il numero di classi di test della versione precedente.

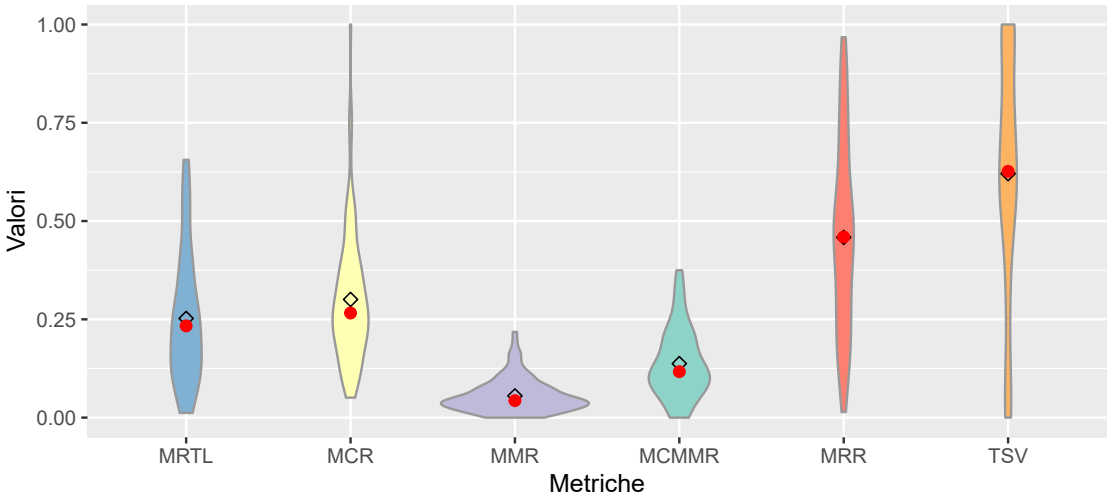


Figura 3.6. Violin plot per le metriche MRTL, MCR, MMR, MCMMR, MRR e TSV.

- MMR, definita dalla formula (2.5), viene calcolata come il rapporto tra il numero di metodi di test modificati e il numero totale di metodi di test.
- MCMMR, definita dall'equazione (2.6), viene calcolata come il rapporto tra il numero di classi di test modificate con almeno un metodo modificato e il numero di classi di test della versione precedente.
- MRR, definita dalla formula (2.7), viene calcolata come il rapporto tra il numero di versioni dove è stata modificata almeno una classe di test e il numero di versioni rilasciate.
- TSV che viene calcolata come il rapporto tra il numero di classi di test modificate almeno una volta e il numero di classi di test totali nella storia del progetto.

Il *violin plot* che rappresenta la distribuzione dei dati per queste metriche di fragilità dei test illustra come:

- per le metriche MRTL e MCR il valore per la maggior parte dei casi si stanzia nell'intervallo 0.1 e 0.5. Inoltre, la media e la mediana per queste due metriche sono molto simili e si attestano intorno allo 0.25.

Tabella 3.2. Valori medi e mediane per le metriche calcolate di qualità del codice sorgente.

Gruppo	Metrica	Valore Medio	Mediana
Complexity	CC	3046	1991
Size	CLOC	5535	2396
	LOC	18272	10996
	NOC	245	147
	NOF	302	190
	NOM	1572	1036
	STAT	7655	5049
Stability	CHANGE	121393	51517
	CODE SMELLS	943	517
	TD	7577	4042
	TDR	0.95	0.78

- per le metriche MMR e MCMMR i dati sono distribuiti tra i valori 0.01 e 0.2, con media e mediana che hanno un valore molto basso, tra 0.04 e 0.11.
- infine per le metriche MRR e TSV i valori sono quasi distribuiti in maniera uniforme per tutto il range che va da 0 e 1. Per queste due metriche infatti i valori di media e mediana risultano essere molto più alti rispetto alle precedenti, rispettivamente 0.45 e 0.62.

Infine in questo insieme di metriche riguardanti le suite di test (MRTL, MCR, MMR, MCMMR, MRR e TSV), valori più alti evidenziano una fragilità maggiore delle classi di test che devono essere modificate più spesso quando avvengono dei cambiamenti nel codice sorgente del software.

I risultati ottenuti per quanto riguarda la fragilità delle suite di test mostrano come per la maggior parte dei progetti si tratta di risultati positivi, cioè i test non sono considerabili molto fragili, mentre soltanto per alcuni progetti con metriche che presentano valori molto alti la fragilità risulta di una certa importanza.

3.5 Qualità del codice sorgente (RQ2)

Nella Tabella 3.2 sono presenti tutte le medie e le mediane per quanto riguarda le metriche di qualità e manutenibilità del codice sorgente dei 101 progetti GitHub che sono stati analizzati.

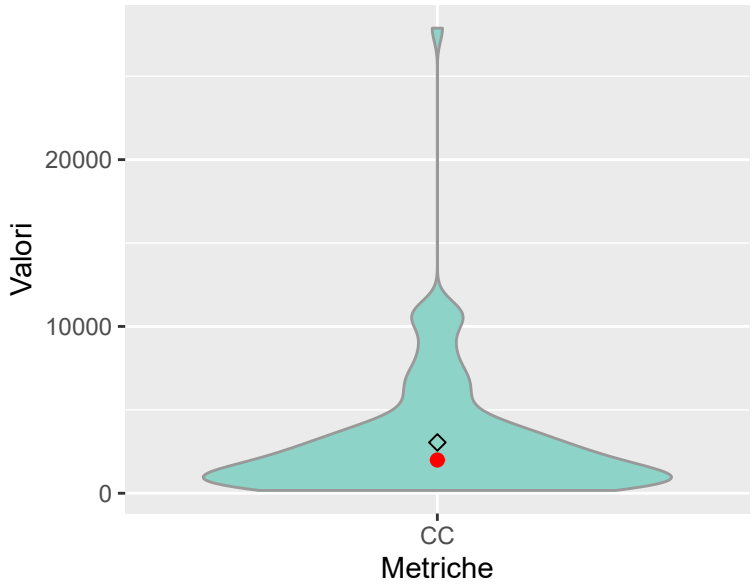


Figura 3.7. Violin plot per la metrica CC.

A differenza di ciò che è stato detto per le metriche di fragilità, dalla Tabella 3.2 e soprattutto dai vari *violin plot* emerge che per tutte le metriche di qualità del codice i valori della media e della mediana non si discostano molto uno dall'altro, soprattutto se questa considerazione viene fatta tenendo conto anche dell'ordine di grandezza di ogni singola metrica.

Per quanto riguarda la metrica CC del gruppo *Complexity* calcolata secondo la sua definizione dal software SonarQube, la Figura 3.7 indica la distribuzione dei dati della metrica CC e mostra come la complessità ciclomatica della maggior parte dei progetti analizzati sia compresa tra i valori 500 e 10000, con qualche progetto outlier che vede la sua CC salire quasi fino a 30000. Il motivo di tale valore è senz'altro il fatto che si tratta di un progetto molto grande, infatti andando a controllare per esempio la metrica LOC anche il suo valore supera di gran lunga tutti gli altri progetti. La Complessità Ciclomatica per definizione calcola il numero di cammini linearmente indipendenti attraverso il grafo di controllo di flusso, che quindi con un codice sorgente molto grande sarà in numero maggiore rispetto agli altri progetti. I valori di media e mediana sono molto vicini rispetto all'ordine di grandezza e si attestano tra i 2000 e i 3000.

