Università degli Studi di Salerno

Dipartimento di Informatica
Corso di Laurea Magistrale in Informatica

Flaky test: mapping study della letteratura e sviluppo di un tool per l'individuazione dei flaky test

Relatori:
Ch.ma Prof.ssa Filomena FERRUCCI
Ch.mo Dott. Pasquale SALZA
Ch.mo Dott. Valerio TERRAGNI

Candiato:
Giammaria Giordano
Matr.: 0522500509

ANNO ACCADEMICO 2018/2019

Indice

Elenco delle figure

Introduzione

Continuous integration e flaky test

Con il propagarsi delle metodologie *Agile*, nelle grandi multinazionali in cui si sviluppano migliaia di righe di codice ogni giorno, si è reso necessario adottare tecniche di *continuous integration*, le quali permettono agli sviluppatori di aggiungere nuove funzionalità ad un sistema software già esistente, per poi eseguire in automatico l'intera suite di test per verificare che le nuove funzionalità non abbiano introdotto *bug* all'interno del software. Se durante questa fase non viene riscontrato il fallimento di nessun caso di test, il software potrà essere reso di nuovo disponibile per ulteriori modifiche, in caso contrario invece bisognerà individuare il difetto introdotto e correggerlo. Non sempre però il fallimento di un caso di test è causato dell'introduzione di nuovi *bug* all'interno del software, infatti questo può derivare dalla presenza di un metodo *flaky* presente all'interno della suite di test.

Un test si definisce *flaky* se il suo comportamento non può essere stabilito in **maniera deterministica**. Appare quindi evidente che la presenza di un metodo *flaky* all'interno della test suite possa incidere non solo sulla qualità, ma anche sui tempi di rilascio e sui costi del software. Inoltre, analisi condotte in diversi studi focalizzati sui *flaky test*, hanno mostrato che una delle criticità più frequenti quando si manifesta un *flaky* è capire che cosa l'ha scatenato.

L'obiettivo che ci si pone all'interno di questa tesi è lo sviluppo di un nuovo tool che possa aiutare gli sviluppatori ad identificare la presenza di *flaky* all'interno dei loro progetti e a individuare la sua *root cause*. Infatti, definire la *root cause* potrebbe rappresentare il primo passo per riuscire a mitigare la problematica dei *flaky*. Il tool sviluppato è in grado di individuare *flaky* test, la cui *root cause* non è la dipendenza dall'ordine di esecuzione, all'interno di progetti scritti in **Java** e che fanno uso di **Maven** come building system. Il tool proposto esegue un certo numero di volte uno specifico caso di test, salvando non solo il risultato ottenuto dall'esecuzione (pass o fail), ma anche delle informazioni aggiuntive legate allo stato della macchina su cui si sta eseguendo il test e il tempo di esecuzione del singolo caso di test.

Queste due informazioni sono state aggiunte poiché non venivano riportate in nessuno dei dataset precedentemente rilasciati, ma possono essere utili per fare delle analisi a grana fine per l'individuazione delle *root cause* e di eventuali pattern. Infine, oltre alla generazione di un file csv con all'interno le informazioni descritte in precedenza, viene anche generato un file aggiuntivo in cui si va a tenere traccia dell'esito della build di ogni progetto analizzato.

Per valutare questo tool è stato considerato il dataset rilasciato dagli sviluppatori di **iDFlakies**, un framework in grado di individuare un *flaky test* e fare una classificazione parziale delle *root cause* per i *flaky* individuati (indipendente dall'ordine di esecuzione e dipendente dall'ordine di esecuzione). Da questo dataset sono state selezionate centosessantasei righe, ognuna rappresentante un caso di test *flaky* la cui *root cause* era diversa dalla dipendenza dall'ordine di esecuzione del test.

Infine, per ognuno dei metodi *flaky* rilevati da questo tool (trentuno test) è stata fatta un'analisi statica del codice sorgente per identificare la *root cause*. Da questa analisi è emerso che per trenta *flaky test* la *root cause*era riconducibile a problemi di **network**, mentre in un solo caso è stata identificata come root cause il **multithreading**.

Una volta terminata l'analisi, si è passati ad immagazzinare i dati ottenuti all'interno di una base di dati ed a creare dei grafici sui risultati delle singole esecuzioni di un caso di test, in modo da evidenziare possibili pattern. Sono quindi emersi in alcuni casi dei comportamenti "deterministici" del *flaky test* che dovranno essere ulteriormente approfonditi.

La tesi è organizzata nel seguente modo.

Nel primo capitolo viene presentata una panoramica generale del testing, soffermandosi in particolare sul testing di regressione, sui sistemi di build e sulla *continuous integration*.

Nel secondo capitolo viene presentata un mapping study sistematico della letteratura dei *flaky test*, che ha permesso di comprendere a fondo lo stato dell'arte e di identificare gli approcci che sono stati utilizzati in questi anni per far fronte a questa problematica.

Nel terzo capitolo è presentata la costruzione di un tool che possa essere di supporto agli sviluppatori per aiutarli ad individuare metodi *"flaky"* all'interno dei loro progetti. Tale tool va ad estendere un dataset già presente in letteratura aggiungendo nuove informazioni che potrebbero essere d'interesse per eventuali ricerche future.

Le conclusioni riassumono i risultati ottenuti e contengono gli sviluppi futuri che potranno essere messi in pratica per ampliare la conoscenza sui *flaky test*.

1. CAPITOLO 1

1.1. Il testing

Come avviene per qualsiasi oggetto, materiale o immateriale, anche il processo di sviluppo di un prodotto software può essere suddiviso e schematizzato in una serie di step o fasi. Nella prima fase è necessario individuare e fissare i requisiti che il sistema software dovrà avere, a tale scopo può essere effettuata un'intervista al cliente da parte del project manager per concordare tali fondamentali caratteristiche; successivamente, il team di sviluppo è incaricato di individuare le tecnologie più efficienti per la realizzazione del sistema ed infine, si passa alla fase di implementazione e collaudo; quest'ultimo, definito in linguaggio tecnico testing, è il procedimento che viene utilizzato per individuare le differenze tra il comportamento atteso del sistema e quello che invece viene effettivamente osservato nel sistema software sviluppato.

Tra i vari tipi di testing, è possibile citare i seguenti come esempio:

- Testing unitario;
- Testing di integrazione;
- Testing di accettazione.

Essi vengono applicati in successione per effettuare un collaudo del software a diversi livelli.

1.1.1. **Testing unitario**

Il testing unitario viene utilizzato per individuare possibili differenze tra le specifiche di un particolare modulo del sistema e la sua realizzazione. In questa fase vengono considerati piccoli *snippet* di codice denominati moduli e vengono testati in maniera indipendente tra loro. Con questa tipologia di testing è possibile individuare eventuali fault presenti all'interno del modulo stesso.

1.1.2. **Testing di integrazione**

Dopo aver terminato la fase di testing unitario, lo step successivo prevede l'applicazione del testing di integrazione. In questo caso vengono integrate tra loro tutte le varie componenti create durante la fase di testing unitario e si valuta il comportamento di un determinato modulo nel momento in cui interagisce con un altro.

Esistono diverse strategie per definire l'ordine di integrazione delle varie componenti:

- Big Bang: Tutte le componenti vengono integrate contemporaneamente e poi si valuta l'effettiva interazione tra i vari moduli;
- Bottom up: Si testano in maniera consequenziale le varie componenti a partire dai layer più bassi fino ad arrivare a quelli più alti;
- Top down: Si comincia testando per prime le componenti dei layer superiori, in ordine inverso rispetto a quanto accade nel bottom up;
- Sandwich: Combina la strategia del testing Bottom up e quella Top down ovvero un team di lavoro inizia a testare le componenti dai layer più bassi a quelli più alti mentre un altro team testa i vari moduli dai layer più alti aquelli più bassi.

1.1.3. Testing di accettazione

Il testing di accettazione solitamente implica l'esecuzione di un insieme di casi di test. Ogni caso di test ha lo scopo di sollecitare una particolare funzionalità del sistema.

L'esecuzione di ogni caso di test viene effettuata direttamente nell'ambiente operativo del cliente finale o in alternativa in un ambiente simulato[1]. Il test di accettazione può essere eseguito sia dagli sviluppatori del sistema, che dal cliente finale, prima di mettere in esercizio il sistema sviluppato. Due esempi di testing di accettazione possono essere i seguenti:

- Alfa testing: Viene testato il sistema da un numero ristretto di utenti e in un ambiente controllato;
- Beta testing: Viene testato il sistema all'interno dell'ambiente in cui sarà messo in esercizio.

1.2. Metodologia Agile

La metodologia *Agile* prevede lo sviluppo di piccoli "incrementi" di funzionalità che solitamente sono rilasciati al cliente ogni due o tre settimane denominati *sprint*. Nello sviluppo di prodotti software tramite la metodologia *Agile* è fondamentale una forte interazione con il cliente, così da ottenere velocemente i feedback necessari alla corretta e pronta soddisfazione di tutte le sue esigenze. Questa metodologia è stata ideata e sviluppata per l'applicazione a team i cui membri, pur lavorando per lunghi periodi di tempo allo sviluppo dello stesso prodotto software, esercitano la loro attività in aree anche geograficamente distanti tra loro. Gli approcci con metodologie *Agile* vanno a diminuire

[1]Con il termine "ambiente simulato" si intende un ambiente del tutto uguale a quello in cui verrà rilasciato il prodotto software. Tali simulazioni risultano necessarie nel momento in cui si può utilizzare il reale ambiente di lavoro.

l'*overhead*[2] di sviluppo software, in quanto non prevedono la produzione di gran parte della documentazione che veniva elaborata nel modello a cascata durante la fase di pianificazione, ma si produce documentazione secondo il principio del *"documentare solo quando è necessario"*. Ciò ha permesso ai programmatori di concentrarsi sullo sviluppo del software piuttosto che sulla fase di progettazione e documentazione dello stesso.

1.3. Configuration Management

I sistemi software sono soggetti a costanti cambiamenti, sia nella fase di sviluppo che nella fase di rilascio. Spesso essi possono subire cambiamenti sia in risposta a nuovi requisiti, sia in risposta all'introduzione di nuove tecnologie hardware. Molti sistemi software possono quindi essere considerati come un insieme di versioni differenti, ognuna delle quali dovrà essere mantenuta e gestita opportunamente. Il *Configuration Management* (CM) racchiude un insieme di policies, processi e tools necessari per gestire gli eventuali cambiamenti del sistema. Si tratta di uno strumento software che ha le seguenti caratteristiche:

1. *Version Control*: Tiene traccia delle molteplici versioni esistenti dello stesso prodotto software sviluppate da diversi programmatori gestendo eventuali conflitti tra le varie versioni;

2. *System building*: Viene definito come avviene il processo di unione delle varie componenti del sistema software. Vengono eseguiti i link delle varie librerie e infine viene compilato il codice sorgente;

3. *Change management*: Tiene traccia delle varie richieste di cambiamenti da parte del cliente finale o da parte degli altri sviluppatori;

[2]Il termine overhead indica le risorse accessorie che sono richieste per completare un particolare task.

4. *Release management*: Tiene traccia di tutte le versioni che sono state rilasciate al cliente finale.

1.3.1. Version Control

I sistemi di Version Control (VC) identificano, immagazzinano e controllano gli accessi delle differenti versioni del sistema software. Esistono attualmente due tipologie di Version Control:

1. *Centralized System*: Consiste in un singolo repository principale che mantiene tutte le versioni software che sono state sviluppate;

2. *Distributed System*: Molteplici versioni di un singolo repository coesistono contemporaneamente.

1.3.2. System building

Attualmente, molti prodotti software sono dotati di sistemi automatici per effettuare la build (ovvero la fase di compilazione, di link ad eventuali librerie, recupero di file di configurazione, etc.). Questo approccio viene utilizzato molto frequentemente, soprattutto in ambito di grandi progetti open-source, dove potenzialmente possono collaborare sviluppatori provenienti da ogni parte del mondo, e che quindi hanno bisogno di un meccanismo semplice per riuscire a configurare tutto quello di cui c'è bisogno per avviare il prodotto software.

Gran parte dei sistemi di build automatici attualmente utilizzati includono le seguenti caratteristiche:

- *Build script generation*: Il sistema di build analizza il programma, identificando tutte le dipendenze delle varie componenti, e genera automaticamente un file di build chiamato *config file*;

- *Version control system integration*: Il sistema di build deve controllare che le versioni richieste delle varie componenti siano disponibili, in caso contrario deve scaricarle;

- *Minimal recompilation*: Il sistema di build deve individuare quale parte del codice sorgente è stata modificata, in maniera tale da poter compilare solo quel modulo, evitando quindi la ricompilazione dell'intero prodotto software ogni volta che avviene una modifica;

- *Executable system creation*: Il sistema di build deve effettuare il link delle librerie, compilare il codice sorgente e generare il codice oggetto richiesto per poter eseguire il sistema;

- *Test automation*: I sistemi di build più moderni dovrebbero generalmente essere in grado di lanciare in automatico tutti i casi di test presenti all'interno del sistema, ed indicare se ci sono eventuali problemi. Se viene riscontrato un problema durante questa fase, l'esecuzione della build passa in stato di "fail" e la build viene definita *broken*;

- *Reporting*: I sistemi di build devono effettuare un report dell'esecuzione ed indicare se questa è avvenuta con successo oppure no (pass or fail);

- *Documentation generation*: Alcuni sistemi di build possono generare anche automaticamente della documentazione aggiuntiva che può essere utilizzata in seguito come indicazione per gli sviluppatori. Tale documentazione può essere ricavata dai commenti presenti all'interno del codice sorgente.

1.3.3. **Change management**

Ogni sistema software può essere soggetto a cambiamenti. Le aziende necessitano di adeguati strumenti per garantire la corretta esecuzione di tali modifiche durante tutto il ciclo di vita del software[3]. Un si-

[3]Con il termine "ciclo di vita del software" si intendono tutte le fasi necessarie per realizzare un prodotto software. Queste fasi tipicamente includono l'analisi, la progettazione, la realizzazione, il

stema software può evolversi per diverse ragioni: cambio di requisiti, correzione di bug, evoluzione dell'ambiente etc.

Il Configuration Manager deve quindi gestire eventuali richieste di cambiamento, dovute sia all'introduzione di nuove funzionalità richieste dal cliente, sia per procedere alla correzione di eventuali bug presenti all'interno del sistema.

1.3.4. **Release Management**

Un system release è un sistema software che viene utilizzato per distribuire il prodotto al cliente o al mercato di massa. Essi sono in genere di due tipologie:

- *Major release*: In questo caso vengono aggiunte delle nuove funzionalità al prodotto software;
- *Minor release*: Vengono eseguiti delle modifiche per eliminare bug che sono stati individuati oppure vengono effettuati dei piccoli cambiamenti a funzioni già esistenti.

