



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

Candidato:

Giammaria Giordano

Matr.: 0522500509

ANNO ACCADEMICO 2018/2019

Indice

1	CAPITOLO 1	4
1.1	Il testing	4
1.1.1	Testing unitario	5
1.1.2	Testing di integrazione	5
1.1.3	Testing di accettazione	6
1.2	Metodologia Agile	6
1.3	Configuration Management	7
1.3.1	Version Control	8
1.3.2	System building	8
1.3.3	Change management	10
1.3.4	Release Management	10
1.4	Continuous Integration	11
1.5	Regression Testing	12
1.6	Problemi di building	13
1.7	Flaky test	14
1.7.1	Esempio di flaky test	17
2	Titolo capitolo	20
2.1	Mapping	20
2.2	Mapping study flaky test	21
2.2.1	Domande di ricerca	21
2.2.2	Criteri di inclusione/esclusione	24

2.2.3	Snowball	25
2.2.4	Valutazione della qualità	25
2.2.5	Estrazione dei dati	28
2.2.6	Minacce alla validità	30
2.2.7	Analisi dei dati e classificazione	32
2.2.8	Conclusioni	43

Elenco delle figure

1.1	<i>Esempio flaky test</i>	17
1.2	<i>Snippet di codice modificato</i>	18
1.3	<i>Codice modificato</i>	19
1.4	<i>Versione finale del codice</i>	19
2.1	<i>Sintesi della selezione degli articoli</i>	30
2.2	<i>Frequenza di pubblicazione del dataset</i>	33
2.3	<i>Numero di progetti su cui sono stati condotti gli studi</i>	34
2.4	<i>Distribuzione dei linguaggi di programmazione analizzati</i>	35
2.5	<i>Sistemi di building utilizzati</i>	36
2.6	<i>Strategie usate per la costruzione del dataset</i>	37
2.7	<i>Root cause dei flaky test</i>	39
2.8	<i>Strategia per individuare i flaky test</i>	41
2.9	<i>Ambito di ricerca per ogni artefatto</i>	42
2.10	<i>Frequenza di pubblicazione suddivisa per anni</i>	43

Elenco delle tabelle

Introduzione

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 dall'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 a quelli 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 simulato¹. 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

¹Con 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;

²Il 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 software³. Un sistema 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;

³Con 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 collaudo, la messa a punto, l'installazione ed eventualmente la fase di manutenzione del software (se prevista da contratto).

- Un programma di supporto per facilitare l'installazione;
- Manuale di installazione;
- Manuale di avvio.