Inoltre sempre secondo la definizione di CC, valori più alti di questa metrica stabiliscono una qualità e manutenibilità del codice sorgente migliore rispetto a

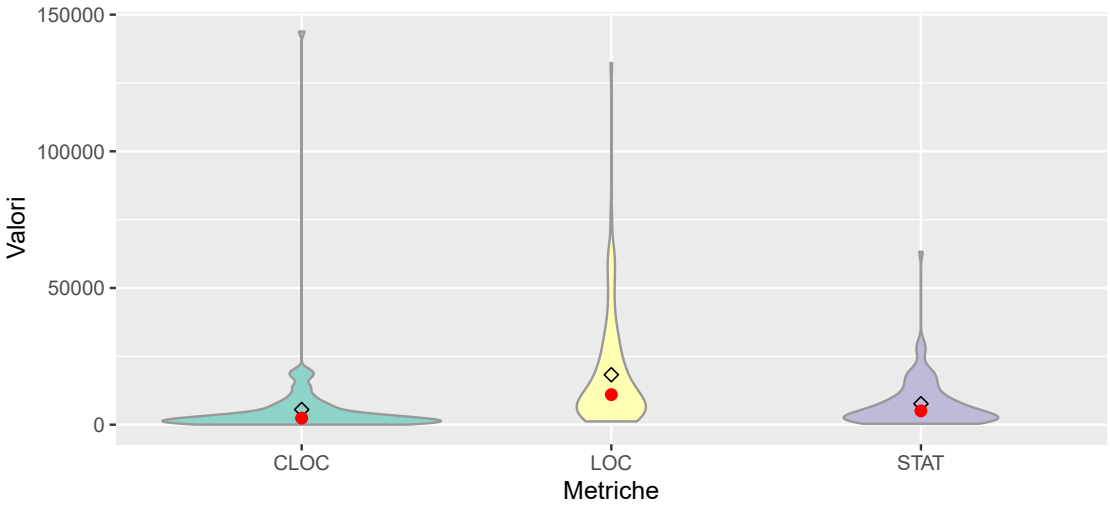


Figura 3.8. Violin plot per le metriche CLOC, LOC e STAT.

progetti con un valore di CC più basso.

Per quanto riguarda il gruppo *Size*, la Figura 3.8 mostra le distribuzione dei dati delle metriche CLOC, LOC e STAT. Le prime due metriche sono utili per calcolare la qualità del codice in quanto illustrano rispettivamente i numeri di righe di codice di commento e il numero di righe di codice logiche relativi al software analizzato. Queste due metriche vengono calcolate attraverso l'utilizzo del software Cloc. La terza metrica invece è il numero di *statements* presenti in tutte le classi Java e Kotlin del progetto che viene calcolata con l'ausilio di SonarQube.

Dal *violin plot* è facile notare come per la metrica CLOC, il valore per la maggior parte dei progetti risulta essere fino a 25000, con qualche outlier che vede il suo valore salire fino a quasi 150000. Va ricordato inoltre che per questa metrica, valori più alti indicano una qualità maggiore del codice in quanto un codice ben commentato e documentato risulta più comprensibile e più facilmente modificabile e mantenibile.

Invece, per la metrica LOC, l'intervallo dei valori che comprende la maggioranza dei progetti va da 0 fino a poco meno di 75000, con un outlier in particolare che tocca i 150000 e ciò è dovuto alla grandezza del progetto così come è stato visto anche per la metrica CC.

Infine per la metrica STAT i valori per la maggioranza dei progetti arrivano appena oltre i 20000, con un andamento molto simile alla metrica CLOC se non

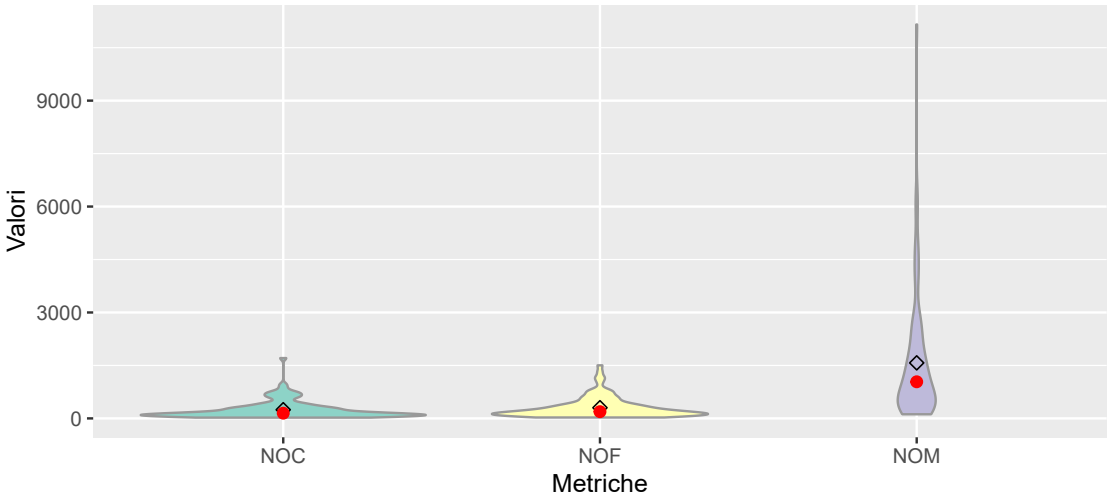


Figura 3.9. Violin plot per le metriche NOC, NOF e NOM.

fosse per la presenza dell'outlier evidenziato in precedenza.

I valori delle medie e delle mediane di queste tre metriche risultano molto simili, con la sola metrica LOC che risulta avere la media decisamente più alta della mediana.

Infine la Figura 3.9 illustra la distribuzione dei dati delle metriche NOC, NOF e NOM che per definizione sono rispettivamente il numero di classi, il numero di file e il numero di metodi presenti nelle classi Java e Kotlin del progetto. Queste tre metriche sono state calcolate con l'ausilio di SonarQube e, per le prime due, la maggioranza dei progetti analizzati comprende l'intervallo di valori che va da 0 a 1000. Questo si tratta di un risultato abbastanza prevedibile in quanto in un progetto software di tipo Android il numero di file si avvicina molto al numero di classi del progetto stesso. Invece per la metrica NOM, l'intervallo dei valori per tutti i progetti analizzati va da 0 a poco meno di 10000.

Inoltre dalla Tabella 3.2 si possono confrontare le medie delle metriche di qualità NOC e NOM per stabilire quanti metodi sono presenti, sempre in media, per classe: questo confronto è utile per stabilire la complessità delle classi dei progetti analizzati, in quanto una classe con meno metodi equivale ad una classe meno complessa e di conseguenza più comprensibile. In questo caso, si può notare che la media di metodi per classe risulta essere uguale a 6.4 e che quindi in media sono presenti pochi metodi all'interno delle classi dei progetti analizzati e di conseguenza

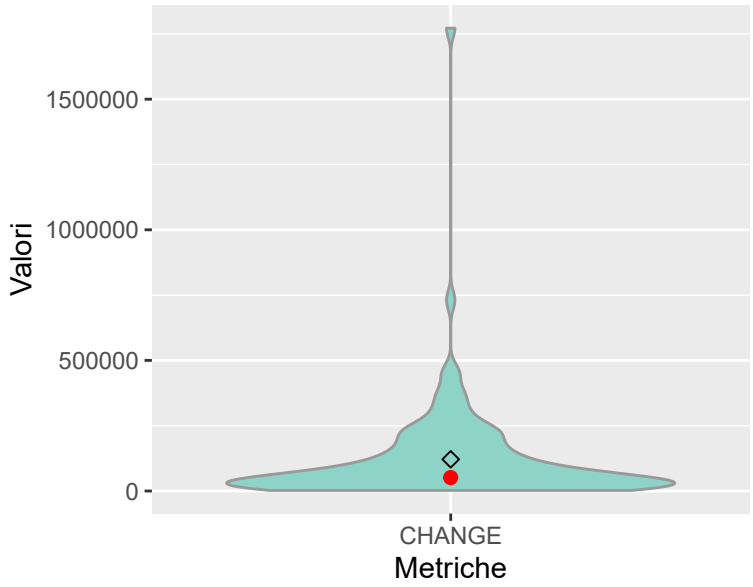


Figura 3.10. Violin plot per la metrica CHANGE.

le classi in generale non sono molto complesse.