Solitamente, il prodotto software rilasciato non comprende esclusivamente il codice sorgente, ma viene corredato di altri feature, quali:

- File di configurazione, ovvero dei file dove viene definito come la release deve essere configurata;
- Un programma di supporto per facilitare l'installazione;
- Manuale di installazione;
- Manuale di avvio.

collaudo, la messa a punto, l'installazione ed eventualmente la fase di manutenzione del software (se prevista da contratto).

1.4. Continuous Integration

La *continuous integration* è una pratica comunemente applicata nell'ingegneria del software. Questa tecnica, nata negli anni '90, è stata inizialmente utilizzata solo da poche grandi aziende come, per esempio, Microsoft, Google etc. Con il passare degli anni il suo impiego è andato via via crescendo, grazie alla sua notevole facilità di costruire processi automatizzati per la compilazione, l'analisi statica ed effettuare la fase di test.

Siccome le metodologie *Agile* prevedono l'aggiunta di poche funzionalità in ogni *sprint*, la *continuous integration* viene effettuata ogni qualvolta avviene un cambiamento all'interno del codice sorgente.

Per effettuare in maniera corretta la continuous integration è necessario seguire i seguenti passi:

- Scaricare dal sistema di versioning la versione più recente del sistema software e crearne una copia privata in locale;
- Eseguire la build del sistema, la quale effettuerà in automatico il run di tutti i casi di test presenti all'interno del software per verificare la corretta esecuzione. In caso di fallimento di uno o più casi di test, la build verrà definita "broken" e dovrà essere contattato l'ultimo sviluppatore che ha apportato modifiche per avvisarlo di tale problematica, visto che sarà lui la persona che si dovrà occupare di riportare la build in stato di "pass";
- Effettuare le modifiche al sistema;
- Eseguire nuovamente la build del sistema per verificare la correttezza delle modifiche apportate. Se uno o più test falliscono, reiterare questo step fino a portare la build in stato di "pass";
- Una volta che tutti i test danno esito positivo, si rieseguono all'interno del sistema di build presente all'interno del server per verificarne l'esito;

- Se tutti i test danno esito positivo, i cambiamenti vengono resi effettivi anche all'interno del sistema di versioning e il sistema aggiornato verrà considerato come nuova baseline.

1.5. **Regression Testing**

Lo sviluppo di un software è un processo iterativo. Gli sviluppatori possono aggiungere nel tempo delle nuove funzionalità o migliorare quelle già presenti. Nel momento in cui ciò avviene, occorre sviluppare nuovi test case per verificare la correttezza di quanto implementato e rieseguire tutti i test precedentemente sviluppati per verificare di non aver introdotto nuovi bug all'interno del sistema. I test che vengono rieseguiti all'interno del sistema per riprodurre i failures sono definiti "test di regressione".

Dalla letteratura è possibile desumere diverse strategie per verificare il corretto funzionamento di tutta la *test suite* durante la fase di CI:

- *Retest frequent use cases*: Nel momento in cui si aggiungono nuove funzionalità, occorre verificare che le funzionalità più utilizzate da parte degli utenti continuino a funzionare regolarmente. Per massimizzare le probabilità del corretto funzionamento di queste funzionalità, quello che viene fatto è rieseguire tutti i test case che verificano quella feature;

- *Retest Risky Use Case*: Per ridurre al minimo la probabilità di fault catastrofici, gli sviluppatori si concentrano principalmente nell'esecuzione di casi di test sulle funzionalità che ritengono più critiche;

- *Retest dependent components*: Le componenti che dipendono da una componente modificata devono essere ritestate dopo tale intervento, per garantire il loro corretto funzionamento anche dopo le modifiche.

In ogni caso, è opportuno eseguire i casi di test diverse volte; inoltre, per grandi progetti, si ha bisogno di un numero elevato di casi di test da effettuare. Per questo sono state sviluppate delle infrastrutture per automatizzare tutte le fasi del testing (dalla scrittura del caso di test, fino alla sua esecuzione) che effettuano automaticamente una comparazione tra i risultati ottenuti e quelli predetti da degli oracoli predefiniti.

1.6. **Problemi di building**

Una delle principali difficoltà che si possono avere durante la *continuous integration* è la lentezza nel processo di esecuzione della build del sistema. Tale problematica impatta direttamente sulla produttività degli sviluppatori stessi, che molto spesso impiegano parte del loro tempo ad attendere il completamento di tale fase. Gran parte dell'*overhead* durante il processo di building è dovuto all'esecuzione e alla verifica del corretto risultato di tutti i casi di test presenti all'interno del prodotto. Siccome durante il processo di building è considerevole la quantità di risorse temporali che vengono impiegate nell'esecuzione dei casi di test, negli anni sono state proposte diverse tecniche per diminuire il numero di test case da eseguire. Tra quelle rinvenute in letteratura possiamo citare le seguenti:

- *Test Suite Minimization*: Tale tecnica va a diminuire il numero dei test case presenti all'interno della suite, andando a rimuovere test ridondanti;
- *Esecuzione dei casi di test in parallelo*: Tale tecnica può essere utilizzata nel momento in cui si possiedono un buon numero di macchine sulla quale far eseguire in contemporanea una parte del processo di build.

Tuttavia, i vari approcci che afferiscono alle due tipologie succitate eseguono tipicamente il run dei casi di test sempre nello stesso ordine,

ma, questa assunzione, come illustrato nel successivo paragrafo è non è sempre da considerarsi veritiera.

1.7. Flaky test

I *flaky test* sono dei particolari test case che possono mostrare un risultato di pass o fail in maniera **non deterministica**, ovvero senza che siano stati apportati cambiamenti nel codice da testare.[1] Solitamente, quando durante il test di regressione si ha il fallimento di un test, quest'ultimo indica che è stato introdotto un fault nel codice sorgente; gli sviluppatori quindi procedono ad effettuare la fase di debug per individuare il bug. Tuttavia, in presenza di un test flaky, il fallimento del testing non è per forza indice dell'introduzione di un fault all'interno del sistema.

Durante la fase di *continuous integration* gli sviluppatori dedicano il loro focus nell'aggiungere nuove funzionalità al sistema software, pertanto la presenza di un test "flaky" può portare alla perdita di molte ore nel tentativo di individuare un fault difficile da replicare, data la loro natura non deterministica.

In letteratura, la problematica dei *flaky test* viene studiata da circa quindici anni, ed è possibile evidenziare i seguenti aspetti:

- Un *flaky test* può essere dipendente dall'ordine in cui viene eseguito: Un test di questo tipo si presenta solo nel momento in cui la suite di test viene eseguita in un particolare ordine. Spesso questo tipo di flaky si presentano nel momento in cui il test flaky fa uso di qualche componente o condivide informazioni con un altro modulo;

- Un *flaky test* può essere indipendente dall'ordine: In questo caso ovviamente è un flaky che non dipende dall'ordine in cui viene eseguito. Spesso tali flaky fanno uso di rete, operazioni di input/output, uso di thread, etc.

Negli anni sono state individuate diverse categorie di *root cause*, le più ricorrenti sono[5]:

- *Async Wait*: Questa categoria è caratterizzata da test che effettuano chiamate asincrone senza però attendere il risultato;
- *Concorrenza*: L'uso di multithread all'interno di un caso di test, può generare metodi flaky se i thread non vengono gestiti in modo *safe*;
- *Dipendenza dalla piattaforma*: In questa categoria rientrano tutti i test che hanno un comportamento differente in base alla piattaforma (es.comportamenti differenti su macchine a 32 bit rispetto a macchine a 64 bit);
- *Input/Output*: Le operazioni che fanno uso di Input/Output, possono generare metodi flaky nel momento in cui non vengono adottate le "buone norme" di programmazione (es. effettuare l'operazione di *close* su un file dopo averlo letto);
- *Operazioni con numeri floating point*: Spesso i numeri floating point possono avere dei problemi di rappresentazione in memoria, questo può generare risultati scorretti nel momento in cui vengono utilizzati per effettuare delle operazioni;
- *Random*: La generazione di numeri random può generare metodi di test "flaky" se non sono stati opportunamente definiti e presi in considerazione il limite inferiore e quello superiore dei numeri pseudo-casuali che possono essere generati;
- *Rete*: Le operazioni che prevedono l'uso della rete possono facilmente portare alla "creazione" di metodi "flaky", poiché le risorse della rete sono sempre difficili da controllare e gestire;
- *Tempo*: In questa categoria rientrano tutti i casi di test il cui comportamento non deterministico è dovuto prevalentemente all'uso

di funzioni legate al tempo (errori di fuso orario, rappresentazioni diverse dell'ora, etc.).

Negli anni sono state adottate diverse strategie per tentare di minimizzare la problematica dei flaky test: Google per esempio ha creato il decoratore "@flaky"[1] in modo da poter rieseguire un certo numero di volte tutti i test con successi ad intermittenza. Anche **JUnit** (versione 5) e **Maven** hanno introdotto la possibilità di rieseguire un test case un certo numero di volte[4][5] o di ignorare un particolare metodo poiché ritenuto "flaky". Tali provvedimenti però non risultano ancora sufficienti per riuscire a minimizzare l'impatto economico dovuto alla presenza di *flaky test* all'interno della suite di test. Attualmente, la tecnica più utilizzata per verificare la presenza di flaky all'interno di un progetto software consiste nel fare un rerun del test case finché esso non viene eseguito con successo. Tale tecnica però risulta onerosa a livello economico e talvolta frustrante per gli sviluppatori, i quali possono spendere ore ed ore prima ottenere un cambio di stato da parte del metodo flaky.

1.7.1. Esempio di flaky test

Per chiarire ulteriormente la problematica in questione, possiamo considerare il seguente snippet di codice in figura 1.1:

Il metodo "*CreateGapLease*" apre il file "*/leases/gap*" ed esegue la scrittura del contenuto della variabile "*contract_data*" al suo interno ed infine chiude il file. Il test case "*testCreateGapLease*" invoca la funzione "*CreateGapLease*" e successivamente controlla che il contenuto del file "*/leases/gap*" sia effettivamente uguale al contenuto della variabile "*contract_data*". Ma cosa avviene nel momento in cui il file "*/leases/-gap*" esiste già e contiene già dati al suo interno? In questo caso il test genererà un fail. In questo caso il flaky è dovuto semplicemente ad un errato controllo della precondizione del test, ma in generale errori di questo tipo possono capitare nei casi più disparati (es. in ambito

```
def CreateGapLease(self):
  data_file = open('/leases/gap', 'w+')
  data_file.write(contract_data)
  data_file.close()

def testCreateGapLease(self):
  contract_writer.CreateGapLease()
  self.assertEqual(ReadFileContents('/leases/gap'),
        contract_data)
```

Figura 1.1: *Esempio flaky test*

di concorrenza, network etc.). Una prima strategia di fix può essere quella di verificare l'effettiva esistenza del file "*/leases/gap*" (eliminandolo in caso affermativo). La figura 1.2 mostra il codice del metodo "testCreateGapLease" dopo la modifica.

```
def testCreateGapLease(self):
  if os.path.exists(lease_file):
    RemoveFile(lease_file)
  ...
```

Figura 1.2: *Snippet di codice modificato*

Tuttavia, neanche questa modifica è sufficiente per garantire la corretta esecuzione del caso di test in quanto, se "*/leases/gap*" ha un

path di tipo NFS e può essere scritto da un altro test, il metodo "*testCreateGapLease*" può ancora fallire improvvisamente. La soluzione in questo caso consiste nell'effettuare alcune piccole modifiche al codice del metodo "*CreateGapLease*" per rendere unica la risorsa di cui ha bisogno.

La figura 1.3 mostra il codice del metodo "*CreateGapLease*" dopo aver effettuato le opportune modifiche.

```
def CreateGapLease(self, lease_path=None):
  if lease_path is None:
    lease_path = '/leases/gap'
  ...
```

Figura 1.3: *Codice modificato*

La chiamata al metodo "CreateGapLease" verrà effettuata nello stesso modo illustrato precedentemente, ma il test potrà passargli un path differente. Tale strategia impedirà fallimenti ad intermittenza del test case.

La figura 1.4 mostra il codice del metodo nella sua versione finale[6].

```
def testCreateGapLease(self):
  lease_file = os.path.join(FLAGS.test_tmpdir, 'gap')
  contract_writer.CreateGapLease(lease_path=lease_file)
  self.assertEqual(ReadFileContents(lease_file),
         contract_data)
```

Figura 1.4: *Versione finale del codice*

2. CAPITOLO 2

2.1. Mapping

I mapping study sono effettuati per avere un quadro completo delle conoscenze attualmente disponibili in una certa area di ricerca. Il mapping viene eseguito attraverso il conteggio dei contributori in relazione alle categorie che sono state approfondite. Un mapping ha lo scopo di approfondire quali argomenti sono stati trattati e dov'è possibile reperire tali informazioni. Essendo però un processo sistematico, condivide molte delle metodologie di una revisione sistematica della letteratura. Quello che però fa discostare un mapping da una *systematic review* sono gli obiettivi finali: durante una revisione sistematica si punta a sintetizzare delle evidenze in un determinato campo di analisi, nello studio delle mappature sistematiche l'obiettivo è piuttosto definire un'area di ricerca. Per definire un mapping sistematico, **Petersen et al**.[4] propongono di seguire la seguente linea guida:

- Domande di ricerca: Individuare le domande per le quali ricercare una risposta. Tali domande guideranno tutto lo studio;
- Ricerca della documentazione: Individuare le parole chiave più opportune da formulare per effettuare la fase di ricerca;
- Criteri di inclusione/esclusione: Individuare dei criteri per scartare risultati non inerenti allo studio;

- Valutazione della qualità: Valutare i risultati in base alla loro qualità;
- Estrazione dei dati: Sintetizzare i risultati in base alle domande di ricerca che sono state poste;
- Analisi e classificazione: Visualizzare le informazioni di ogni item di ricerca e raggrupparle in base a caratteristiche comuni;
- Validazione: Descrivere eventuali minacce dello studio condotto e valutare la ripetibilità dello studio.

2.2. Mapping study flaky test

2.2.1. Domande di ricerca

La prima fase per effettuare il mapping study legato alla problematica dei *flaky test* è stata l'individuazione delle domande di ricerca. Sono state quindi poste le seguenti domande:

(**RQ0**): È stato rilasciato un dataset?

(**RQ1**): Hanno lavorato anche su progetti closed source?

(**RQ2**): Viene detto in che linguaggio di programmazione sono scritti i progetti utilizzati?

(**RQ3**): Viene descritto che tipo di sistema di building utilizzano?

(**RQ4**): Sono state definite le tecniche che hanno utilizzato per sviluppare il dataset?

(**RQ5**): Vengono definite quali sono le root cause più frequenti che sono state individuate?

(**RQ6**): Viene definito che metodo empirico hanno utilizzato per individuare le root cause?

(**RQ7):** Viene detto a chi è indirizzata questa ricerca?

(**RQ8):** In che anno è stata pubblicata la ricerca?