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 memo-

ria, 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:

```
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*

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 preconditione del test, ma in generale errori di questo tipo possono capitare nei casi più disparati (es. in ambito 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. Titolo capitolo

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 rilette 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 selezione 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.

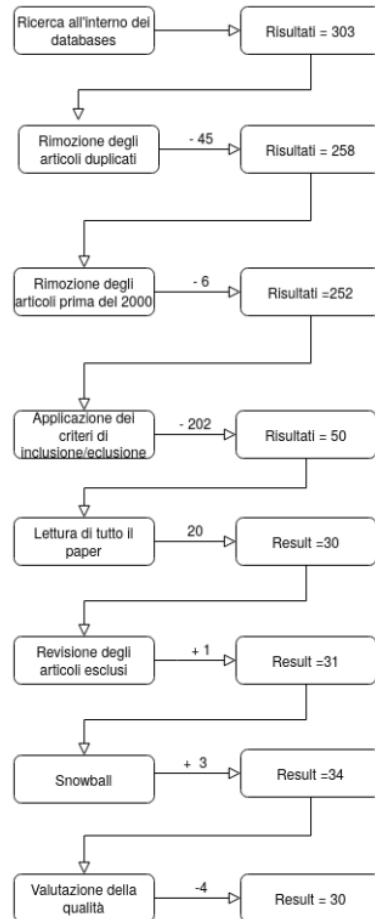


Figura 2.1: *Sintesi della selezione degli articoli*

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.