Per le medie e le mediane vale lo stesso discorso fatto fino ad ora per tutte le metriche di qualità del codice sorgente, cioè i valori sono molto simili come si evince dal *violin plot*.

Passando alle metriche di tipo *Stability*, la Figura 3.10 illustra la distribuzione dei dati della metrica CHANGE, il cui valore viene calcolato sommando i diversi numeri di modifiche che vengono effettuate per ogni classe Java o Kotlin all'interno dei progetti per ogni versione. Dal grafico si evince che per la maggioranza dei progetti il valore della CHANGE si stabilisce tra lo 0 e i 500000, con qualche progetto che addirittura supera il milione. Valori così alti stanno a significare che nel progetto in questione le modifiche che sono state fatte versione per versione presentano un quantitativo di aggiunte, correzioni ed eliminazioni di righe di codice sorgente molto alto.

La Figura 3.11 invece mostra la distribuzione dei dati della metrica CODE SMELLS che viene calcolata attraverso SonarQube.

Il *violin plot* illustra come la metrica sia definita, per la maggior parte dei progetti, in un intervallo di valori che va tra 0 e 4000, con qualche progetto outlier che arriva fino a 10000 circa. Va ricordato che questa metrica non individua il numero di bachi presenti nel codice, ma il numero di piccole problematiche che

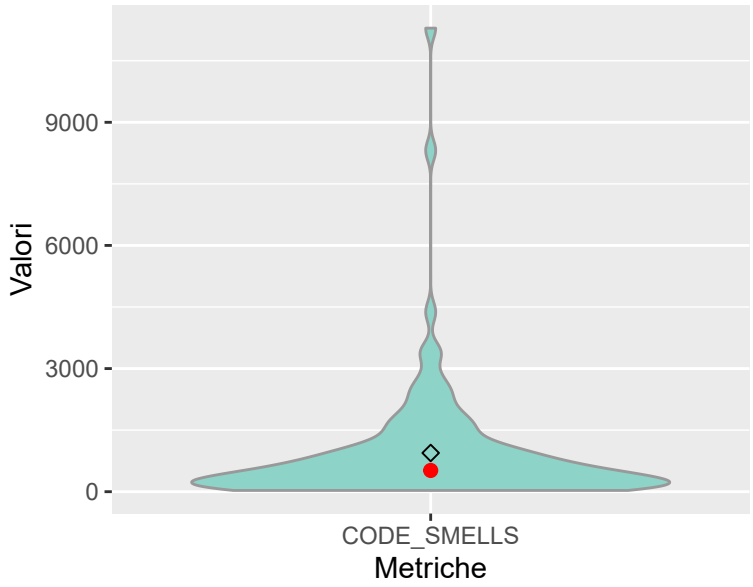


Figura 3.11. Violin plot per la metrica CODE SMELLS.

portano il codice ad essere meno di qualità. Dunque maggiore sono i valori di tale metrica e minore è la qualità del programma.

Grafico e tabella mostrano infine che, anche per questa metrica di manutenibilità del codice sorgente, media e mediana presentano dei valori molto simili tra loro.

La Figura 3.12 mostra la distribuzione dei dati della metrica TD ed evidenzia come il *Technical Debt* dei progetti analizzati, calcolato sempre grazie all'aiuto di SonarQube, vede la maggior parte di loro avere un valore tra 0 e 25000, con qualche progetto che sale tra i 50000 e i 75000. Per definizione questa metrica risulta essere il numero di secondi necessari per sistemare le violazioni alle regole presenti all'interno del codice sorgente, di conseguenza valori così alti stanno ad evidenziare progetti composti da codice di una scarsa qualità in quanto necessitano di molto tempo per essere sistemati.

Dal grafico e dalla Tabella 3.2 riguardanti questa metrica si evince che i valori di media e mediana sono molto vicini tra loro.

La Figura 3.13 illustra la distribuzione dei dati per quanto riguarda la metrica TDR del gruppo *Stability*. Anche questa metrica viene calcolata attraverso il software SonarQube e va ad individuare il rapporto tra il costo per sviluppare il software e il costo per rimediare i problemi che si verificano durante il suo sviluppo. Si nota dal grafico che i valori di TDR sono molto bassi e per il numero più grande

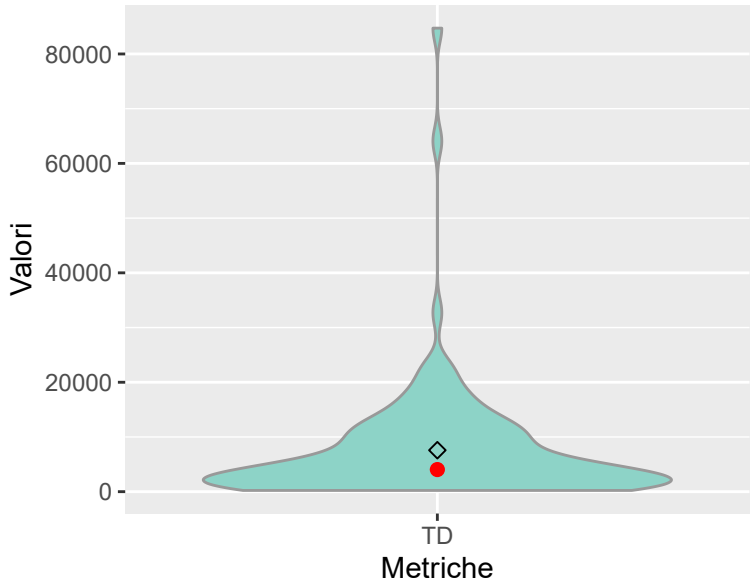


Figura 3.12. Violin plot per la metrica TD.

di progetti si stanziano tra 0.5 e 2. Valori così bassi, per definizione di tale metrica, indicano una buona qualità del codice dei progetti che sono stati analizzati. Media e mediana inoltre sono molto vicine come valore e si attestano tra 0.78 e 0.95.