Dopo aver individuato le domande di ricerca, si è passati a identificare le stringhe da utilizzare. Per poterle individuare è stato seguito il modello PICO (*Population, Intervention, Comparison and Outcomes*) sviluppato da **Kitchenham** e *Charters* che ha portato ai seguenti risultati:

- *Population*: Ingegneri del software;
- *Intervention*: Continuous integration;
- *Comparison*: Strategie per identificare i flaky test e comparare la costruzione di dataset per progetti open source e closed source;
- *Outcomes*: comprendere lo stato dell'arte e le cose che sono state fatte.

Tramite questo modello, sono state individuate le seguenti keywords: *"Software testing", "Continuous Integration", "Flaky test", "dataset", "open source", "closed source"*.

Successivamente sono stati individuati tre tra i più importanti databases per la raccolta di dati in letteratura, ovvero **IEEE**, **ACM** e **Scopus** e sono state formulate le seguenti query.

Database	**Search**
IEEE	((("flaky" OR "flakiness" OR "test pass and fail" OR "non deterministic") AND "test" AND ("SE" OR "software engineering" OR "computer science")) AND ((("dataset" OR "database")OR ("project open source" OR "close* source" OR "source code"))OR ("build* system*" OR "maven" OR "gradle" OR "continuous integration" OR "integration")OR ("root cause" AND ("detect*" OR "improve*"))))
ACM	((("flaky" OR "flakiness" OR "test pass and fail" OR "non deterministic test*") AND "test" AND ("SE" OR "software engineering" OR "computer science")) AND ((("dataset" OR "database" OR "db")OR ("project open source" OR "close* source" OR "source code")) OR ("build* system*" OR "maven" OR "gradle" OR "building" OR "continuous integration" OR "travis" OR "integration test") OR ("root cause" AND ("detect*" OR "improve*"))))
Scopus	((("flaky" OR "flakiness" OR "test pass and fail" OR "non deterministic test*") AND "test" AND ("SE" OR "software engineering" OR "computer science"))AND ((("dataset" OR "database" OR "db")OR ("project open source" OR "close* source" OR "source code"))OR ("build* system*" OR "maven" OR "gradle" OR "building" OR "continuous integration" OR "travis" OR "integration test")OR ("root cause" AND ("detect*" OR "improve*"))))

E sono stati ottenuti seguenti risultati:

Database	**#Results**
IEEE	13
ACM	153
Scopus	137
Total	303

2.2.2. **Criteri di inclusione/esclusione**

Per ogni artefatto sono state riportate su un foglio di calcolo le seguenti informazioni:

Paper name	Item type	Database	Authors	Publication year	Publication title	Notes

Sono state successivamente eseguite le seguenti fasi:

- Eliminare gli articoli ripetuti (45 articoli);
- Eliminare gli articoli precedenti agli anni 2000 (6 articoli).

Con tale tecnica è stato possibile ridurre gli articoli da analizzare da 303 ad un totale di 252. Successivamente sono stati letti sia il titolo che l'abstract di ogni artefatto e si è deciso di scartare tutti gli articoli che non presentavano dei riferimenti diretti all'argomento trattato. Tale fase ha permesso di eliminare dalla lista 202 artefatti, ottenendo così un totale di 50 articoli.

Si è passata poi alla lettura totale degli articoli rimanenti. Grazie a questa fase sono stati eliminati venti artefatti poiché non erano pertinenti allo studio o lo erano solo in minima parte. Successivamente sono stati riletti tutti gli articoli scartati dalle fasi precedenti e si è ritenuto di riprendere in considerazione uno degli artefatti scartati. Tale tecnica ha permesso di avere nella lista trentuno articoli in totale.

2.2.3. **Snowball**

Dopo aver fatto una prima seleziona degli artefatti di interesse per il mapping sistematica, è stata applicata la tecnica dello "*snowballing*"; tale tecnica consiste nel leggere le "*reference*" di ogni articolo finora considerato, con lo scopo di individuare qualche artefatto che possa essere ritenuto d'interesse per la ricerca, ma che non è ancora stato preso in considerazione. La tecnica è stata applicata sui trentuno articoli e sono stati così individuati altri tre artefatti. Oltre ai tre articoli, sono stati tenuti in considerazione anche nove siti web citati nelle *reference* che hanno permesso di avere un quadro completo della problematica. La tecnica dello snowballing ha permesso quindi di incrementare la lista totale degli articoli portandola ad un totale di trentaquattro.

2.2.4. **Valutazione della qualità**

Nel processo di valutazione della qualità, sono stati identificati dei criteri per valutare la qualità degli articoli presi in considerazione. Lo scopo di questa fase è quella di eliminare dalla lista tutti gli articoli che non soddisfano i criteri minimi di qualità richiesti.

Sono state così definite le seguenti domande:

(**R0**): Gli obiettivi sono chiaramente indicati?

(**R1**): Quanto sono credibili i risultati?

(**R2**): I partecipanti allo studio o le unità di osservazione sono state adeguatamente descritte?

(**R3**): Se lo studio prevede la valutazione di una tecnologia, essa è stata chiaramente identificata?

(**R4**): Tutte le domande dello studio hanno una risposta?

(**R5**): Sono presentati tutti i risultati?

(**R6):** I metodi di raccolta dei dati sono descritti in modo adeguato?

Per ognuna delle seguenti domande è stato assegnato un intervallo compreso tra 0 ed 1 per l'eventuale risposta. La seguente tabella mostra gli intervalli individuati per la valutazione di ogni domanda.

Legenda	
0.0	No
0.1 - 0.3	Raramente
0.4 - 0.6	Parzialmente
0.7 - 0.9	Abbastanza
1.0	Si

Il punteggio totale è stato ottenuto tramite la somma del punteggio assegnato a ogni domanda. Si è deciso di prendere in considerazione solo gli articoli che hanno raggiunto uno standard di qualità almeno pari a "parziale".

La seguente tabella mostra gli intervalli che sono stati individuati per valutare la qualità totale di ogni artefatto.

Legenda	
0.0	No
0.1 - 2.1	Raramente
2.2 - 4.5	Parzialmente
4.6 - 6.2	Abbastanza
>6.3	Si

Di seguito sono mostrati i risultati che si sono ottenuti da tutti gli artefatti analizzati, mentre nell'Appendice A sono riportati i riferimenti degli articoli studiati con il relativo codice.

Study No.	
S1	Si
S2	Abbastanza
S3	Abbastanza
S4	Abbastanza
S5	Abbastanza
S6	Abbastanza
S7	Raramente
S8	Raramente
S9	Abbastanza
S10	Abbastanza
S11	Si
S12	Abbastanza
S13	Si
S14	Si
S15	Si
S16	Si
S17	Abbastanza
S18	Si
S19	Si
S20	Abbastanza
S21	Si
S22	Si
S23	Abbastanza
S24	Si
S25	Si
S26	Raramente
S27	Si
S28	Abbastanza
S29	Abbastanza
S30	Abbastanza
S31	Si
S32	Abbastanza
S33	Abbastanza
S34	Raramente

Dalla valutazione della qualità è emerso che quattro articoli non soddisfano la soglia minima di accettazione. Si è quindi deciso di eliminare dall'elenco gli artefatti numero S7, S8, S26 e S34 e di continuare le successive fasi sui restanti trenta artefatti e i nove siti web.

2.2.5. **Estrazione dei dati**

Per la fase di estrazione dei dati è stata effettuata seguendo il modello proposto da **Petersen et al**.

In questa fase è stata generata una tabella contenente tutti gli studi che sono stati presi in considerazione con le rispettive domande di ricerca a cui rispondevano.

La tabella sottostante sintetizza i risultati che sono stati ottenuti analizzando ogni articolo.

Study	**Anno**	**RQ1**	**RQ2**	**RQ3**	**RQ4**	**RQ5**	**RQ6**	**RQ7**	**RQ8**
S1	2014	-	X	-	X	X	X	-	-
S2	2015	-	X	X	X	X	X	-	X
S3	2015	-	X	-	-	-	-	-	-
S4	2015	-	X	-	-	-	-	-	-
S5	2015	-	X	X	X	-	-	-	-
S6	2015	-	X	-	-	-	-	-	-
S7	2016	-	X	-	-	-	-	-	X
S8	2017	-	X	-	X	-	-	-	-
S9	2017	-	X	X	X	-	-	-	-
S10	2017	-	-	-	-	X	X	-	-
S11	2017	-	X	-	X	-	-	-	-
S12	2018	X	X	X	X	-	X	-	X
S13	2018	-	X	X	X	X	X	X	-
S14	2018	-	X	-	X	-	-	-	-
S15	2018	-	-	-	-	-	X	-	-
S16	2019	-	-	-	-	X	X	-	X
S17	2019	-	X	-	X	X	X	-	X
S18	2019	-	-	-	-	-	-	-	-
S19	2019	-	X	-	-	-	-	-	-

Study	**Anno**	**RQ1**	**RQ2**	**RQ3**	**RQ4**	**RQ5**	**RQ6**	**RQ7**	**RQ8**
S20	2019	X	X	X	X		X	-	-
S21	2019	-	X	-	-	-	-	-	-
S22	2019	X	X	X	X	-	X	-	-
S23	2019	-	X	-	-	-	-	-	-
S24	2019	-	X	X	-	-	-	-	-
S25	2019	-	-	-	-	-	-	-	-
S26	2019	-	X	-	-	X	X	-	-
S27	2019	-	X	-	-	-	-	-	-
S28	2019	X	X	X	X	X	X	-	-
S29	2019	-	X	-	-	X	X	-	-
S30	2019	-	-	-	-	X	X	-	-

Il grafico in figura 2.1 sintetizza tutte le fasi che sono state effettuate per la selezione degli artefatti utili per il mapping sistematico.

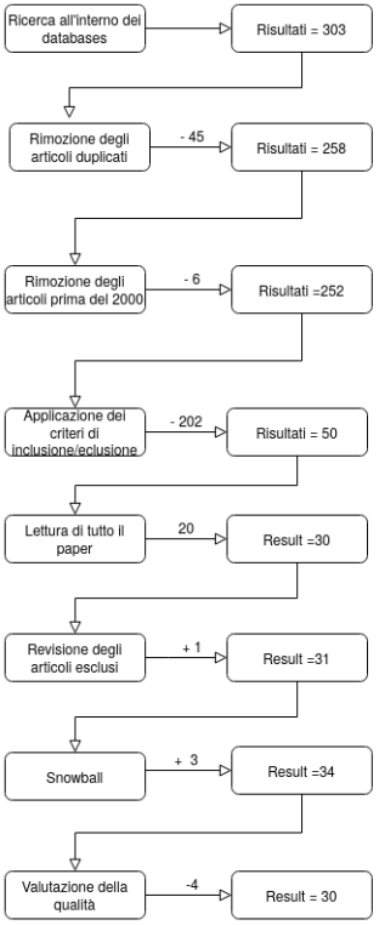


Figura 2.1: *Sintesi della selezione degli articoli*

2.2.6. **Minacce alla validità**

Minaccia alla validità descrittiva

La validità descrittiva è la misura con cui le osservazioni sono descritte in maniera accurata ed oggettiva. Questo tipo di minaccia è solitamente maggiore negli studi di tipo qualitativo rispetto a quelli di tipo quantitativo. Questa minaccia quindi viene considerata sotto controllo.

Minaccia alla validità teorica

La validità teorica è data dalla capacità dell'autore che ha effettuato il mapping study di "cogliere" quello che gli autori degli articoli volevano effettivamente dimostrare. Il fattore che gioca un ruolo fondamentale all'interno di questa minaccia è il pregiudizio che si può avere su determinati autori che a loro volta influenzano gli articoli da analizzare.

Un'altra minaccia alla validità teorica può derivare dalla formulazione delle query che potrebbero aver portato a non includere alcuni risultati importanti ai fini dello studio.

Per mitigare questa minaccia si è deciso di applicare la tecnica dello *snowballing* su tutti gli articoli che sono stati selezionati durante la fase di analisi. La tecnica ha infatti il fine di individuare artefatti utili che possono essere stati non considerati durante la fase di ricerca.

Infine, la fase di estrazione dei dati effettuata da un singolo ricercatore potrebbe aver portato a non considerare studi importanti. Per mitigare questa minaccia è stato chiesto ad esperto del dominio applicativo ma esterno alla ricerca di valutare eventuali studi ritenuti dal primo autore "poco chiari".

Generalizzazione

Petersen et al. effettuano una distinzione tra la generalizzazione esterna ed interna. Con la prima si intende la generalizzazione tra gruppi o organizzazioni, mentre con la seconda si indica la generalizzazione all'interno di un singolo gruppo. I risultati ottenuti potrebbero non essere applicabili all'interno di una revisione sistematica della letteratura poiché gli obiettivi delle ricerche sono differenti.

Tuttavia, visto che si è seguito un approccio di tipo sistematico, gran parte delle strategie di ricerca risulta in comune, pertanto questa minaccia si ritiene mitigata.

Validità di interpretazione

La validità di interpretazione viene raggiunta quando le conclusioni che sono state tratte dai dati risultano essere ragionevoli. Una delle minacce di interpretazione dei dati può essere dovuta al pregiudizio che l'autore del mapping ha verso alcune conclusioni che sono state mostrate nei differenti artefatti. Per mitigare tale minaccia è stato chiesto ad un esperto del dominio applicativo ma esterno alla ricerca di dare un parere sulla veridicità dei dati presi in esame.

Ripetibilità

È stato effettuato un mapping study sistematico proprio al fine di garantire la ripetibilità di tutto ciò che si è fatto. Sono state inoltre seguite le linee guida dettate da **Kai Petersen, Sairam Vakkalanka et al**. all'interno dell'articolo "*Guidelines for conducting systematic mapping studies in software engineering: An update*".

Sono inoltre dettagliate tutte le fasi seguite con assoluto rigore, pertanto tale minaccia risulta mitigata.

2.2.7. Analisi dei dati e classificazione

L'obiettivo dell'analisi dei dati e della classificazione è anche quello di organizzare a livello visivo le informazioni estratte dalle varie domande di ricerca, in modo che esse possano essere rapidamente individuate e comprese. Le informazioni su ogni articolo sono state mappate ed illustrate visivamente attraverso l'utilizzo di grafici e diagrammi. Successivamente è stato assegnato una categoria ad ogni artefatto e si è passati quindi alla fase di conteggio.

Numero di dataset (RQ1)

Dalla prima domanda di ricerca è emerso che in quasi tutti gli studi non è stato rilasciato un dataset.

Gli unici dataset che sono stati rilasciati (in totale quattro) rientrano nel periodo 2018-2019; questo dato è indice di un argomento di ricerca poco conosciuto e che necessita di ulteriori sviluppi.

Il grafico in figura 2.2 mostra la frequenza di pubblicazione dei dataset messi a disposizione.

Figura 2.2: *Frequenza di pubblicazione del dataset*

Dal grafico è emerso che quasi la totalità dei dataset messi a disposizione sono stati implementati nel corso del 2019 (75%). Il numero dei dataset risulta essere comunque molto esiguo.

Lavoro su progetti closed source (RQ2)

Dalla seconda domanda di ricerca è emerso che la quasi totalità degli studi effettuati non ha considerato progetti closed-source nella loro

valutazione e in alcuni articoli questa problematica è stata considerata solo come minaccia per la validità.