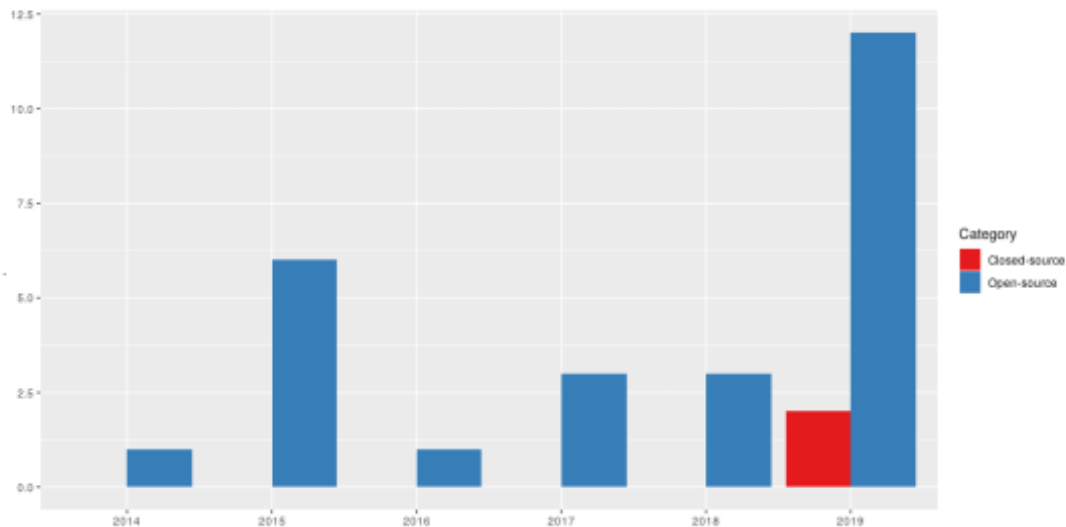


Figura 2.3: *Numero di progetti su cui sono stati condotti gli studi*

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.

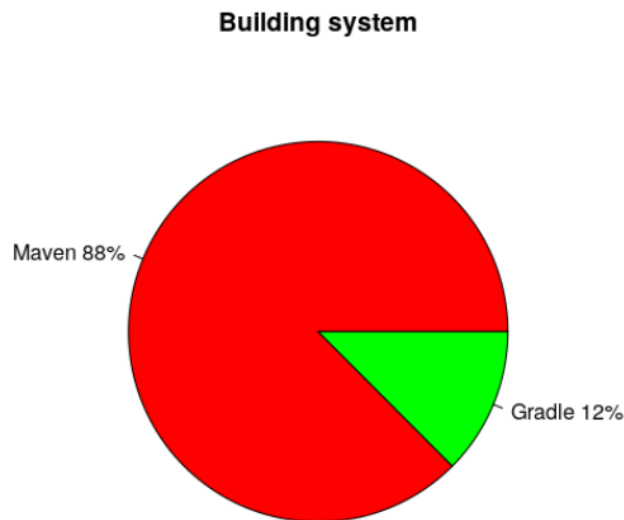


Figura 2.5: *Sistemi di building utilizzati*

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.

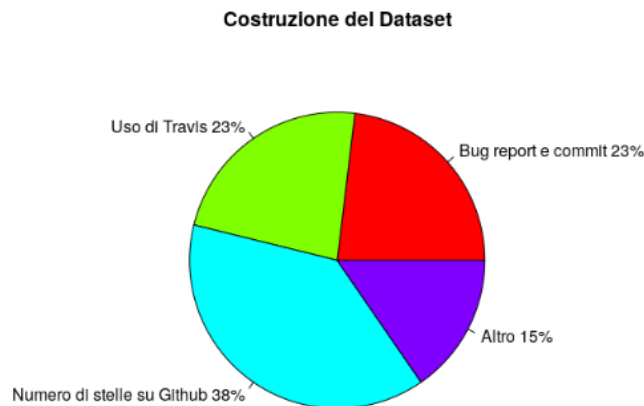


Figura 2.6: *Strategie usate per la costruzione del dataset*

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.

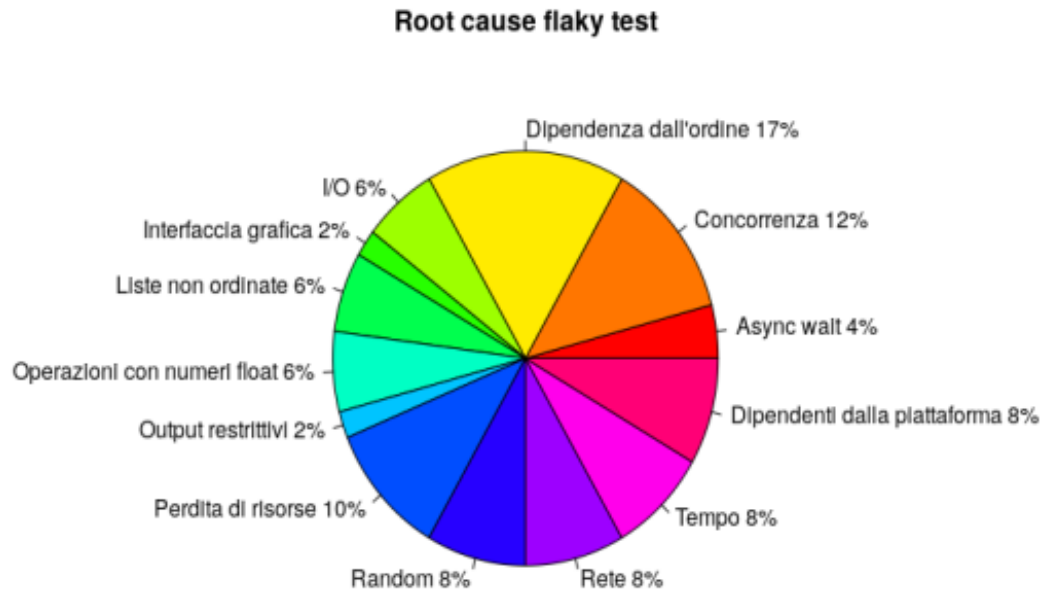


Figura 2.7: *Root cause dei flaky test*

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.

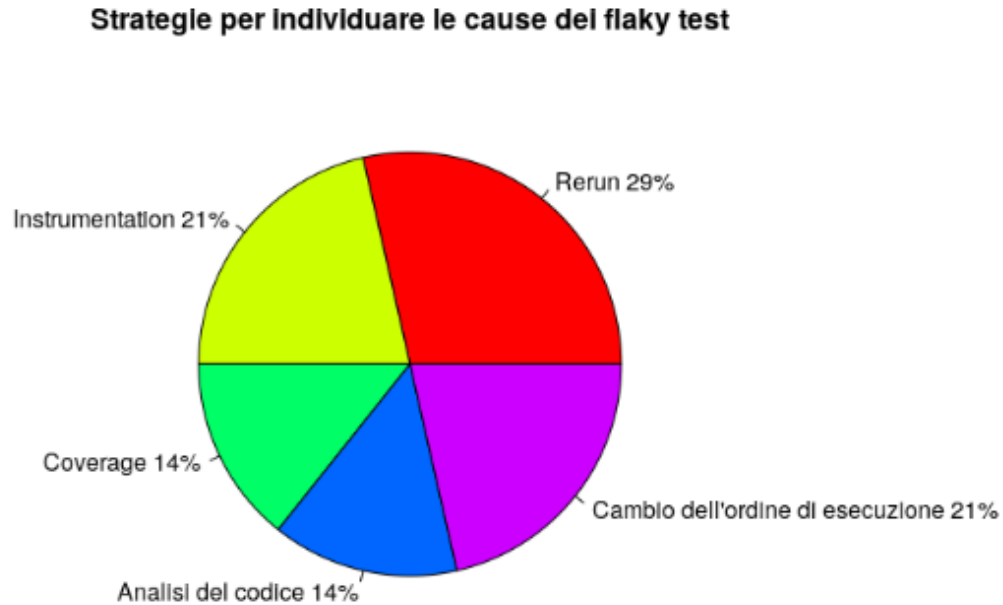


Figura 2.8: *Strategia per individuare i flaky test*

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.