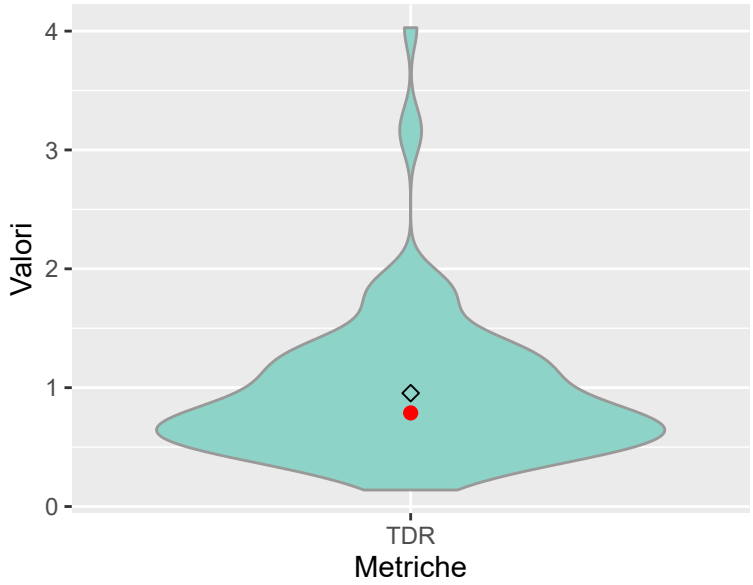


Figura 3.13. Violin plot per la metrica TDR.

3.6 Studio di correlazione (RQ3)

Per raggiungere l'ultimo obiettivo di questo lavoro di tesi, cioè la ricerca di una possibile correlazione tra le metriche di manutenibilità del codice sorgente e le metriche di fragilità delle suite di test, si è deciso di utilizzare la tecnica statistica della regressione lineare semplice, applicata più volte e per tutte le metriche che sono state analizzate.

La regressione lineare è un approccio di tipo lineare utile a trovare una relazione tra due variabili adattando un'equazione lineare ai dati osservati. La prima variabile è considerata di tipo esplicativo, cioè indipendente, mentre l'altra di tipo dipendente.

In questo studio, le metriche di qualità del codice sono le variabili indipendenti, mentre le metriche di fragilità di test sono considerate le variabili dipendenti. Questa scelta è dovuta dal fatto che la manutenibilità del software è una caratteristica primaria del programma, mentre la fragilità delle suite di test viene introdotta in un secondo momento e cioè quando il software viene testato.

Per applicare la regressione lineare semplice è necessario che i dati soddisfino 4 presupposti principali:

- **Indipendenza dei dati:** non deve essere presente nessuna autocorrelazione, ma come è stato già definito le due variabili sulle quali viene applicata la regressione lineare sono una di tipo esplicativo e l'altra di tipo dipendente, di conseguenza questa prima assunzione è rispettata.

- **Normalità:** le variabili sottoposte alla regressione lineare devono avere una distribuzione normale dei dati, cioè con la classica distribuzione dei dati a campana, con più osservazioni nel mezzo e meno nelle code. Eliminati alcuni outlier, è facilissimo notare dai *violin plot* di tutte le metriche analizzate, che tutti i dati raccolti rispettano questo secondo punto.

- **Linearità:** la relazione tra la variabile indipendente e quella dipendente deve essere di tipo lineare. Questo è facilmente verificabile attraverso un semplice grafico a dispersione, come gli *scatter plot* che saranno illustrati in seguito.

- **Omoschedasticità:** conosciuta anche come Omogeneità della varianza e sta ad indicare una proprietà che possiedono una collezione di variabili aleatorie nel momento in cui tutte hanno la stessa varianza.

Tabella 3.3. Tabella dei $Pr(>|t|)$, *p value*, delle regressioni lineari semplici.

	NTR	NTC	TTL	TLR	MTLR	MRTL	MCR	MMR	MCMMR	MRR	TSV
CC	**0.0497**	**0.0003**	**0.0002**	**0.0178**	**0.0387**	**0.0001**	**0.0170**	0.1362	**0.0127**	0.7931	0.6727
CLOC	0.2377	**0.0283**	0.1246	0.2511	0.4277	**0.0456**	**0.0498**	0.6547	0.0622	0.4920	0.9632
LOC	0.0682	**3e-06**	**1e-06**	0.0624	**0.0047**	**0.0004**	**0.0088**	0.0804	**0.0074**	0.3357	0.5999
NOC	0.2918	**1e-08**	**1e-06**	0.1393	0.0516	**0.0005**	**0.0008**	**0.0355**	**0.0014**	0.1741	0.6585
NOF	0.2100	**4e-09**	**7e-08**	0.0865	0.0957	**0.0002**	**0.0031**	**0.0339**	**0.0041**	0.0917	0.8682
NOM	0.1456	**8e-07**	**1e-06**	0.0795	0.0788	**0.0004**	**0.0018**	**0.0273**	**0.0018**	0.2813	0.5797
STAT	**0.0226**	**4e-05**	**7e-06**	0.0681	**0.0100**	**0.0009**	**0.0179**	0.1486	**0.0146**	0.5459	0.5840
CHANGE	**0.0002**	**0.0142**	**0.0006**	0.1733	**3e-24**	**0.0387**	0.6183	0.5010	0.2756	0.6518	0.1669
CODESMELLS	0.2802	**0.0002**	**1e-06**	0.5278	0.1213	**0.0333**	0.0859	0.0936	**0.0244**	0.4336	0.9899
TD	0.1406	**0.0002**	**5e-06**	0.5663	**0.0314**	**0.0276**	**0.0287**	0.1587	**0.0127**	0.7080	0.6624
TDR	0.9029	0.5568	0.1148	**0.0004**	0.9523	**0.0057**	0.4436	0.2637	0.1077	0.3256	0.3781

Quando viene eseguita una regressione lineare semplice vengono calcolati diversi tipi di coefficienti, alcuni dei quali saranno utili per rispondere alla domanda sulla correlazione di questo studio di tesi:

- Le stime (*Estimate*) per i parametri del modello che sta ad indicare di quanto aumenta il valore della variabile dipendente all'aumentare di 1 della variabile indipendente.
- L'errore standard (*Std. Error*) dei valori stimati.
- Il test statistico (*t value*)
- Il *p value*, conosciuto anche come $Pr(>|t|)$, cioè la probabilità di trovare il *t value* nel caso in cui la *null hypothesis* di nessuna relazione fosse vera.

Per il nostro caso, il coefficiente più importante risulta essere il *p value* in quanto se il suo valore risulta essere minore di 0.05, questo indicherà che il modello si adatta bene ai dati e inoltre che esiste una correlazione di tipo lineare tra la variabile indipendente relativa alla manutenibilità del codice e la variabile dipendente relativa alla fragilità dei test.

Nella Tabella 3.3 sono presenti i valori dei *p value* di tutte le regressioni lineari semplici che riguardano le 11 metriche di manutenibilità del codice sorgente e le 11 metriche di fragilità dei test. Inoltre, sono stati evidenziati in grassetto tutti i valori al di sotto della soglia (0.05) che di conseguenza vengono identificati come correlazioni tra metriche.

Si può benissimo notare come i valori evidenziati, e dunque le correlazioni, siano quasi la metà, 59, delle regressioni lineari semplici che sono state calcolate, 121.

3.6.1 Correlazioni principali

Per verificare la presenza di una correlazione e per poter dire qualcosa in più, la scelta è ricaduta sugli *scatter plot* che permettono di evidenziare con una retta all'interno del grafico stesso la relazione di tipo lineare che viene trovata con la regressione lineare semplice.