Il grafico in figura 2.3 mostra il numero di studi sui quali sono state effettuate ricerche anche su progetti closed-source suddivisi per anno.

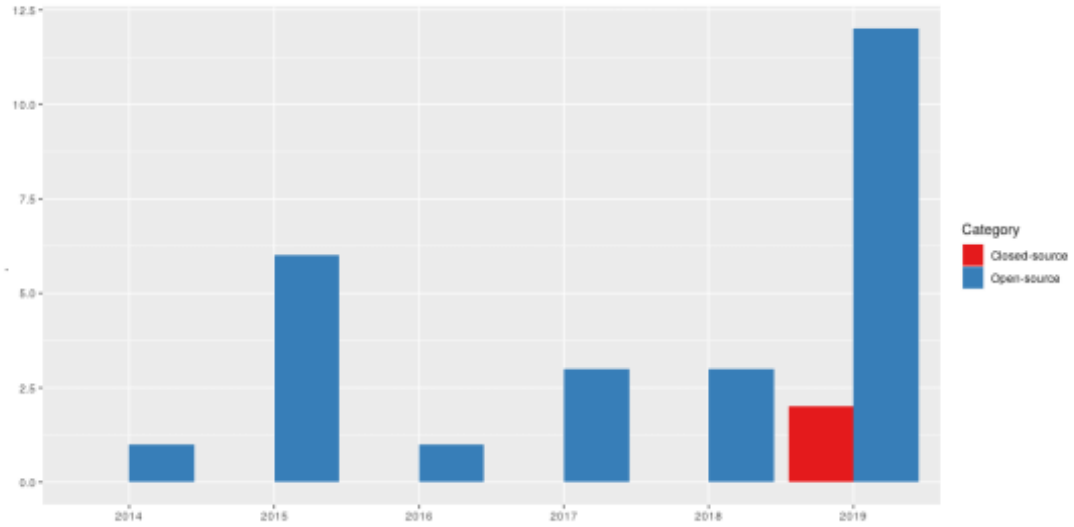


Figura 2.3: *Numero di progetti su cui sono stati condotti gli studi*

Linguaggi di programmazione utilizzati (RQ3)

Dall'analisi effettuata risulta che quasi tutti gli articoli analizzati riportano le caratteristiche del linguaggio di programmazione impiegato per definire gli oggetti sperimentali (ventiquattro su trenta). Di questi, quasi la totalità dei progetti analizzati è scritta in Java (viene infatti definita come una minaccia in alcuni degli articoli analizzati, in quanto il linguaggio potrebbe presentare alcune caratteristiche non comuni a tutti i linguaggi di programmazione). È stata quindi approfondita la problematica per capire quale fosse la distribuzione dei linguaggi di programmazione.

Il grafico in figura **??** mostra la frequenza di ogni linguaggio di programmazione.

Figura 2.4: *Distribuzione dei linguaggi di programmazione analizzati*

È possibile notare che più della metà dei progetti analizzati sono stati sviluppati in Java. Tale dato non stupisce in quanto attualmente il linguaggio Java risulta essere uno dei più utilizzati al mondo. È altresì importante sottolineare che per alcuni linguaggi la problematica è stata trattata solo in minima parte o non è stata per nulla trattata.

Tipi di build utilizzati (RQ4)

Dall'analisi effettuata è emerso che non sempre è stato dichiarato esplicitamente il sistema di build adoperato durante la costruzione del dataset.

Negli articoli in cui tale informazione è presente, il linguaggio di programmazione utilizzato è sempre stato Java. Dei trenta articoli, in nove è stato riportato anche il tipo di build system utilizzato (in alcuni casi anche più di uno).

Il grafico in figura 2.5 mostra la frequenza dei sistemi di building utilizzati all'interno dei progetti che sono stati testati.

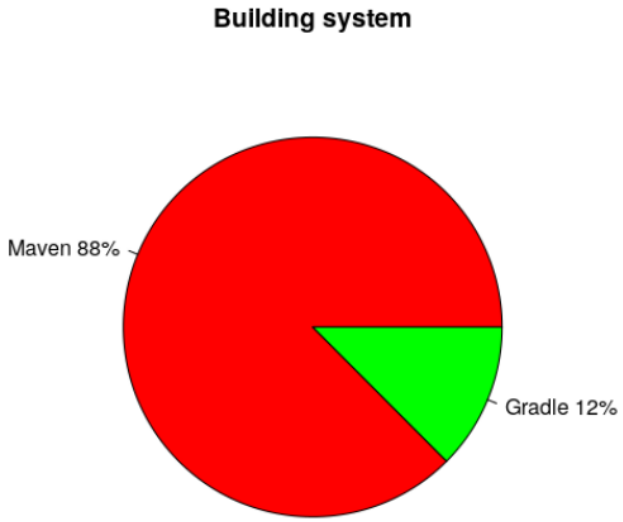


Figura 2.5: *Sistemi di building utilizzati*

È possibile notare che quasi la totalità dei progetti analizzati utilizzano come sistema di build Maven (preso in considerazione in sette articoli), mentre solo il 12% utilizzano Gradle (preso in considerazione in un solo articolo).

Individuazione del dataset (RQ5)

In tutti gli articoli in cui è stata dichiarata la costruzione del dataset (diciotto articoli), è stato anche descritto il metodo utilizzato per la descrizione.

La strategia più comune risulta quella di scaricare da GitHub i repository più popolari per poi verificare la presenza di metodi flaky (38%).

Le altre strategie utilizzate si basano principalmente sull'uso di Travis (valutando lo stato della build oppure i file di log generati) e sul cercare la stringa "flaky" all'interno dei bug report o nei messaggi di commit all'interno del motore di ricerca di GitHub.

Il grafico in figura 2.6 mostra la percentuale di utilizzo di ogni strategia per individuare i dataset.

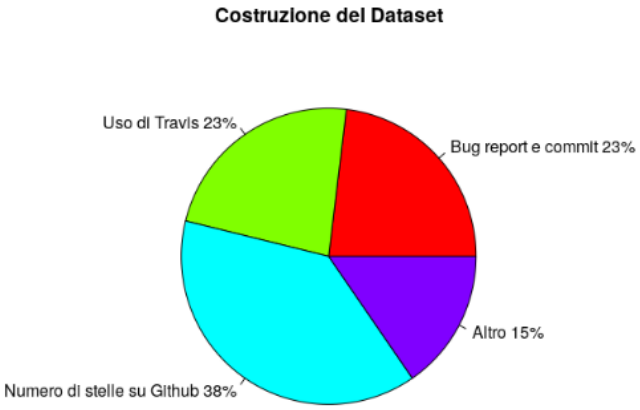


Figura 2.6: *Strategie usate per la costruzione del dataset*

Quali sono le root cause più frequenti (RQ6)

Sui trenta articoli esaminati solo in dieci casi sono state individuate le *root cause* dei *flaky test*.

In gran parte degli articoli le root cause risultano le stesse, ossia: dipendenza dall'ordine, concorrenza, async wait, tempo, rete, random, perdita di risorse, operazioni con numeri float e problemi di I/O.

È però interessante soffermarsi sulle root cause "Output restrittivi" e "Interfaccia grafica", tali root cause infatti sono presenti con una percentuale molto bassa (2%) poiché non sono state mai individuate precedentemente, ma sono frutto di studi effettuati nel 2019.

Possiamo quindi definirle nel seguente modo:

- *Output restrittivi*: Sono degli output validi, ma considerati fuori dai "range" consentiti per questioni di design;
- *Interfaccia grafica*: Un'interfaccia grafica viene definita come "flaky" nel momento in cui avviene una mancata comunicazione tra il processo che esegue il rendering e l'interfaccia stessa.

La figura 2.7 mostra le root cause più comuni.

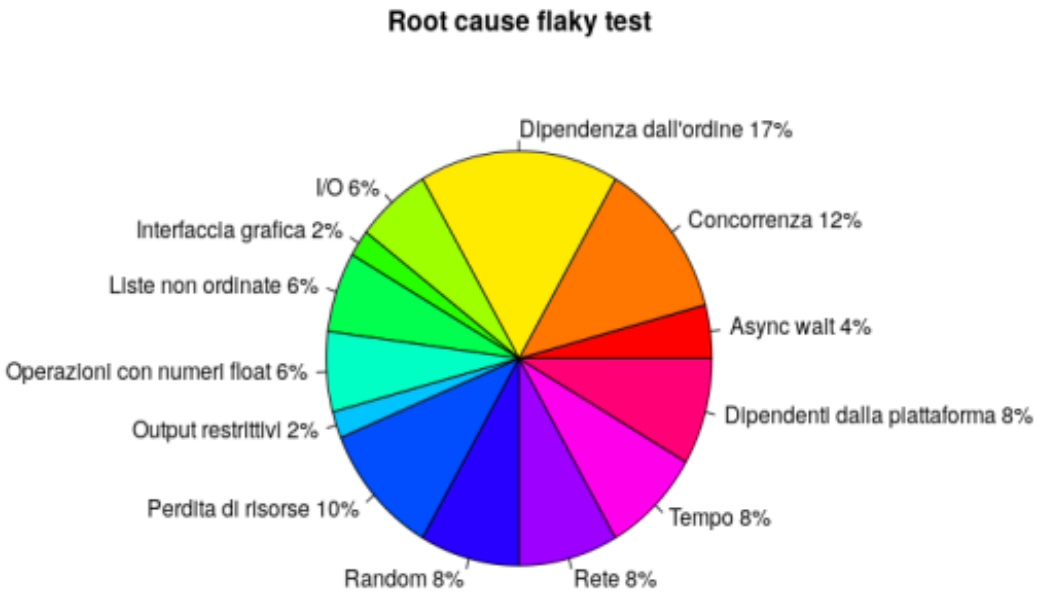


Figura 2.7: *Root cause dei flaky test*

Metodo empirico per la valutazione delle root cause (RQ7)

Dagli articoli che rispondono a questa domanda di ricerca sono state estratte le tecniche adoperate per valutare le root cause dei flaky test.

Nel 29% dei casi, il metodo comunemente utilizzato al momento consiste nell'effettuare nuovamente l'esecuzione di un caso di test e verificare se quest'ultimo cambia stato rispetto all'esecuzione precedente (pass/fail o viceversa). Le altre strategie adoperate sono:

- Strumentazione del codice: Creazione di un codice che monitora comportamenti specifici di una applicazione;
- Coverage: Con questa tecnica viene calcolata la coverage (solitamente la line) ogni volta che il caso di test viene eseguito, se si

riscontra un cambio di coverage in parti del codice che non sono state modificate, il test viene classificato come flaky;

- Analisi del codice: L'analisi del codice può essere eseguita in due modi:

  Statica: In questo caso gli sviluppatori definiscono un metodo flaky se rilevano che nel codice sorgente del test viene fatto uso di qualche funzione che può generare un flaky;

  Dinamica: Viene eseguito il codice sorgente del test e si valutano glioutput generati e i file di log scritti. Se si nota una discordanza tra le due esecuzioni, il metodo viene definito flaky.

Il grafico in figura 2.8 mostra le frequenze di utilizzo di ogni tecnica che è stata descritta precedentemente.

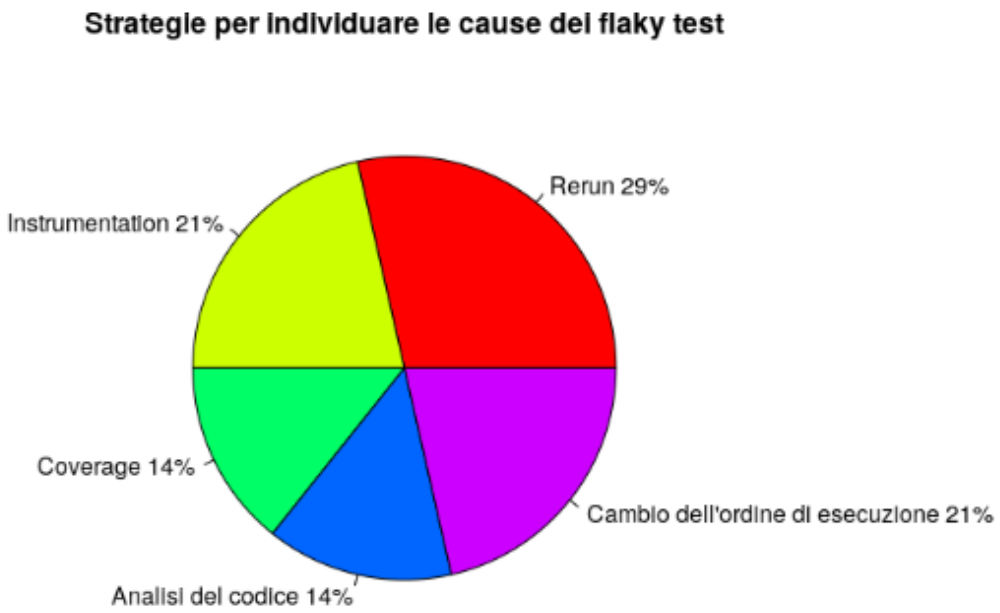


Figura 2.8: *Strategia per individuare i flaky test*

A chi viene indirizzata la ricerca mostrata nei diversi studi (RQ8)

Dall'analisi effettuata è emerso che in cinque artefatti è chiaramente indicato a chi fosse rivolto lo studio (grandi aziende e ambito accademico); invece, nei restanti artefatti si può intuire che lo studio è rivolto ad un pubblico prettamente accademico.

La figura 2.9 mostra la percentuale degli articoli dove vengono menzionati sia le aziende che gli accademici come fruitori finali dello studio, comparato a quelli dove vengono citati solo gli accademici.

Figura 2.9: *Ambito di ricerca per ogni artefatto*

È interessante soffermarsi su quest'ultima percentuale poiché nonostante il problema sia stato posto inizialmente dalle grandi aziende (Google, Facebook, Huawei in primis), esso è diventato un argomento su cui sono nati molteplici studi da parte degli accademici.

Anno di pubblicazione (RQ9)

Gran parte degli studi sono stati pubblicati tra il 2018 e il 2019 (il 63%), questo ci suggerisce che l'argomento rappresenta una problematica attuale ed è tuttora molto studiato. La figura 2.10 mostra la frequenza di pubblicazione degli artefatti suddivisa per anni.