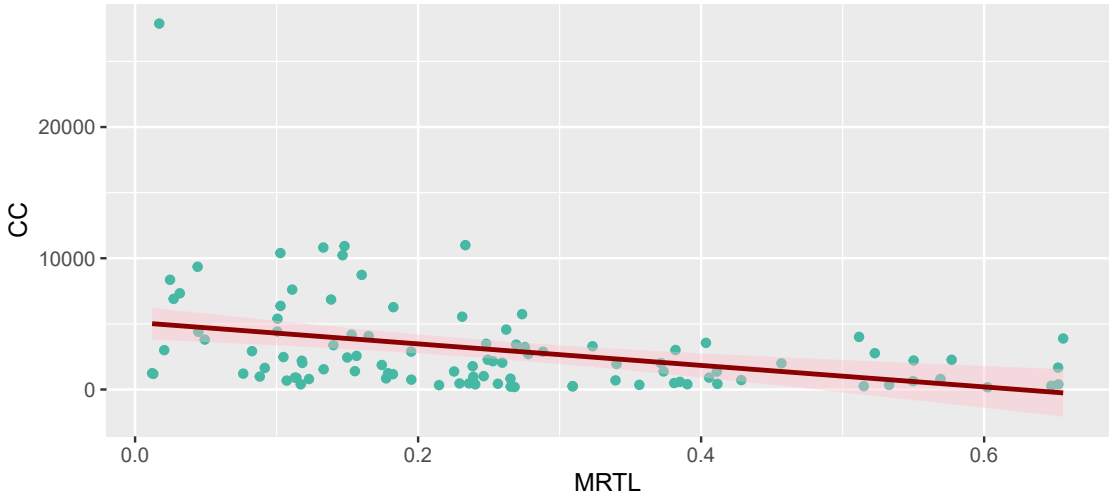


Figura 3.14. Scatter plot della regressione lineare tra le metriche CC e MRTL.

La prima relazione da analizzare riguarda la metrica di qualità del codice CC e la metrica di fragilità dei test MRTL. Partendo da un *p value* di 0.0001 che riguarda la regressione lineare, lo *scatter plot* nella Figura 3.14 evidenzia una relazione lineare semplice che si può sintetizzare così: al diminuire della complessità ciclomatica CC, la metrica di fragilità MRTL aumenta. Inoltre, la relazione lineare che lega queste due metriche è descritta dalla retta $y = 5100 - 8200x$. Questi risultati stanno ad indicare che un programma con una complessità ciclomatica più bassa necessita di più interventi per quanto riguarda le classi di test, che di conseguenza risultano più fragili, portando il valore della metrica MRTL ad aumentare.

La seconda relazione analizzata in questo studio di tesi interessa la metrica di manutenibilità del codice sorgente CHANGE e la metrica di fragilità dei test MTLR. Il valore del *p value* della regressione lineare semplice che riguarda queste

due metriche è di $3e-24$. Nella Figura 3.15 si può notare come la relazione lineare semplice, descritta dalla retta $y = 96000 - 21000x$, mostra come all'aumentare della metrica CHANGE, anche la metrica MTLR tende ad aumentare il suo valore. Queste relazione stabilisce che un programma con poca qualità e di conseguenza con un valore di CHANGE che tende a salire versione per versione, necessita di un valore di MTLR, che indica le modifiche effettuate sulle classi di test, più alto in quanto i test risultano più fragili.

La terza relazione analizzata riguarda la metrica di qualità del codice CODE SMELLS e la metrica di fragilità delle suite di test TTL. La regressione lineare semplice ha portato ad un valore del *p value* di $1e-06$ e la Figura 3.16 evidenzia la relazione lineare che riguarda queste due metriche: all'aumentare della metrica che riguarda piccoli problemi, non veri e propri bug, nella struttura e nella qualità del codice sorgente di un software, il valore delle righe di codice di test aumenta notevolmente. La retta che governa questa relazione lineare è $y = 420 - 0.16x$. Da questa relazione inoltre possiamo dire come anche in questo caso una qualità di codice inferiore porta all'aumento di righe di codice delle classi di test ed a una fragilità maggiore della suite di test stessa.

L'ultima relazione analizzata interessa la metrica di manutenibilità del software TD e la metrica di fragilità dei test MCR. Partendo da un *p value* di 0.0287 che riguarda la regressione lineare, lo *scatter plot* nella Figura 3.17 evidenzia la seguente

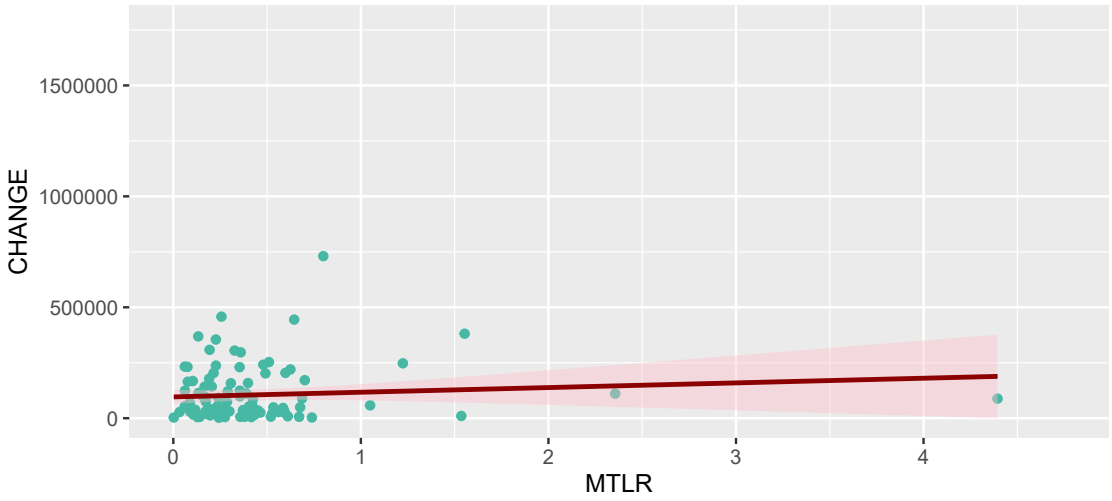


Figura 3.15. Scatter plot della regressione lineare tra le metriche CHANGE e MTLR.

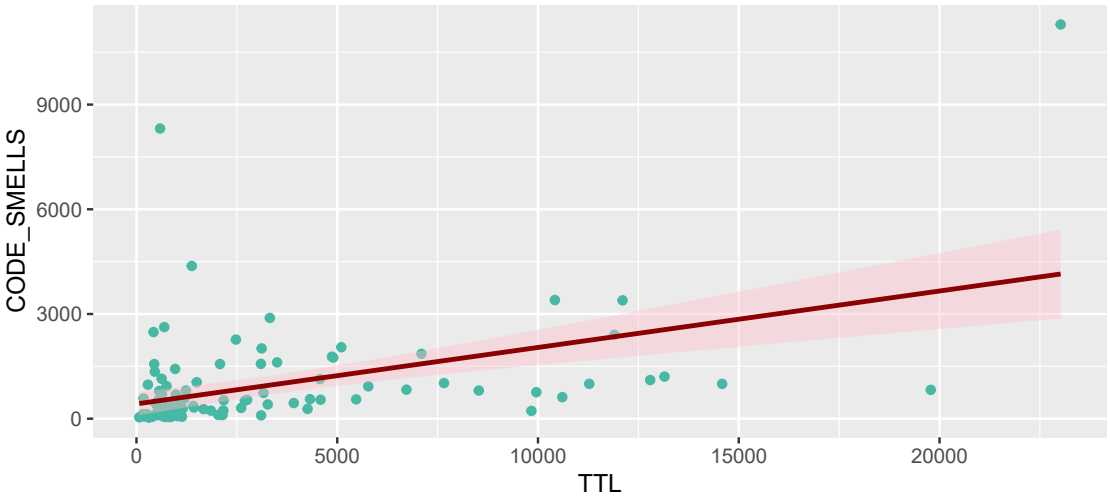


Figura 3.16. Scatter plot della regressione lineare tra le metriche CODE SMELLS e TTL.

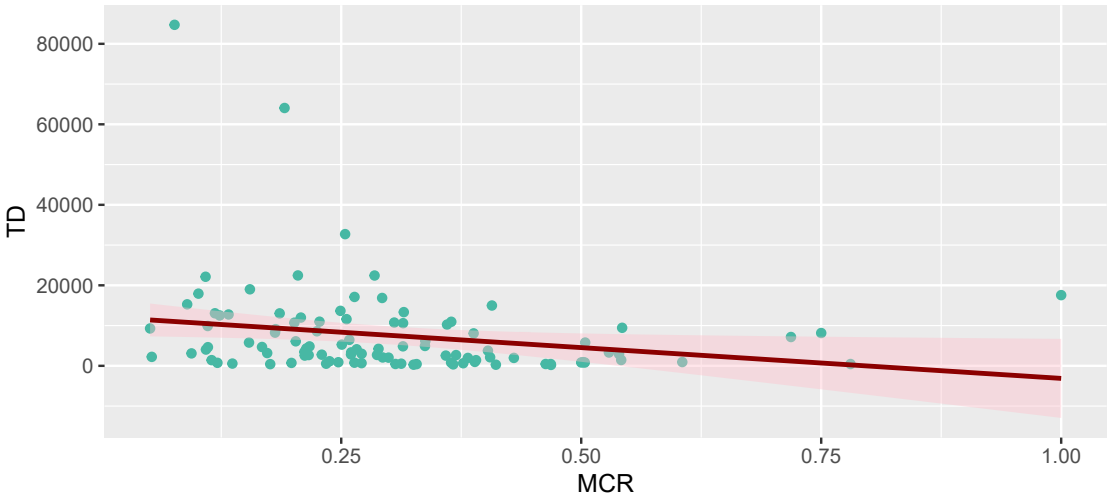


Figura 3.17. Scatter plot della regressione lineare tra le metriche TD e MCR.

relazione lineare semplice: al diminuire del valore dell'indice di manutenibilità e cioè al diminuire della manutenibilità del programma, il valore della metrica MCR, che sarebbe il numero di classi di test modificate versione per versione, aumenta fino ad arrivare al suo massimo per definizione. Inoltre la retta che governa questa relazione è $y = 12000 - 15000x$. Da questi risultati possiamo dire che anche per questa regressione lineare, la qualità del codice di un programma influenza la fragilità dei

test, infatti al diminuire del *Technical Debt* cioè all'aumentare della qualità del software, le modifiche e la fragilità dei test aumentano.

3.6.2 Altre relazioni

Oltre agli *scatter plot* presentati nel paragrafo precedente, per le altre relazioni evidenziate nella Tabella 3.3 si è deciso di non fornire i grafici ma solo di riportare una piccola spiegazione per brevità di presentazione.

Partendo dalle metriche di fragilità dei test NTC e TTL, risulta molto intuitivo dare una spiegazione alla possibile relazione con le metriche di qualità CLOC, LOC, NOC, NOF, NOM e STAT: è ovvio come all'aumentare delle metriche di manutenibilità elencate sopra, il numero di classi di test e il relativo numero di righe di codice subiscano un incremento dovuto all'aumentare della dimensione generale del progetto GitHub analizzato, che richiede più classi di test o più in generale più righe di codice per testare l'intera applicazione.

Per quanto riguarda la metrica MRTL, dalla Tabella 3.3 si può notare come tutti i *p value* delle regressioni lineari riguardanti tutte le metriche di qualità di codice e la metrica di fragilità MRTL sono al di sotto del valore minimo per decretare una relazione lineare. Questo significa che esiste una correlazione tra le metriche di manutenibilità analizzate e la metrica MRTL: più il codice sorgente di un programma è di qualità, più il valore di tale metrica di fragilità mostrerà un valore più basso.

Allo stesso modo per le metriche MCR e MCMMR che riguardano le modifiche di classi di test e dei loro metodi, sono diversi i *p value* molto bassi che stanno ad indicare una relazione lineare tra le metriche. Questo è dovuto dal fatto che un codice di qualità solitamente porta a dei valori di MCR e MCMMR più bassi in quanto le modifiche alle classi di test saranno inferiori rispetto ad un programma di scarsa qualità.

Invece per le metriche TLR, MTLR, MMR, MRR e TSV, dalla Tabella 3.3 non risultano presenti delle relazioni lineari con le metriche di qualità del codice, sono pochi infatti i valori dei *p value* al di sotto della soglia minima di 0.05. Questo significa che per queste metriche non è stato possibile verificare una correlazione dalle osservazioni effettuate.

Capitolo 4

Conclusioni

Il lavoro descritto nella seguente tesi è incentrato su due aspetti principali che riguardano lo sviluppo del software di tipo Android. La prima si basa su una delle parti fondamentali del ciclo di vita di un software, la fase di *testing* e una delle sue caratteristiche principali, la fragilità, la seconda riguarda la qualità e la manutenibilità del codice sorgente sviluppato durante l'evoluzione dell'applicazione.

Dopo aver dimostrato e verificato la veridicità degli studi fatti in precedenza su tali tematiche, l'obiettivo principale della tesi è stato trovare una correlazione tra la qualità del codice sorgente e la fragilità delle suite di test.

4.1 Limitazioni del lavoro di tesi

Nello svolgimento di questo studio è chiaro come ci siano state delle scelte che hanno portato ad un ridimensionamento della quantità di lavoro per far si che tutto ciò potesse restare all'interno di una tesi.

Dunque è stato necessario porre due tipologie di limitazioni:

- *Limitazioni di generalizzabilità*: lo script che applica le metriche riportate in questo studio ha analizzato 101 progetti che appartengono alla famiglia di applicazioni di tipo Android e che sono presenti all'interno di GitHub. Non è possibile stabilire che le conclusioni di questo lavoro di tesi riguardino le applicazioni appartenenti a tutte le altre famiglie di software esistenti.

- *Possibili limitazioni di tipo concettuale*: le metriche che sono state analizzate potrebbero non essere le più adatte per identificare una possibile correlazione

presente tra le due tipologie, qualità di codice e fragilità dei test, in quanto, soprattutto le metriche del secondo tipo, sono state presentate in letteratura da pochi anni e di conseguenza non si è certi della loro validità per questo tipo di studio.

4.2 Riassunto dei risultati

I risultati che sono stati ampiamente discussi nel Capitolo precedente evidenziano la distribuzione dei valori delle metriche di qualità del codice sorgente e le metriche di fragilità delle suite di test e la presenza di una correlazione parziale tra i due tipi di metriche analizzati.

La correlazione parziale è dimostrata soprattutto dalla Tabella 3.3 che presenta in grassetto le varie relazioni lineari presenti tra le due tipologie di metriche. Si può benissimo notare come circa la metà delle regressioni lineari semplici effettuate portino ad una correlazione tra le metriche e ciò sta ad indicare come le metriche di qualità del codice sorgente influenzino le metriche di fragilità dei test.

Inoltre gli *scatter plot* di esempio evidenziano come ogni singola relazione lineare si possa rappresentare e dimostrare anche attraverso dei grafici.

Dalla Tabella 3.3 si può notare la metrica di complessità CC correli con quasi tutte le metriche di fragilità: questo sta ad indicare che un'alterazione della complessità del codice sorgente del software porterà i test ad essere più fragili, in quanto necessiterebbero di un numero maggiore di modifiche.