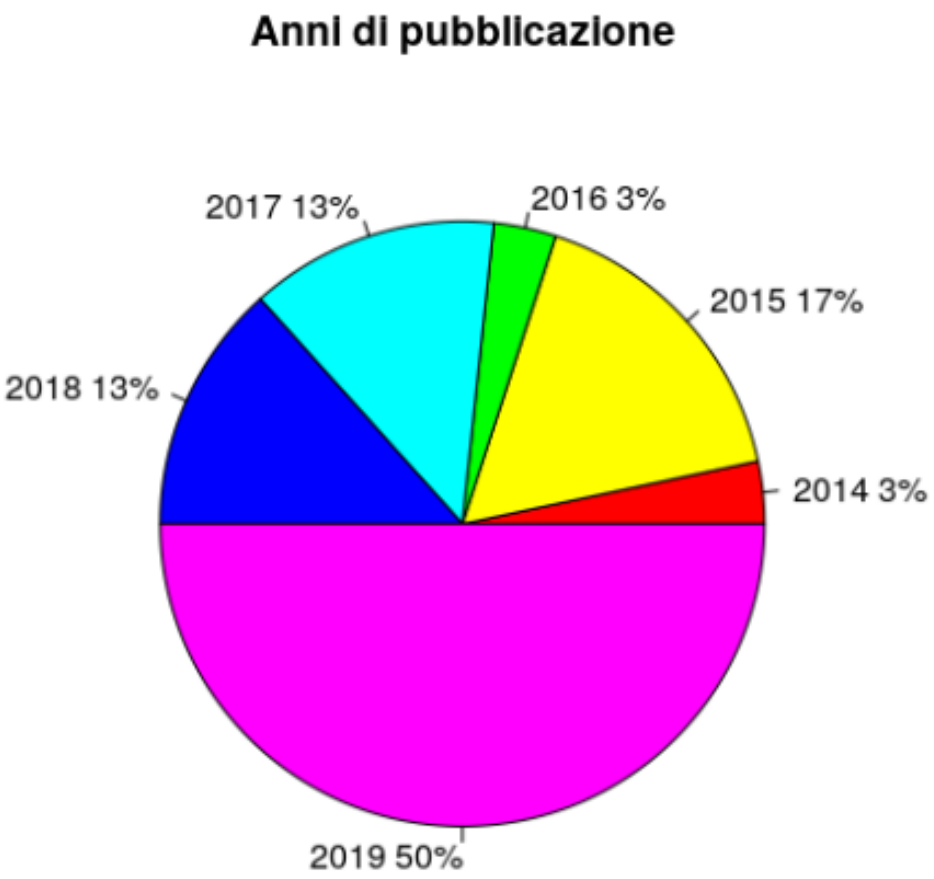


Figura 2.10: *Frequenza di pubblicazione suddivisa per anni*

2.2.8. **Conclusioni**

Dallo studio effettuato è emerso che:

- Gli unici studi sui quali sono stati effettivamente rilasciati dei dataset sono solo quelli presentati tra il 2018 e il 2019, tale dato ci porta a sottolineare l'importanza e la necessità da parte sia delle aziende che degli accademici di rilasciare nuovi dati o di ampliare quelli già esistenti;

- Bisogna analizzare con particolare attenzione anche progetti closed-source poiché le dinamiche e i meccanismi con cui vengono costruiti tali progetti possono essere completamente differenti da quelle open-source;

- Occorre prestare maggiore attenzione anche ad altri tipi di linguaggi di programmazione, poiché gran parte degli studi sono stati effettuati soltanto su progetti scritti in Java e che utilizzano Maven come build system;

- L'uso di GitHub per individuare progetti open-source affetti da flakiness è effettivamente un buon punto di partenza ma bisognerebbe analizzare anche repository private o progetti closed-source;

- Negli anni le root cause individuate sono state ampliate per comprendere nuove piattaforme mai considerate in precedenza (es. Android); tale ampliamento ha permesso di considerare all'interno delle root cause anche problemi legati all'hardware (es. fotocamera non disponibile). Attualmente questi problemi sono ancora classificati all'interno della causa "problemi dipendenti dalla piattaforma", ma non si esclude che possano diventare una nuova categoria in futuro.

3. Capitolo 3

3.1. Creazione del dataset

Dall'analisi effettuata nel capitolo precedente sono emerse diverse criticità nello studio dei *flaky test*. Tali criticità sono sinonimo di un argomento di ricerca ancora molto da approfondire. Le problematiche maggiori che sono state sono evidenziate all'interno della ricerca sono principalmente legate alla carenza di dati messi a disposizione per la comunità. Infatti, in molti degli studi precedentemente analizzati, la carenza di dati viene considerata una minaccia alla validità poiché i progetti considerati sono quasi tutti scritti in linguaggio **Java** ed usano **Maven** come sistema di building. Avere diverse tipologie di dataset risulta estremamente importante in quanto tale problematica risulta ancora di notevole interesse da parte delle grandi multinazionali che ogni giorno sviluppano centinaia di progetti utilizzando differenti linguaggi di programmazione.

Nel 2019, nell'ambito della conferenza "*Testing and Verification Symposium*" organizzata da Facebook, sono stati proposte diverse idee innovative che potessero in qualche modo far fronte alla problematica flakiness. Tra le idee maggiormente apprezzate è possibile citare: "*A scalable infrastructure for fuzzy-driven root causing of flaky tests*" ideato da Filomena Ferrucci (Università di Salerno), Pasquale Salza (Università di Zurigo) e Valerio Terragni (Università della Svizzera italiana).

La loro idea prevede un nuovo approccio per identificare le *root*

cause di un *flaky test*, che non prevede nessun tipo di "*instrumentation*" del codice, ma va ad identificare la *root cause* dei flaky test, eseguendo il caso di test in differenti configurazioni definite da cluster. Ogni cluster esplora attivamente uno specifico spazio di esecuzione **non deterministico**. Il cluster che mostrerà i risultati più bilanciati tra "pass" e "fail" sarà utilizzato per identificare la root cause di quello specifico *flaky test*. Il primo passo da fare per sviluppare questo nuovo tool è creare un nuovo dataset con quanti più **flaky** test possibili.

3.2. Approccio alla problematica

L'idea di base è stata quella di partire da un dataset già esistente, ossia quello rilasciato dagli sviluppatori di **iDFlakies**, ed ampliarlo con nuove informazioni. Questo framework, rilasciato nel 2019 permette l'individuazione di un *flaky test* con una classificazione parziale (dipendente dall'ordine e indipendente dall'ordine).

Il dataset rilasciato con questo framework conta al suo interno quattrocentoventidue *flaky test*, di cui il 50.5% sono dipendenti dall'ordine, mentre i restanti 49.5% sono non dipendenti dall'ordine. L'individuazione dei *flaky test* all'interno del tool di **iDFlakies** avviene tramite una serie di configurazioni che cambiano l'ordine di esecuzione sia delle classi di test che dei test case al loro interno. Tra le configurazioni presenti all'interno di **iDFlakies**, possiamo citare:

- *Original-order*: Esegue tutti i casi di test nell'ordine in cui sono stati scritti dallo sviluppatore;
- *Random-Class*: Esegue le classi della suite di test in ordine pseudo-casuale, ma mantenendo l'ordine originale di esecuzione dei test all'interno di ogni classe;
- *Random-class-method*: Esegue gerarchicamente prima in ordine pseudo-casuale le classi che compongono la test suite per

poi eseguire in maniera pseudo-casuale i singoli casi di test che compongono le varie classi;

- *Reverse-class*: Inverte l'ordine di esecuzione delle classi di test ma mantiene invariato l'ordine di esecuzione dei singoli test all'interno della classe;
- *Reverse-class-method*: Inverte l'ordine sia delle classi di test che dei metodi di test al suo interno.

Analizzando il dataset messo a disposizione è emerso che ogni progetto contiene le informazioni presentate nella tabella sottostante: La

URL	**SHA**	**Test count**	**Module Name**	**Test Name**	**Category**	**Version**

figura 3.1 mostra un esempio per descrivere il contenuto del dataset rilasciato.

URL	SHA	Test Count	Module Name	Test Name	Categ	Version
https://github.com/alien4cloud/alien4c	eb57d0fe	14	alien4cloud-alien4cloud-security	alien4cloud.security.LdapAuthenticationProviderTest.testLdapUserImport	OD	Extended
https://github.com/ctco/cukes	b483e1a8	14	cukes-cukes-http	lv.ctco.cukes.http.facade.HttpAssertionFacadeImplTest.shouldReturnBodyWhenEnabledAndN	OD	Extended
https://github.com/espertechinc/esper	590fa9c9	49	esper-runtime	com.espertech.esper.runtime.internal.schedulesvcimpl.TestSchedulingServiceImpl.testTrigge	OD	Extended
https://github.com/fhoeben/hsac-fitnes	a64c18d9	251	hsac-fitnesse-fixtures	nl.hsac.fitnesse.fixture.slim.FileFixtureTest.testDelete	OD	Extended
https://github.com/gooddata/GoodDat	58728ec3	8	GoodData-CL-connector	com.gooddata.csv.DataTypeGuessTest.testIsDate	OD	Extended
https://github.com/hexagonframework	dd11b976	48	spring-data-ebean	org.springframework.data.ebean.repository.UserRepositoryIntegrationTest.deleteById	OD	Extended
https://github.com/jhipster/jhipster-reg	00db3661	53	jhipster-registry	io.github.jhipster.registry.web.rest.AccountResourceTest.testGetUnknownAccount	OD	Extended
https://github.com/ktuukkan/marine-a	af000384	925	marine-api	net.sf.marineapi.ais.event.AbstractAISMessageListenerTest.testSequenceListenerWithIncom	OD	Extended
https://github.com/ktuukkan/marine-a	af000384	925	marine-api	net.sf.marineapi.ais.parser.AISMessageFactoryTest.testCreateWithTwo	OD	Extended
https://github.com/openpojo/openpojo	9badbcc4	1185	openpojo	com.openpojo.issues.issue112.IssueTest.shouldNotFail	OD	Extended
https://github.com/openpojo/openpojo	9badbcc4	1185	openpojo	com.openpojo.validation.test.impl.GetterTesterTest.shouldSuccessfullyValidateEvenIfLogging	OD	Extended
https://github.com/openpojo/openpojo	9badbcc4	1185	openpojo	com.openpojo.validation.test.impl.SerializableTesterTest.shouldSkipOverNonSerializableClas	OD	Extended
https://github.com/openpojo/openpojo	9badbcc4	1185	openpojo	com.openpojo.validation.test.impl.SetterTesterTest.shouldSuccessfullyValidateEvenIfLogging	OD	Extended
https://github.com/openpojo/openpojo	9badbcc4	1185	openpojo	com.openpojo.validation.utils.ValidationHelperTest.shouldReportMissingASMProperty	OD	Extended
https://github.com/pholser/junit-quickc	9361b6da	11	junit-quickcheck-guava	com.pholser.junit.quickcheck.guava.generator.FunctionPropertyParameterTypesTest.arrayOf	OD	Extended
https://github.com/sonatype-nexus-co	60a9e8de	18	nexus-repository-helm	org.sonatype.repository.helm.internal.metadata.IndexYamlBuilderTest.testChartIndexPassed	OD	Extended
https://github.com/spring-projects/spri	5637994b	10	spring-data-envers	org.springframework.data.envers.repository.support.QueryDslRepositoryIntegrationTests.test	OD	Extended
https://github.com/spring-projects/spri	e8d89c9e	119	spring-ws-spring-ws-security	org.springframework.ws.soap.security.xwss.callback.SpringDigestPasswordValidationCallbac	OD	Extended
https://github.com/spring-projects/spri	e8d89c9e	119	spring-ws-spring-ws-security	org.springframework.ws.soap.security.xwss.callback.SpringDigestPasswordValidationCallbac	OD	Extended
https://github.com/tbsalling/aismessa	7b0c4c70	44	aismessages	dk.tbsalling.aismessages.nmea.NMEAMessageHandlerTest.canHandleFragmentedMessage	OD	Extended
https://github.com/tbsalling/aismessa	7b0c4c70	44	aismessages	dk.tbsalling.aismessages.nmea.NMEAMessageHandlerTest.canHandleUnfragmentedMessag	OD	Extended
https://github.com/tools4j/unix4j	367da7d2	288	unix4j-unix4j-core-unix4j-command	org.unix4j.unix.FindFileTimeDependentTest.find_fileCreatedBeforeNow	OD	Extended
https://github.com/wikidata/wikidata-to	20de6f7f1	49	Wikidata-Toolkit-wdtk-dumpfiles	org.wikidata.wdtk.dumpfiles.MwDumpFileProcessingTest.testMwDailyDumpFileProcessing	OD	Extended
https://github.com/wikidata/wikidata-to	20de6f7f1	49	Wikidata-Toolkit-wdtk-dumpfiles	org.wikidata.wdtk.dumpfiles.MwDumpFileProcessingTest.testMwMostRecentFullDumpFilePro	OD	Extended
https://github.com/wikidata/wikidata-to	20de6f7f1	49	Wikidata-Toolkit-wdtk-dumpfiles	org.wikidata.wdtk.dumpfiles.MwDumpFileProcessingTest.testMwRecentCurrentDumpFilePro	OD	Extended
https://github.com/wikidata/wikidata-to	20de6f7f1	23	Wikidata-Toolkit-wdtk-util	org.wikidata.wdtk.util.DirectoryManagerFactoryTest.createDefaultDirectoryManagerPath	OD	Extended
https://github.com/wikidata/wikidata-to	20de6f7f1	23	Wikidata-Toolkit-wdtk-util	org.wikidata.wdtk.util.DirectoryManagerFactoryTest.createDirectoryManagerIoException	OD	Extended
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeClaimReleaseTest.aCreateStandaloneTaskForGrou	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeClaimReleaseTest.bClaimAndRelease	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeClaimReleaseTest.eCreateStandaloneTaskAndClai	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeClaimReleaseTest.fCreateStandaloneTaskAndClaim	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeCompleteTaskTest.aCreateStandaloneTaskAndCom	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeCompleteTaskTest.bCreateStandaloneTask	OD	Comprehensive
https://github.com/Activiti/Activiti	b11f757a	42	Activiti-activiti-spring-boot-starter	org.activiti.spring.boot.tasks.TaskRuntimeCompleteTaskTest.ctryCompletingWithUnauthorize	OD	Comprehensive

Figura 3.1: *Parte del dataset di IDFlakies*

Questo framework recupera il link e lo sha di ogni repository e crea un'immagine Docker[1]. Una volta scaricato il progetto, prova ad eseguire una delle configurazioni sopra descritte su una delle immagini Docker appena create. È possibile inoltre settare *iDFlakies* per per-

[1]Docker è un progetto open-source che automatizza il deployment di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.

mettergli di eseguire tutte le configurazioni in maniera consequenziale, oppure scegliere solo alcune di queste configurazioni.

Nel momento in cui viene riscontrato un "fail" in un particolare test case in una di queste cinque configurazioni, il tool riesegue la classe di test con lo stesso ordine. Se nella seconda esecuzione il metodo mostra ancora un "fail" allora viene classificato come OD (**O**rder **D**ependent); in caso contrario, il caso di test viene classificato come NO (**N**o **O**rder Dependent).

Nel dataset considerato è presente anche la colonna "Version", tale voce può assumere due possibili valori: "Comprehensive" ed "Extended". Con l'etichetta "Comprehensive" viene identificato un test su cui sono state utilizzate tutte e cinque le configurazioni sviluppate al fine di individuare il *flaky test*, mentre con l'etichetta "Extended" si indica che su quello specifico caso di test è stata utilizzata unicamente la configurazione "*random-class-method*" (descritta precedentemente).

Tale scelta è stata fatta poiché in alcuni casi l'esecuzione di tutte e cinque le configurazioni risultava estremamente lenta e costosa.

Per costruire il nuovo dataset si è deciso di estrarre circa duecento righe del dataset di **iDFlakies**. L'estrazione è stata fatta selezionando soltanto le linee etichettate come "NO". Di queste duecento righe sono state successivamente scartate tutte quelle che facevano riferimento al progetto "**Apache Hadoop**" poiché la build di tale progetto impiegava svariate ore per terminare. Si è quindi deciso di lavorare sui restanti centosessanta sei metodi.

La prima operazione effettuata sul dataset originale è stata quella di eliminare le colonne "Test Count", "Category" e "Version", poiché ritenute poco utili ai fini dello studio. La procedura è stata effettuata manualmente con l'aiuto di un plugin di "Visual Studio Code". Alla fine di questa fase è stato quindi ottenuto un csv con i seguenti attributi:

- URL;
- SHA;

- Module Name;
- Test Name.

3.2.1. **Primo approccio**

La prima prova di classificazione delle root cause è stata fatta tramite l'utilizzo dell'IDE di IntelliJ IDEA. L'IDE in questione ha la possibilità di eseguire un caso di test in diverse configurazioni. Tra le configurazioni possibili, si possono citare:

- *Ripeti finché non fallisce*: Tale configurazione riesegue un caso di test (o una suite di test) finché non viene individuato un fallimento;
- *Ripeti N volte*: Tale configurazione ripete un caso di test (o una suite) un numero prestabilito di volte.

Sono stati quindi importati diversi progetti all'interno dell'ambiente di sviluppo e sono stati eseguiti diversi "run" per verificare manualmente la flakiness dei metodi analizzati. Tale strategia ha però evidenziato uno dei limiti di questa ricerca, in quanto in svariati casi sono stati riscontrati problemi nell'effettuare con successo la build del sistema. Le problematiche più comuni in questa fase sono state sia legate a dipendenze non presenti (es. librerie non più supportate o mancanti) sia a problemi di configurazione all'interno del file POM. A causa di questi problemi legati alla configurazione dei progetti si è deciso di scartare questo approccio di classificazione manuale.

3.2.2. **Secondo Approccio**

Dato che l'approccio precedente non aveva fornito una soluzione accettabile si è deciso di slegarsi dagli IDE e di trovare un approccio alternativo. In questo secondo approccio si è deciso di analizzare e automatizzare il processo di esecuzione dei singoli casi di test.

Per effettuare tali automatismi, sono stati creati una serie di script BASH. Di seguito vengono elencati nello specifico tutti i file che sono stati generati per la prima creazione del progetto.

- ConfigFile;
- Input.csv;
- IDFlakies.sh;
- FirstStep.sh;
- SecondStep.sh;
- MainStep.sh.

IDFlakies.sh

Lo script "IDFlakies.sh" prende in input un file csv precedentemente modificato (ovvero senza le colonne di "Test Count", "Category" e "Version") e scarica i repository indicati all'interno del file. Successivamente esegue il comando *git checkout* e si allinea alla commit indicata. Esegue poi il comando *mvn install - DskipTests -fn -B*, e ricostruisce il path del "Module Name" indicato nel progetto cercando il sotto modulo più "vicino" alla classe presa in esame. Infine, le informazioni di interesse per lo studio vengono salvate all'interno di un nuovo file csv denominato "outIDFlakies.csv". Lo script ha eseguito i passaggi sopra indicati per ogni riga presente all'interno del file "Input.csv".

FirstStep.sh

Lo script "FirstStep.sh" legge il contenuto del file "outIDFlakies.csv" e per ogni riga esegue il comando *mvn install -DskipTests -fn -B*. Tale comando viene rieseguito anche in questa fase in quanto all'interno del dataset sono presenti alcuni progetti ripetuti, ma con sha differenti,

si ha quindi bisogno di allinearsi ogni volta al branch corrente; l'output di questo comando viene salvato all'interno di un csv denominato "outFirstStep.csv".

SecondStep.sh

Lo script "SecondStep.sh" può essere considerato il cuore del progetto. Lo script legge il contenuto del file csv "outFirstStep.csv" e successivamente per ogni repository esegue il comando (messo a disposizione da **Maven**) *mvn - Dtest=ClassName#MethodName test*. Tale comando permette l'esecuzione di un singolo caso di test isolandolo dai restanti. Lo script analizza l'output generato e in base al risultato del test aumenta il contatore dei pass o dei fail di una unità. Questa operazione viene eseguita N volte per poi salvare all'interno di un csv denotato "finalCSV.csv" il risultato finale.

MainStep.sh

È stato infine creato uno script per eseguire in modo sequenziale gli step sopra descritti. Tale script fa uso di un file di configurazione chiamato "ConfigFile" dove al suo interno è stato possibile configurare il numero delle ripetizioni ed il nome dei file che dovevano essere generati alla fine di ogni fase.

3.2.3. Ottimizzazione degli script

Una volta terminata la prima versione del progetto, quest'ultimo è stato effettivamente testato all'interno di un ambiente reale. In questa fase sono emerse diverse criticità dovute soprattutto alla lentezza di esecuzione della build per alcune repository presenti; inoltre, si è notato che le informazioni di cui si è effettivamente tenuto traccia non erano sufficienti per permettere un'analisi dettagliata dei *flaky test* ri-

scontrati. Si è quindi deciso di creare una nuova versione per sopperire a tali mancanze.

IDFlakies.sh 2.0

L'unica modifica effettuata sullo script di "IDFlakies" è stata quella di aggiungere il nome del repository in una delle colonne di output del file "outputIDFlakies.csv". Tale informazione infatti viene spesso utilizzata anche nelle fasi successive, ma nella prima versione veniva ricavato ogni volta.

Di seguito è mostrato il codice dello script "IDFlakies.sh".

```
1  #!/bin/bash
2  common() {
3      string1=$pathFile
4      string2=${arrayPom[$i]}
5      first_diff_char=$(cmp <(echo "$string1") <(
6          echo "$string2"
7      ) | cut -d "␣" -f 5 | tr -d ",")
8      ris=${string1:0:$((first_diff_char - 1))}
9      echo "$ris"
10 }
11 getclassName() {
12     local stringPath=$testName
13     IFS='.'                          # hyphen (-)
         is set as delimiter
14     read -ra ADDR <<<"$stringPath" # str is read
          into an
15     array as tokens separated by IFS
16     sizeArray=${#ADDR[@]}
17     className=${ADDR[sizeArray - 2]}
18     echo "$className"
19 }
20 getNameMethod() {
```

```
21      local stringPath=$testName
22      IFS='.'
23      read -ra ADDR <<<"$stringPath"
24      sizeArray=${#ADDR[@]}
25      nameMethod=${ADDR[sizeArray - 1]}
26      echo "$nameMethod"
27  }
28  cloneAndCheckRepo() {
29      local link=$linkRepo
30      local sha=$shaCommit
31      local baseDir=$BASEDIR
32      local repoFolder=$REPOFOLDER
33      cd $baseDir
34      cd $repoFolder
35      git clone $linkRepo
36      basename=$(basename $link)
37      nameRepo=${basename%.*}
38      cd $nameRepo
39      echo "git␣checkout␣" $sha
40      git checkout $sha
41  }
42  BASEDIR=$1
43  REPOFOLDER=$2
44  fileOutput=$BASEDIR"outIDFlakies.csv"
45  maxLength=-1
46  path=$BASEDIR"input.csv"
47  while IFS= read -r line; do
48      IFS=',' read -r -a array <<<"$line"
49      linkRepo=${array[0]}
50      shaCommit=${array[1]}
51      testName=${array[3]}
52      cloneAndCheckRepo "$linkRepo" "$shaCommit" "
```

```
        $BASEDIR"
53      "$REPOFOLDER"
54      className="$(getclassName)"
55      nameMethod="$(getNameMethod)"
56      pathFile=$(find -name "$className".java)
57      readarray -d '' arrayPom < <(find . -name
          pom.xml -print0)
58      lengthArray=${#arrayPom[@]}
59      var=$((lengthArray - 1))
60      for i in $(seq 0 $var); do
61          common $pathFile ${arrayPom[$i]}
62          length=${#ris}
63          if [ $length -gt $maxLength ]; then
64              maxLength=$length
65              moduleName=$ris
66          fi
67      done
68      echo
69      $nameRepo,$linkRepo,$shaCommit,$moduleName,
          $className,$nameMe
70      thod >>$fileOutput
71      cd $BASEDIR
72      longPath=""
73      length=-1
74      maxLength=-1
75      ris=""
76  done <"$path"
```

L'output di questo script consiste in un file csv ("outputIDFlakies.csv") con la seguente intestazione:

- Name Repository;
- Link Repository;

- Sha della commit;
- Module Name;
- Class Name;
- Method Name.

In figura **??** è possibile osservare un esempio completo dell'output generato dallo script "IDFlakies.sh".

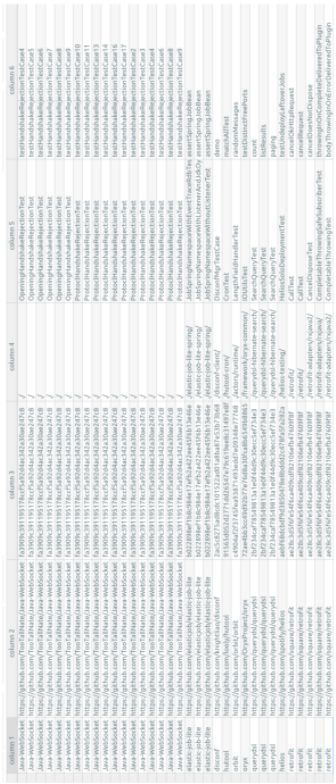


Figura 3.2: *Esempio di output generato da IDFlakies.sh*

FirstStep.sh 2.0

Per quanto riguarda "FirstStep.sh", è stato nuovamente sviluppato il meccanismo per eseguire la build del progetto. In questa nuova versione dello script la build del progetto viene eseguita solo nel momento in cui il repository non è posizionata sul branch di nostro interesse. Per individuare l'attuale branch, viene quindi effettuato il comando di git checkout seguito dallo sha individuato nel file di input. Se il comando dà esito positivo (ovvero viene segnalato il posizionamento su un nuovo branch), allora viene nuovamente eseguito il comando di mvn install, altrimenti, se dà esito negativo (ovvero, viene segnalato di essere già sul branch desiderato), non viene effettuata la build del progetto. Tale

controllo ha permesso di ottimizzare i tempi di esecuzione in quanto viene effettuata la build del sistema solo in caso di effettiva necessità. Infine, si è deciso di salvare all'interno del file csv di output anche il risultato della build del progetto al fine di rendere disponibile tale informazione per analisi future.

Di seguito è mostrato il codice per effettuare le operazioni descritte precedentemente.

```
1  #!bin/bash
2  checkout() {
3      #entra nella repo
4      local sha=$SHACOMMIT
5      git checkout $sha
6  }
7  cloneRepo() {
8      local nameRepo=$NAMEREPO
9      local linkRepo=$LINKREPO
10     local shaCommit=$SHACOMMIT
11     local baseDir=$BASEDIR
12     local repoFolder=$REPOFOLDER
13     cd $baseDir
14     cd $repoFolder
15     message=$(git clone $linkRepo 2>&1)
16     TOSEARCH="fatal: destination path"
17     toReturn=0
18     cd $nameRepo
19     if echo "$message" | grep -q "$TOSEARCH";
         then
20       actualSha=$(git rev-parse HEAD)
21       if echo "$actualSha" | grep -q "
           $shaCommit"; then
22         cd $baseDir
23         toReturn=0
```

```
24          else
25              git checkout $shaCommit
26              toReturn=1
27          fi
28      else
29          git checkout $shaCommit
30          toReturn=1
31      fi
32  }
33  mvnStep() {
34      local baseDir=$BASEDIR
35      TOSEARCH="BUILD␣FAILURE"
36      OUTPUTBUILD=""
37      message=$(mvn install -DskipTests -fn -B)
38      if echo "$message" | grep -q "$TOSEARCH";
          then
39          OUTPUTBUILD="BUILD␣FAILED"
40      else
41          OUTPUTBUILD="BUILD␣PASS"
42      fi
43      cd $baseDir
44      echo "$OUTPUTBUILD"
45  }
46  CSVINPUTFIRSTSTEP=$1
47  REPOFOLDER=$2
48  BASEDIR=$3
49  CSVOUTPUTFIRSTSTEP=$4
50  while IFS= read -r line; do
51      IFS=',' read -r -a array <<<"$line"
52      NAMEREPO=${array[0]}
53      LINKREPO=${array[1]}
54      SHACOMMIT=${array[2]}
```

```
55      MODULENAME=${array[3]}
56      cloneRepo "$NAMEREPO" "$LINKREPO" "
          $SHACOMMIT"
57      "$BASEDIR" "$REPOFOLDER"
58      echo $toReturn
59      if [ $toReturn -eq 1 ]; then
60          mvnStep $BASEDIR
61          echo
62          "$NAMEREPO","$LINKREPO","$SHACOMMIT","
              $OUTPUTBUILD" >>"$CSVOUTPUTFIRSTSTEP"
63      fi
64  done <"$CSVINPUTFIRSTSTEP"
```

Il file csv creato da questo script conterrà le seguenti informazioni:

- Name Repository;
- Link Repository;
- Sha della commit;
- Build result.