Stessa cosa si evince per quanto riguarda le metriche di grandezza del progetto NOC, NOF, NOM e STAT, che influenzano diverse metriche di fragilità tra cui NTC, TTL, MRTL, MCR e MCMMR e anche qui il significato è molto semplice: una variazione delle metriche di tipo *Size* di qualità del codice, porterebbe ad una alterazione per quanto riguarda le metriche di fragilità dei test coinvolte.

Infine anche le metriche di stabilità CHANGE, CODE SMELLS e TD correlano con le metriche di fragilità NTC, TTL, MTLR, MRTL e MCMMR evidenziando come in questo caso valori più alti di tali metriche, che stanno ed evidenziare una qualità del codice minore, porterebbero le suite di test ad essere più fragili e di conseguenza a subire più modifiche versione per versione.

Un'ultima osservazione va fatta per le due metriche di fragilità MRR e TSV che risultano essere le due uniche metriche di fragilità che non presentano nessuna correlazione con quelle di manutenibilità. Questo potrebbe essere dovuto dal fatto

che si tratta di metriche che non vengono calcolate versione per versione ma che valgono per l'intero progetto, influenzate ovviamente dall'andamento delle suite di test durante tutta l'evoluzione dell'applicazione.

4.3 Lavori futuri

Di seguito sono elencati alcuni punti di miglioramento per poter estendere e perfezionare questo studio. Si prevede di sviluppare tali migliorie anche per limare le questioni relative alla generalizzabilità e alle limitazioni concettuali analizzate in precedenza.

- Per ampliare la veridicità di questo lavoro si considera la necessità di analizzare applicazioni di diverse famiglie e di diversi domini in modo tale da capire se i risultati ottenuti sono validi per tutte le tipologie di applicazioni.
- Per rafforzare questo studio sarebbe interessante fare un'analisi più approfondita sulle metriche di fragilità e sulla loro validità per quanto riguarda l'effetto provocato dalle metriche di qualità del codice sorgente sulle metriche riguardanti i test.
- Infine si prevede di utilizzare strumenti più raffinati per quanto riguarda l'analisi della correlazione tra le due metriche. Tra i tanti l'attenzione dovrebbe cadere su due in particolare: la *regressione lineare multipla*, che altro non è che una regressione lineare semplice ma con la possibilità di includere due o più variabili esplicative per verificare contemporaneamente l'effetto sulla variabile dipendente; la *stepwise regression*, dove la regressione avviene secondo una scelta delle variabili indipendenti che viene effettuata attraverso una procedura automatica e in ogni passo viene aggiunta o sottratta da quelle esplicative.

Appendice A

Progetti GitHub selezionati

La tabella A.1 elenca i vari progetti che sono stati selezionati per l'esperimento di questo lavoro di tesi insieme alle loro caratteristiche *NTR*, *NTC* e *Data dell'ultimo aggiornamento*.

Tabella A.1: Set progetti Github.

Progetto GitHub	**NTR**	**NTC**	**Ultimo update**
Adyen/adyen-android	59	27	2021-02-10
akvo/akvo-caddisfly	34	117	2020-11-15
alibaba/atlas	83	35	2020-09-23
AlphaWallet/alpha-wallet-android	29	24	2021-01-30
andrzejchm/RESTMock	21	12	2020-04-29
AntennaPod/AntennaPod	95	72	2021-02-07
apache/cordova-android	139	25	2021-01-19
ArcBlock/arcblock-android-sdk	41	9	2020-05-18
asksven/BetterBatteryStats	142	13	2019-10-26
auth0/Auth0.Android	55	85	2020-12-21
Automattic/simplenote-android	105	9	2021-02-08
AzureAD/azure-activedirectory-library-for-android	60	54	2021-02-05
BaseballCardTracker/bbct	51	61	2020-05-10
blabbertabber/blabbertabber	21	10	2020-06-14
bluesnap/bluesnap-android-int	31	80	2020-08-19
bookdash/bookdash-android-app	24	18	2020-01-15
bperel/WhatTheDuck	56	10	2021-01-26

Continua nella prossima pagina

Tabella A.1 – *Continua dalla pagina precedente*

Progetto GitHub	**NTR**	**NTC**	**Ultimo update**
braintree/braintree_android	128	168	2021-02-04
ByteWelder/Spork	38	123	2020-09-24
cemrich/zapp	49	10	2021-01-23
checkout/frames-android	14	18	2020-11-02
citiususc/calendula	28	23	2020-02-03
codinguser/gnucash-android	117	39	2020-12-02
cohenadair/anglers-log	38	13	2020-10-24
concretesolutions/canarinho	13	13	2021-02-02
dhis2/dhis2-android-capture-app	26	182	2021-02-02
domjos1994/SchoolTools	16	14	2020-09-28
drakeet/MultiType	47	14	2021-01-03
edx/edx-app-android	75	126	2021-02-09
eidottermihi/rpicheck	29	15	2020-09-03
erdo/android-fore	21	68	2021-01-23
evant/binding-collection-adapter	16	16	2020-08-14
fabioCollini/DaggerMock	23	183	2020-10-07
facebook/fresco	45	223	2021-02-10
federicoiosue/Omni-Notes	127	43	2021-01-24
forrestguice/SuntimesWidget	71	21	2021-02-10
fuzz-productions/CutoutViewIndicator	13	22	2019-10-15
helloworld1/AnyMemo	22	38	2020-10-03
Ifsttar/NoiseCapture	42	9	2020-07-03
igrek51/android-songbook	29	26	2020-03-29
iSoron/uhabits	37	130	2021-02-09
Iterable/iterable-android-sdk	61	45	2021-02-09
jie-meng/UtilDroid	13	9	2019-04-25
johnjohndoe/Umweltzone	37	20	2020-12-11
jpush/aurora-imui	71	11	2019-07-18
kiwix/kiwix-android	27	99	2021-02-01
kontalk/androidclient	99	24	2020-11-19
leapcode/bitmask_android	43	37	2021-01-25
lexica/lexica	28	16	2021-02-09
linkedin/test-butler	14	10	2021-02-10
material-components/material-components-android	38	166	2021-02-04

Continua nella prossima pagina

Tabella A.1 – *Continua dalla pagina precedente*

Progetto GitHub	**NTR**	**NTC**	**Ultimo update**
maxirosson/jdroid-android	23	38	2020-12-01
medic/medic-gateway	51	20	2020-10-19
Microsoft/AppCenter-SDK-Android	62	167	2021-01-26
miguelbcr/RxPaparazzo	47	8	2019-03-09
MinterTeam/minter-android-wallet	40	17	2021-01-22
mobgen/halo-android	16	181	2020-05-08
MobileTribe/pandroid	41	11	2019-08-01
mozilla-mobile/focus-android	136	62	2021-02-03
natario1/CameraView	41	65	2021-01-28
nearit/Android-UI-Bindings	70	20	2020-03-03
omise/omise-android	22	37	2021-01-20
open-keychain/open-keychain	94	50	2021-02-03
optimizely/android-sdk	61	49	2021-02-10
orgzly/orgzly-android	100	59	2021-01-09
owncloud/android	113	117	2021-02-10
particle-iot/photon-tinker-android	23	29	2021-01-06
patrickfav/under-the-hood	15	19	2020-11-29
projectbuendia/client	60	43	2020-01-12
quran/quran_android	130	24	2021-02-07
RandoApp/Rando-android	42	14	2019-09-10
Sage-Bionetworks/mPower-2-Android	48	18	2020-09-16
schaal/ocreader	56	9	2021-02-04
sewerk/Bill-Calculator	32	65	2019-03-23
sjwall/MaterialTapTargetPrompt	61	31	2020-12-17
SpaceMadness/lunar-unity-console	50	42	2020-09-04
splitwise/TokenAutoComplete	22	10	2021-02-09
SpryServers/sprycloud-android	13	99	2020-01-16
square/leakcanary	29	65	2021-02-01
SRGSSR/SRGMediaPlayer-Android	30	21	2020-09-09
stantmob/card-show-taken-pictures-view	58	8	2020-10-12
stealthcopter/AndroidNetworkTools	21	10	2021-01-30
stuxo/REDAndroid	15	19	2020-01-16
suragch/mongol-library	60	15	2020-07-15
surespot/android	22	9	2019-04-26