La figura 3.3 mostra parte del csv generato in output dal file "outFirstStep.sh".

column 1	column 2	column 3	column 4
sos	https://github.com/odrotbohm/sos	d4ed68a143b637c7e2d523edad1b105bf6c2d996	BUILD FAILED
openpojo	https://github.com/openpojo/openpojo	9badbcc4593e797accfed5e51bec9f2b843f0f67	BUILD FAILED
undertow	https://github.com/undertow-io/undertow	d0efffad5d2034bb07525cac9b299dac72c3045d	BUILD PASS
wro4j	https://github.com/wro4j/wro4j	185ab607f1d649ca38b4a772831ee754cd4649fb	BUILD PASS
redpipe	https://github.com/fromage/redpipe	0aff891d6befdf0dcc2bdbfda22262cdf82ac66f	BUILD PASS
vertexium	https://github.com/visallo/vertexium	f1fd59e23610312cf0ec6756ea01baaf56cdb74e	BUILD PASS
java-cas-client	https://github.com/apereo/java-cas-client	574b74fa64e4c95bda00ff41d06d358684a0b2e6	BUILD FAILED
redpipe	https://github.com/fromage/redpipe	0aff891d6befdf0dcc2bdbfda22262cdf82ac66f	BUILD PASS
vertexium	https://github.com/visallo/vertexium	f1fd59e23610312cf0ec6756ea01baaf56cdb74e	BUILD PASS
java-cas-client	https://github.com/apereo/java-cas-client	574b74fa64e4c95bda00ff41d06d358684a0b2e6	BUILD FAILED
c2mon	https://github.com/c2mon/c2mon	d80687b119c713dd177a58cf53a997d8cc5ca264	BUILD FAILED
vertx-completable-future	https://github.com/cescoffier/vertx-completable-future	011d3cd963b9a810c2e8e616da27bbfd8a3c717e	BUILD PASS
excelastic	https://github.com/codingchili/excelastic	6bb7884b1e70f9e588d75bdaf132a84f6e896a7a	BUILD PASS
rxjava2-extras	https://github.com/davidmoten/rxjava2-extras	d0315b6ecb0d24437ec0d440b6d768e89247d56c	BUILD PASS
tyrus	https://github.com/eclipse-ee4j/tyrus	d86e0cb0a4f26ba89daf8668137c156e57fea565	BUILD PASS
esper	https://github.com/espertechinc/esper	590fa9c9eb854f1420b9d337b802aca19f963cc0	BUILD FAILED
yawp	https://github.com/feroult/yawp	b3bcf9c958d51b3b3f8fdd0ccc26923d5a531da1	BUILD PASS
luwak	https://github.com/flaxsearch/luwak	c27ec08c803db6ca6be1b6e8017f45667e603161	BUILD PASS
fluent-logger-java	https://github.com/fluent/fluent-logger-java	da14ec349bf0904da9865940b57b87563616ee04	BUILD PASS
delight-nashorn-sandbox	https://github.com/javadelight/delight-nashorn-sandbox	da35edc0a75424bad8cbf60959fae253202154c4	BUILD PASS
db-scheduler	https://github.com/kagkarlsson/db-scheduler	4a8a28e60640dcd03e8d3bf212f483d35b0b6310	BUILD PASS
one	https://github.com/lcw2004/one	7b9b249ab1c09f39be885f95a27a756288be4548	BUILD PASS
sawmill	https://github.com/logzio/sawmill	e493c2e279b8820c0fdc2c79ab760d6759eb5a4a	BUILD FAILED
spring-cloud-zuul-ratelimit	https://github.com/marcosbarbero/spring-cloud-zuul-ratelimit	ae9603f45af05c4a0f940d81b313f273f62f3464	BUILD PASS
timely	https://github.com/nationalsecurityagency/timely	3a8cbd3378cffdc96aad1c52e9b0b2e1481ca45f	BUILD PASS
sos	https://github.com/odrotbohm/sos	d4ed68a143b637c7e2d523edad1b105bf6c2d996	BUILD FAILED
openpojo	https://github.com/openpojo/openpojo	9badbcc4593e797accfed5e51bec9f2b843f0f67	BUILD FAILED
oci-java-sdk	https://github.com/oracle/oci-java-sdk	2e9982f268420efd4010d8dc8370e9d9ade67ec2	BUILD FAILED
aletheia	https://github.com/outbrain/aletheia	024a4b7ea4c186329bbe1e20bb31048d95add836	BUILD PASS
pippo	https://github.com/pippo-java/pippo	ae898b6c478af7365b04c2f5bf9331abf496b166	BUILD PASS
recast4j	https://github.com/ppiastucki/recast4j	01d478e190b731d2fe861e3b969b4c8dd04f0ede	BUILD PASS
noxy	https://github.com/spinn3r/noxy	d53a49421f385c70b5abe7e8cda84ff3a7b59c71	BUILD PASS
spring-cloud-aws	https://github.com/spring-cloud/spring-cloud-aws	64ec6d15c67e98c3372ab88c9e5b31d5a89a68f8	BUILD PASS
vertx-mqtt	https://github.com/vert-x3/vertx-mqtt	c8213e0f41150a1a02f4ad1185762494b52c6350	BUILD FAILED
vertx-rabbitmq-client	https://github.com/vert-x3/vertx-rabbitmq-client	7305bcd5173cd32a0d2b382a69bdb7027851bd10	BUILD PASS

Figura 3.3: *File creato da OutFirstStep.sh*

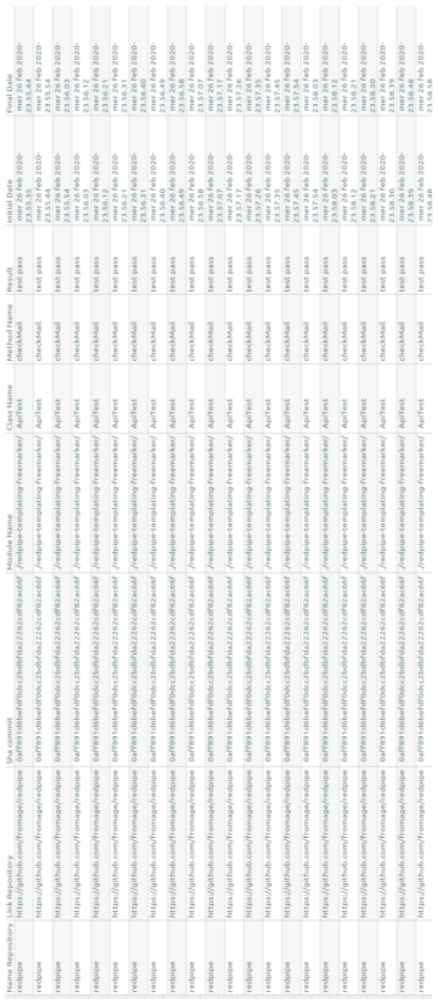
SecondStep.sh 2.0

Per ottenere maggiori dettagli dallo script "SecondStep.sh", si è deciso di generare un file csv intermedio per ogni metodo analizzato. Tale file contiene al suo interno le informazioni sulle N iterazioni eseguite su quel determinato caso di test. Inoltre, si è deciso di salvare su un file di log lo stato della macchina prima e dopo l'esecuzione del metodo ed il tempo di inizio e quello di fine per ogni esecuzione effettuata, come meglio dettagliato di seguito. Tali metodi hanno permesso di ottenere dei dettagli aggiuntivi che potrebbero servire per una futura analisi. Le informazioni salvate in questo csv intermedio sono:

- Name Repository;
- Link Repository;

- Sha della commit;
- Module Name;
- Class Name;
- Method Name;
- Result dell'esecuzione;
- Data di inizio esecuzione;
- Data di fine esecuzione.

La figura 3.4 mostra un esempio di file csv intermedio. Le informazioni raccolte nel csv intermedio sono state poi riassunte in un csv finale in cui in ogni riga vengono riportate le informazioni sul totale delle iterazioni, sul numero di successi e numero di fallimenti per quel determinato caso di test. In figura 3.5 viene riportata un estratto del csv finale generato.



Name Repository	Link Repository	Sha commit	Module Name	Class Name	Method Name	Result	Initial Date	Final Date
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.55.44	mer 26 Feb 2020-23.55.54
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.55.54	mer 26 Feb 2020-23.56.03
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.03	mer 26 Feb 2020-23.56.12
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.12	mer 26 Feb 2020-23.56.21
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.21	mer 26 Feb 2020-23.56.31
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.31	mer 26 Feb 2020-23.56.40
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.40	mer 26 Feb 2020-23.56.49
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.49	mer 26 Feb 2020-23.56.59
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.56.59	mer 26 Feb 2020-23.57.07
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.07	mer 26 Feb 2020-23.57.17
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.17	mer 26 Feb 2020-23.57.26
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.26	mer 26 Feb 2020-23.57.35
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.35	mer 26 Feb 2020-23.57.45
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.45	mer 26 Feb 2020-23.57.54
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.57.54	mer 26 Feb 2020-23.58.03
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.03	mer 26 Feb 2020-23.58.12
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.12	mer 26 Feb 2020-23.58.21
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.21	mer 26 Feb 2020-23.58.30
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.30	mer 26 Feb 2020-23.58.39
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.39	mer 26 Feb 2020-23.58.48
redpoje	https://github.com/fromage/redpoje	0aff991ddbebfdf0ddcc2bdbf0da22262cdf82ac6d	/redpoje-templating-freemarker/	ApiTest	checkMail	test pass	mer 26 Feb 2020-23.58.48	mer 26 Feb 2020-23.58.56

Figura 3.4: *Esempio del csv intermedio*

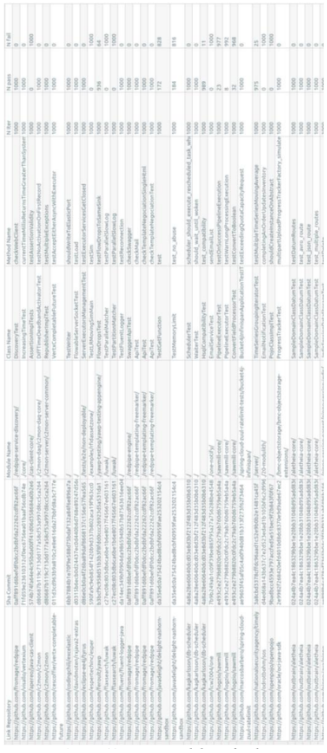


Figura 3.5: *Esempio del csv finale*

Di seguito è riportato il codice dello script “secondStep.sh”.

```
1  #!bin/bash
2  searchFlaky() {
3      local concatName=$CLASSNAME"#"$METHODNAME
4      local nrounds=$NUMBERSROUNDS
5      local nameRepo=$NAMEREPO
6      local moduleName=$MODULENAME
7      local baseDir=$BASEDIR
8      local RepoFolder=$REPOFOLDER
9      local csvOutput=$CSVOUTPUT
10     local resultTest=""
11     local shaCommit=$SHACOMMIT
12     local linkRepo=$LINKREPO
```

```
13      local outputLog=$OUTPUTLOG
14      local stateLog=$STATELOG
15      PASSTEST=0
16      TESTFAIL=0
17      TOSEARCH="BUILD␣SUCCESS"
18      i=0
19      for i in $(seq 1 $nrounds); do
20          cd $baseDir
21          cd $RepoFolder
22          cd $nameRepo
23          cd $moduleName
24          timestampInitial=$(date +%s)
25          timestampInitialDate=$(date -d
              @$timestampInitial)
26          echo "exec␣command:␣mvn␣-Dtest="
              $concatName"␣test" $i
27          stateMachineInitial=$(vmstat -t)
28          dirLog="$baseDir""$outputLog""$CLASSNAME
              "_"$METHODNAME"".txt"
29          message=$(mvn -Dtest=$concatName test)
30          echo "$message" >>"$dirLog"
31          stateMachineFinal=$(vmstat -t)
32          echo "$stateMachineInitial", "
              $stateMachineFinal" >>"$baseDir""
              $stateLog""$CLASSNAME""_""$METHODNAME"
              ".txt"
33          timestampFinal=$(date +%s)
34          timestampFinalDate=$(date -d
              @$timestampFinal)
35          if echo "$message" | grep -q "$TOSEARCH"
              ; then
36              echo "test␣pass"
```

```
37              resultTest="test␣pass"
38              PASSTEST=$((PASSTEST + 1))
39          else
40              resultTest="test␣fail"
41              echo "test␣fail"
42              TESTFAIL=$((TESTFAIL + 1))
43          fi
44          echo "$nameRepo","$linkRepo", "
              $shaCommit","$moduleName",
45          "$CLASSNAME","$METHODNAME","$resultTest"
              ,
46          $timestampInitialDate,
              $timestampFinalDate >>"$baseDir""
              $csvOutput""$CLASSNAME""_""$METHODNAME
              "".csv"
47      done
48      echo "number␣of␣pass:␣"$PASSTEST
49      echo "number␣of␣failure:␣"$TESTFAIL
50      if [ $PASSTEST -gt 0 -a $TESTFAIL -gt 0 ];
          then
51          echo "test␣flaky"
52          echo "numbers␣of␣runs:"$i
53      else
54          echo "test␣isn't␣flaky"
55      fi
56  }
57  CSVINPUTFIRSTSTEP=$1
58  CSVOUTPUT=$2
59  BASEDIR=$3
60  REPOFOLDER=$4
61  NITER=$5
62  NUMBERSROUNDS=$6
```

```
63 CSVOUTPUTFINAL=$7
64 OUTPUTLOG=$8
65 STATELOG=$9
66 while IFS= read -r line; do
67     IFS=',' read -r -a array <<<"$line"
68     NAMEREPO=${array[0]}
69     LINKREPO=${array[1]}
70     SHACOMMIT=${array[2]}
71     MODULENAME=${array[3]}
72     CLASSNAME=${array[4]}
73     METHODNAME=${array[5]}
74     searchFlaky $NAMEREPO $CLASSNAME $METHODNAME
75     $NUMBERSROUNDS $BASEDIR $REPOFOLDER
         $CSVOUTPUT $SHACOMMIT
76     $LINKREPO $OUTPUTLOG $STATELOG
77     cd $BASEDIR
78     echo
79     $LINKREPO,$SHACOMMIT,$MODULENAME,$CLASSNAME,
         $METHODNAME,
80     $NUMBERSROUNDS,$PASSTEST,$TESTFAIL >>"
         $CSVOUTPUTFINAL"".csv"
81 done <"$CSVINPUTFIRSTSTEP"
```

Com'è possibile notare, la funzione *searchFlaky()* si posiziona all'interno del module name indicato nel file csv ed effettua le seguenti operazioni:

1. Salva all'interno della variabile timestampInitial il *timestamp* attuale e all'interno della variabile *timestampInitialDate* il timestamp convertito nel formato "Date";

2. Salva all'interno della variabile stateMachineInitial lo stato attuale della macchina prima dell'esecuzione del test, ottenuto con il comando *vmstat -t*;

3. Esegue il comando *mvn -Dtest=$concatName test* e salva all'interno di un file di log il risultato;

4. Riesegue il comando *vmstat -t* per tener traccia dello stato della macchina dopo l'esecuzione del test;

5. Calcola il nuovo timestamp di fine esecuzione e lo salva nella variabile *timeStampFinal*;

6. Controlla l'output del comando *mvn -Dtest* per verificare il risultato dell'esecuzione;

7. Se all'interno dell'output è presente la stringa "*BUILD SUCCESS*" allora verrà incrementato il contatore della variabile *PASSTEST* di una unità, altrimenti verrà incrementato di un'unità il contatore *FAILTEST*;

8. Alla fine di ogni esecuzione vengono salvate le informazioni sul nome e il link del repository, lo *sha* della commit, il nome della classe, il nome del metodo, il risultato ottenuto in quella iterazione ed infine il timestamp iniziale e finale entrambi convertiti in formato "Date".

Una volta terminata la funzione *searchFlaky* viene creato un csv riassuntivo di tutte le varie iterazioni in cui sono riportate le informazioni sul numero totale di test che hanno avuto successo ed il numero totale di test che hanno avuto un fallimento come risultato.

3.2.4. **Esecuzione degli script su metodi OD**

Sono stati successivamente provati gli script anche sui casi di test etichettati come OD all'interno del csv rilasciato da **iDFlakies**. Lo scopo di questo test è stato quello di verificare l'effettiva dipendenza dall'ordine di esecuzione dei metodi da loro indicati. Su ogni metodo testato, l'esperimento ha dato esito negativo (mostrando o tutti "pass" o tutti

"fail"). Tale esperimento ha effettivamente comprovato che in quei casi il flaky riscontrato dipendeva dall'ordine in cui veniva eseguito il test, pertanto verranno inseriti all'interno del dataset finale attenendosi alla classificazione data da **iDFlakies**.

3.2.5. Grafico dei dati ottenuti

L'ultima operazione effettuata è stata la creazione di grafici per i risultati ottenuti. Per effettuare questa fase sono stati individuati i *flaky test* presenti all'interno del file "finalCSV.csv" e sono stati considerati i corrispettivi file csv intermedi; si è poi andati a generare i grafici corrispondenti. Per effettuare questa fase è stato creato uno script Python per caricare i dati all'interno di un database e successivamente è stato utilizzato un tool per visualizzarli.

Creazione dello script

Lo script Python è stato suddiviso in due moduli. Il primo modulo legge tutto il contenuto di una directory e genera in output una lista contenente tutti i file in formato csv presenti all'interno. Successivamente tramite un'apposita libreria i dati sono stati manipolati (es. sostituendo la dicitura: "test pass" con il valore 1 e "test fail" con il valore 0) e poi sono stati passati al modulo denominato "load_data_influxDB". Tale metodo sfruttando le API di InfluxDB carica i dati all'interno del database.

InfluxDB e Chronograf

Per l'immagazzinamento dei dati è stato individuato il database "InfluxDB". Tale database è particolarmente indicato nel momento in cui bisogna rappresentare dati temporali. Nello scegliere il tool da adoperare per graficare i dati si è optato per "**Chronograf**" in quanto è un tool sviluppato dalla stessa software house di InfluxDB.

Di seguito vengono mostrati alcuni dei grafici generati.

Il grafico in figura3.6 mostra l'andamento del metodo "*runReconnectBlockingScenario2*" appartenente alla classe "*Issue256Test*" del progetto "**Java-WebSocket**".

Figura 3.6: *Andamento di uno dei metodi flaky*

Com'è possibile osservare dal grafico, tale metodo presenta quattro fallimenti su mille esecuzioni. Il grafico in figura 3.7 mostra il fallimento del metodo "*testBulkClusterJoining*" appartenente alla classe "*ZKTest*" del progetto "**Noxy**".

Figura 3.7: *Andamento del metodo testBulkClusterJoining*

Il metodo preso in esame presentava una serie di successi consecutivi (152) per poi iniziare una serie di fallimenti (848). Il metodo quindi è stato ritestato ma ha dato sempre risultati simili a quello mostrato precedentemente. L'ultimo metodo che si vuole illustrare è del progetto "timely". Il metodo in questione è: "*testMultipleTimeSeriesMovingAverage*" appartenente alla classe "*TimeSeriesGroupingIteratorTest*". Tale metodo risulta molto interessante poiché mostra un fallimento ogni volta che scatta il 58esimo minuto di un'ora. La figura 3.8 mostra l'esecuzione del metodo Per ottenere una maggiore sicurez-

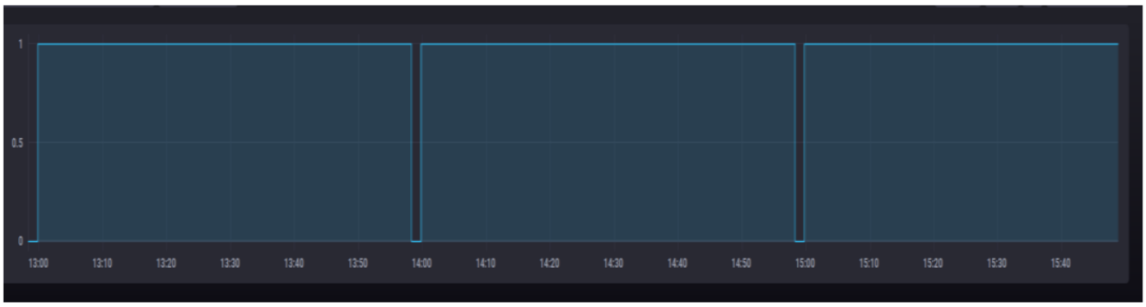


Figura 3.8: *Andamento di testMultipleTimeSeriesMovingAverage*

za, si è quindi proceduto ad eseguire altre due volte il metodo sopra elencato. Le figure 3.9 e 3.10 mostrano i risultati delle altre esecuzioni effettuate.

Figura 3.9: *Seconda esecuzione del test in analisi*

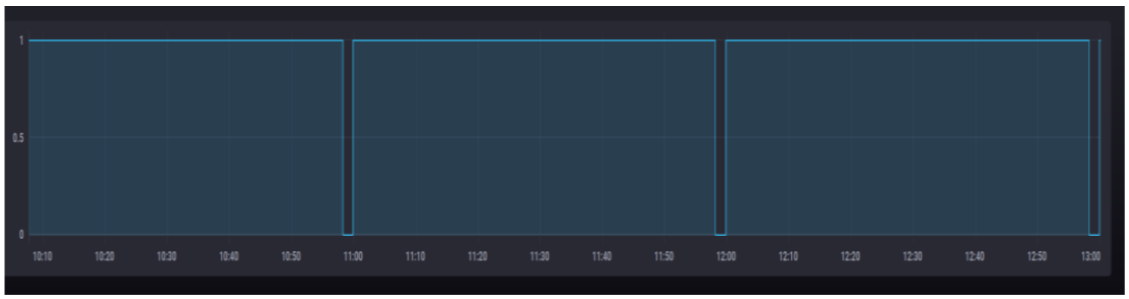


Figura 3.10: *Terza esecuzione del test in analisi*

Possiamo quindi considerare questo tipo di flaky "deterministico" in quanto presenta un pattern ben preciso.

3.2.6. **Root cause più frequenti**

Sono state infine analizzate le root cause dei flaky test individuati tramite una analisi statica del codice. Tale analisi è stata fatta sui repository con comportamenti più "stabili" (ovvero un numero bilanciato sia di "pass" che di "fail"), e sono emersi i seguenti risultati:

- 99% Errore di Network;
- 1% Multithreading.

Tali risultati andrebbero però inquadrati in contesti più ampi e con più dati a disposizione.

3.2.7. **Limiti della ricerca**

Durante tutta la fase di ricerca sono emersi diversi limiti, alcuni dei quali sono stati trattati precedentemente, altri invece vengono presentati in questa sezione.

- Lentezza dell'esecuzione della build: In alcuni casi sono state analizzati progetti la cui build ha impiegato oltre due/tre ore per terminare. Tale problematica è sintomo di sistemi di building ancora non adeguatamente efficienti;

- Lentezza nell'eseguire un singolo caso di test: In alcuni casi sono stati impiegati svariati minuti per eseguire un singolo caso di test; da una indagine effettuata, il sistema di build di Maven riconfigura parte dell'ambiente ad ogni singola esecuzione. Tale riconfigurazione può impiegare anche diversi minuti;

- Limiti hardware: Gli esperimenti sono stati eseguiti su una macchina Ubuntu 16.04 4GB di RAM 2. Intel Core i3-2100 CPU 3.10GHzx4;

- Sha non validi: In alcuni casi non c'è stata la possibilità di eseguire l'allineamento sul branch indicato dal dataset in quanto è stato restituito il seguente errore: "reference is not a tree" nel momento in cui è stato eseguito il comando "git checkout".

4. Conclusioni

Nella prima parte della tesi è stata effettuato un mapping study sistematico della letteratura. Questa ricerca ha evidenziato una conoscenza ancora in stato embrionale sull'argomento ed in molti casi una carenza di dati. È emerso che nel tempo sono stati rilasciati pochi dataset sulla problematica trattata e prevalentemente di natura open-source. Inoltre, quasi la totalità degli studi si è concentrata su un unico linguaggio di programmazione (**Java**) e su un unico sistema di building (**Maven**). Non sempre sono state descritte in maniera accurata le tecniche utilizzate per la costruzione dei dataset (in alcuni casi sono stati riutilizzati dei dataset preesistenti), inoltre le *root cause* sono state individuate in maniera parziale in quanto in alcuni articoli più recenti ne sono comparse di nuove (soprattutto per sistemi operativi mobile) che non erano ancora state individuate.

Si è quindi deciso di contribuire alla costruzione di un nuovo tool per l'identificazione dei *flaky test* indipendenti dall'ordine. Partendo dal dataset già rilasciato dagli sviluppatori di **iDFlakies**, sono state analizzati centosessantasei casi di test classificati come non dipendenti dall'ordine. Di questi, trentuno hanno mostrato un comportamento "flaky". Sono state poi analizzate le root cause di questi metodi ed è emerso che nella quasi totalità dei casi (trenta *flaky test*) la *root cause* era legata al network, mentre soltanto per un *flaky test* la *root cause* era legata al multithreading. Il tool ha permesso di ottenere informazioni più dettagliate su particolari metodi "flaky", come lo stato della macchina prima e dopo l'esecuzione del test e il timestamp di ogni

singola iterazione. Inoltre, il tool ha evidenziato anche i limiti degli attuali sistemi di build che in molti casi risultano poco ottimizzati per affrontare adeguatamente lo studio della problematica in esame.

Infine, i risultati sono stati immagazzinati all'interno di una base di dati e sono stati rappresentati sotto forma di grafici in modo da evidenziare eventuali pattern. Dai risultati è emerso che in alcuni casi è stato riscontrato effettivamente la presenza di un pattern, aprendo la strada all'ipotesi che alcuni flaky test abbiano un comportamento deterministico. I risultati ottenuti dovranno essere ulteriormente approfonditi. Gli sviluppi futuri del lavoro saranno:

- "Instrumentation" del codice per poter monitorare ed avere maggiore controllo sui casi di test che sono in esecuzione;
- Ampliare il tool in modo da poter individuare flaky test dipendenti dall'ordine;
- Rendere il tool indipendente dal linguaggio di programmazione in cui sono scritti i progetti in analisi.

A. Appendice A

Di seguito sono riportati i riferimenti bibliografici degli articoli usati per la revisione sistematica della letteratura.

[S1] Qingzhou L., Farah H., Lamyaa E., Darko M. (2014) "An Empirical Analysis of Flaky Tests". Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014.

S2] Jonathan B., Gail K., Eric M., Mohan D. (2015) "Efficient Dependency Detection for Safe Java Test Acceleration". Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015.

[S3] Zebao G., Yalan L., Myra B. C., Atif M. M., Zhen W. (2015) "Making system user interactive tests repeatable: when and what should we control?". ICSE '15: Proceedings of the 37th International Conference on Software Engineering – Volume 1.

[S4] Alex G., August S., Farah H., Darko M. (2015) "Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency". Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore MD, USA, 2015.

[S5] Milos G., Lamyaa E., Darko M. (2015) "Practical regression test

selection with dynamic file dependencies". ISSTA 2015: Proceedings of the 2015 International Symposium on Software Testing and Analysis.

[S6] Lam W, Zhang S, Ernst MD (2015) "When tests collide: Evaluating and coping with the impact of test dependence".

[S7] Shi A., Gyori, A., Legunsen O., Marinov D. (2016) "Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications". Proceedings of the 2016 International Conference on Software Testing, Verification and Validation, USA.

[S8] Thomas R., Waldemar H., Philipp L., Stefan S (2017). "An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-based Open-source Software". Proceedings of the 14th International Conference on Mining Software Repositories, Buenos Aires, Argentina, 2017.

[S9] Candido, J., Melo L., D'Amorim, M. (2017) "Test suite parallelization in open-source projects: A study on its usage and impact". ASE 2017 – Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.

[S10] Fabio P., Andy Z. (2015) "Does Refactoring of Test Smells Induce Fixing Flaky Tests?". International Conference on Software Maintenance and Evolution, Shanghai, China, 2017.

[S11] Labuschagne, A., Inozemtseva, L., Holmes, R. (2017) "Measuring the cost of regression testing in practice: a study of Java projects using continuous integration". Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering August 2017.

[S12] Jonathan B., Owolabi L., Michael H., Lamyaa E., Tifany Y.,

Darko M. (2018) “DeFlaker: Automatically Detecting Flaky Tests”. Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 2018.

[S13] Swapna T., Chandani S., Na M. (2018) “An Empirical Study of Flaky Tests in Android”. International Conference on Software Maintenance and Evolution, Madrid, Spain, 2018.

[S14] Md T. R., Peter C. R. (2018) “The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds”. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista FL, USA, 2018.

[S15] Gambi, A., Bell, J., Zeller, A. (2018) “Practical Test Dependency Detection Proceedings”. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018.

[S16] Eck M., Palomba F., Castelluccio M., Bacchelli A. (2019) “Understanding Flaky Tests: The Developer’s Perspective”. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019.

[S17] Wing L., Patrice G., Suman N., Anirudh S., Suresh T. (2019) “Root Causing Flaky Tests in Large-scale Industrial Setting”. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 2019.

[S18] Armin N., Peter C. R.,Weiyi S. (2019) “Bisecting commits and modeling commit risk during testing”. ESEC/FSE 2019: Proceedings

of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

[S19] Kai Presler-M., Eric H., Sarah H., Kathryn S. (2019) "Wait, Wait. No, Tell Me. Analysing Selenium Configuration Effects on Test Flakiness". 14th International Workshop on Automation of Software Test, Montreal, QC, Canada, 2019.

[S20] August S., Wing L., Reed O., Tao X., Darko M. (2019) " iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests" In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019.

[S21] Biagiola M., Stocco A., Mesbah A., Ricca F., Tonella P. (2019) "Web test dependency detection". ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

[S22] August S., Jonathan B., Darko M. (2017) "Mitigating the Effects of Flaky Tests on Mutation Testing". Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 2017.

[S23] Andreas S., Marvin K., Gordon F. (2019) "Testing Scratch Programs Automatically". Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019.

[S24] Pontes F., Gheyi R., Souto S., Garcia A., Ribeiro M. (2019) "Ja-

va reflection API: revealing the dark side of the mirror". ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

[S25] Böhme M. (2019) "Assurance in software testing: a roadmap". ICSE- NIER '19: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results.

[S26] Schwahn O., Coppik N., Winter S., Suri N. (2019) "Assessing the state and improving the art of parallel testing for C". ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.

[S27] Zhiyu F., (2019) "A Systematic Evaluation of Problematic Tests Generated by EvoSuite". 41st International Conference on Software Engineering: Companion Proceedings, Montreal, QC, Canada, 2019.

[S28] Wing L., Reed O., August S., Darko M., Tao X. (2019) "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests". 12th IEEE Conference on Software Testing, Validation and Verification, Xi'an, China, 2019.

[S29] Morán, J., Augusto, C., Bertolino, A., De La Riva, C., Tuya, J. (2019) "Debugging flaky tests on web applications" WEBIST 2019 – Proceedings of the 15th International Conference on Web Information Systems and Technologies.

[S30] Palomba F., Zaidman A. (2019)" The Smell of Fear: On the Relation Between Test Smells and Flaky Tests". Empirical Software Engineering, Salerno, Italy, 2019.

Bibliografia