Continua nella prossima pagina

Tabella A.1 – *Continua dalla pagina precedente*

Progetto GitHub	**NTR**	**NTC**	**Ultimo update**
tanrabad/survey	54	137	2019-06-08
tarek360/Instacapture	15	9	2019-03-04
TheCacophonyProject/cacophonometer	42	16	2020-12-01
thefuntasty/infinity	21	23	2020-02-11
tunjid/android-bootstrap	56	25	2021-02-10
unfoldingWord-dev/ts-android	69	18	2019-07-26
vase4kin/TeamCityApp	65	114	2020-12-20
veritrans/veritrans-android	99	36	2021-01-22
VoIPGRID/vialer-android	41	17	2019-11-28
VREMSoftwareDevelopment/WiFiAnalyzer	19	138	2020-12-23
wayfair/brickkit-android	81	31	2021-02-10
Webtrekk/webtrekk-android-sdk	31	75	2020-04-21
WycliffeAssociates/translationRecorder	32	27	2019-08-23
xdtianyu/CallerInfo	68	9	2020-10-20
xlagunas/AndRTC	23	18	2020-05-21
zulip/zulip-android	45	8	2019-01-15

Bibliografia

Aggarwal, K. K., Singh, Y., & Chhabra, J. K. (2002). An integrated measure of software maintainability. In *Annual reliability and maintainability symposium. 2002 proceedings (cat. no.02ch37318)* (pp. 235–241).

Alégroth, E., Feldt, R., & Ryrholm, L. (2015). Visual gui testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, *20*, 694-–744.

Antonellis, P., Dimitris, A., Kanellopoulos, Y., Makris, C., Theodoridis, E., Tjortjis, C., & Tsirakis, N. (2007). A data mining methodology for evaluating maintainability according to iso/iec-9126 software engineering–product quality standard. In (pp. 1–11).

Ardito, L., Coppola, R., Barbato, L., & Verga, D. (2020). A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, *2020*, 1–26.

Baars, S., & Meester, S. (2019). Codearena: Inspecting and improving code quality metrics using minecraft. In *Proceedings of the second international conference on technical debt* (p. 68—70). IEEE Press.

Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., & Gyimóthy, T. (2011). A probabilistic software quality model. In *2011 27th ieee international conference on software maintenance (icsm)* (pp. 243–252).

Bauer, V., Heinemann, L., Hummel, B., Juergens, E., & Conradt, M. (2012). A framework for incremental quality analysis of large software systems. In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 537–546).

Brett, D., Tihomir, G., & Darko, M. (2010). On test repair using symbolic execution. In *Proceedings of the 19th international symposium on software testing and analysis* (p. 207-–218). Association for Computing Machinery.

Coppola, R., Morisio, M., Torchiano, M., & Ardito, L. (2019). Scripted gui testing of android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering*, *24*, 3205-–3248.

Cruz, L., Abreu, R., & Lo, D. (2019). To the attention of mobile software developers: Guess what, test your app! *Empirical Software Engineering*, *24*, 2438—2468.

Cunningham, W. (1992). The wycash portfolio management system. In *Addendum to the proceedings on object-oriented programming systems, languages, and applications (addendum)* (p. 29-–30). Association for Computing Machinery.

di Biase, M., Rastogi, A., Bruntink, M., & van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. In *2019 ieee/acm international conference on technical debt (techdebt)* (pp. 113–122).

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (p. 50-–60). Association for Computing Machinery.

Gao, Z., Chen, Z., Zou, Y., & Memon, A. M. (2016). Sitar: Gui test script repair. *IEEE Transactions on Software Engineering*, *42*, 170–186.

Garousi, V., & Felderer, M. (2016). Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, *33*, 68–75.

Geay, E., Yahav, E., & Fink, S. (2006). Continuous code-quality assurance with safe. In *Proceedings of the 2006 acm sigplan symposium on partial evaluation and semantics-based program manipulation* (p. 145—149). Association for Computing Machinery.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (quatic 2007)* (p. 30-39).

Jiang, L., Rewcastle, R., Denny, P., & Tempero, E. (2020). Comparecfg: Providing visual feedback on code quality using control flow graphs. In *Proceedings of the 2020 acm conference on innovation and technology in computer science education* (p. 493-–499). Association for Computing Machinery.

Kothapalli, C., Ganesh, S. G., Singh, H. K., Radhika, D. V., Rajaram, T., Ravikanth, K., ... Rao, K. (2011). Continual monitoring of code quality. In *Proceedings of the 4th india software engineering conference* (p. 175-–184).

Association for Computing Machinery.

Krishnan, R., Krishna, S. M., & Bharill, N. (2007). Code quality tools: Learning from our experience. *SIGSOFT Softw. Eng. Notes*, *32*(4), 5–es.

Letouzey, J. (2012). The sqale method for evaluating technical debt. In *2012 third international workshop on managing technical debt (mtd)* (pp. 31–36).

Linares-Vásquez, M. (2015). Enabling testing of android apps. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 763–765).

Linares-Vásquez, M., Moran, K., & Poshyvanyk, D. (2017). Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 ieee international conference on software maintenance and evolution (icsme)* (pp. 399–410).

Mateen, A., Zhu, Q., & Afsar, S. (2018). Comparitive analysis of manual vs automotive testing for software quality. In *Proceedings of the 7th international conference on software engineering and new technologies (icsent'18)* (pp. 1–7).

Ostberg, J.-P., & Wagner, S. (2014, 10). On automatically collectable metrics for software maintainability evaluation. In (p. 32-37).

Pirrigheddu, S. (2021, 2). progetti_analizzati.csv.. Retrieved from `https://figshare.com/articles/thesis/progetti_analizzati_csv/14035802` doi: 10.6084/m9.figshare.14035802.v1

Sarwar, M., Tanveer, W., Sarwar, I., & Mahmood, W. (2008). A comparative study of mi tools: Defining the roadmap to mi tools standardization. In *2008 ieee international multitopic conference* (p. 379-385).

Stocco, A., Yandrapally, R., & Mesbah, A. (2018). Visual web test repair. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 503-–514). Association for Computing Machinery.

Strecansky, P., Chren, S., & Rossi, B. (2020). Comparing maintainability index, sig method, and sqale for technical debt identification. *Scientific Programming*, *2020*, 1–14.

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., ... Streit, J. (2012). The quamoco product quality modelling and assessment approach. In *2012 34th international conference on software engineering (icse)* (pp. 1133–1142).

Yusifoglu, V. G., Amannejad, Y., & Can, A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, *58*, 123–147.