Testing Suite Report

Progetti:

Bookkeeper e Syncope

Di Palma Flavio - 0366635

30 gennaio 2026

1 Introduzione

In questo report documento il design e l'implementazione di test case per due classi de progetti Apache BookKeeper e Syncope

L'intero processo di build e validazione è automatizzato su GitHub CI che esegue build Maven, test Surefire (unit test) / Failsafe (integration test), report Jacoco e mutation testing PITest, pubblicando inoltre i report come artifact ad ogni push

Per ciascuna classe mi sono concentrato maggiormente sulla metodologia. Ho definito test case minimali e significativi, ampliati con test generati da LLM e test di control flow (in seguito ad analisi jacoco) per coprire rami critici ignorati in precedenza. Ho validato i risultati tramite Jacoco, analizzando il miglioramento della copertura prima e dopo l'aggiunta dei test di control flow

N.B: In questo report mi sono concentrato sulla stesura della corretta metodologia dei casi di test, tutta (o quasi) l'implementazione è accuratamente dettagliata all'interno delle classi di test nelle repository. Il lavoro maggiore è stato fatto su Syncope, in quanto ho scelto classi più complesse rispetto a bk. Mockito è stato usato prevalentemente in Syncope, in quanto le classi che ho scelto su bookkeeper richiedevano l'estensione di PowerMock, che ho provato ad implementare ma con scarsi risultati

Parte I
Bookkeeper

Repository: https://github.com/Flaasks/bookkeeper

2 Classe: RecyclableArrayList<T>

2.1 Descrizione

RecyclableArrayList<T> consente il riciclo delle istanze per migliorare le performance attraverso il riuso di oggetti gia' allocati, riducendo la pressione sulla garbage collection

2.2 Category Partition

Ho identificato le seguenti categorie di partizione:

Tabella 1: Categorie di partizione per RecyclableArrayList

Categoria	**Scelte**	**Descrizione**
Stato lista	Empty, Single	0, 1
Indice	First, Last	Accessi validi e invalidi
Tipo elemento	Null, NonNull	Elemento null o non-null
Capacita'	WithinCapacity, ExpandNeeded	Array vs espansione
Parametro equals	Null, Same, Different, WrongType	Confronti

Ho selezionato 8 test che coprono i boundary values critici:

Tabella 2: Test Boundary Values per RecyclableArrayList

ID	**Metodo**	**Boundary**	**Motivazione**
T1	add()	Empty to Single	da 0 a 1
T2	add()	Tipo null	aggiunta null
T3	get(int)	Indice 0	Accesso minimo
T4	remove(int)	Single to Empty	da 1 a 0
T5	equals(Obj)	Parametro null	Boundary null
T6	get(int)	Out of bounds	Violazione confine
T7	recycle()	Non-pooled	Handle null
T8	add()	Expand capacity	Espansione array

3 Test LLM

Ho generato 4 test LLM per questa classe:

- **LLM-1**: remove() dopo recycle su lista vuota
- **LLM-2**: contains() con null e presenza/assenza
- **LLM-3**: iteratore su transizioni di stato
- **LLM-4**: subList() su boundary di indici

Strumento: Chat-GPT5 - Prompt utilizzato: (black-box su documentazione) "Data la documentazione di RecyclableArrayList [Link Documentazione], genera N test JUnit 4 black-box secondo metodologia category partition"

Considerazioni personali sull'uso dell'LLM L'LLM ha fornito scenari sensati senza conoscere l'implementazione, ma senza specificare davvero una category partition, si è focalizzata maggiormente sulla scrittura del test, tra l'altro proponendo valori non boundary. Inoltre ho dovuto rigenerare i test più volte affinché funzionassero, e spesso dichiara variabili che poi non usa realmente

4 Analisi Jacoco: Prima dei Test di Control Flow

Ho eseguito i 12 test iniziali (8 boundary + 4 LLM) e analizzato la copertura su RecyclableArrayList:

Tabella 3: Jacoco Coverage PRIMA dei test CF

Metrica	**Missed**	**Covered**	**Totale**	**Coverage**
Istruzioni	14	16	30	53%
Linee	5	7	12	58%
Branch	1	1	2	50%
Metodi	0	5	5	100%

Ho osservato che:

- istruzioni coperte al 53%: quasi la metà dei percorsi non testati
- branch coverage al 50%: un ramo critico non coperto (handle != null in recycle())
- linee al 58%: diverse linee rimangono non testate

5 Identificazione di Test di Control Flow

Ho quindi testato due rami critici:

1. **Handle non-null in recycle()**: il codice ha una condizione che non viene mai esercitata dai test boundary/LLM (che usano istanze non-pooled). Ho creato il test CF-1 che ottiene un'istanza pooled dal recycler e verifica che recycle() riutilizzi l'istanza e la svuoti

2. **Percorsi di equals()**: il metodo ha rami per type mismatch e size diversity non completamente coperti. Ho creato il test CF-2 che forza confronti con liste di size diverso e tipi errati

6 Test di Control Flow

Ho implementato 2 test di control flow mirati:

- **CF-1 - Pooled Recycling**: crea un'istanza pooled via RecyclableArrayList.Recycler, la riempie con elementi, chiama recycle() e verifica che l'istanza rimane valida
- **CF-2 - Equals Type Mismatch**: testa equals() con liste di size diverso e tipi errati

7 Analisi Jacoco: Dopo i Test di Control Flow

Ho eseguito i 14 test completi (8 boundary + 4 LLM + 2 CF) e analizzato il miglioramento:

Tabella 4: Jacoco Coverage DOPO i test CF (14 test: 8 boundary + 4 LLM + 2 CF)

Metrica	**Missed**	**Covered**	**Totale**	**Coverage**
Istruzioni	3	27	30	90%
Linee	1	11	12	92%
Branch	0	2	2	100%
Metodi	0	5	5	100%

Tabella 5: Confronto PRIMA vs DOPO i test CF

Metrica	**PRIMA (12 test)**	**DOPO (14 test)**	**Miglioramento**
Istruzioni	53% (16/30)	90% (27/30)	+37%
Linee	58% (7/12)	92% (11/12)	+34%
Branch	50% (1/2)	100% (2/2)	+50% (100% raggiunto)
Metodi	100%	100%	

I 2 test di control flow, come aspettato, hanno prodotto un miglioramento significativo

8 Classe: MathUtils

8.1 Descrizione

MathUtils e' una classe di utilita' con metodi statici:

- **signSafeMod(long, int)**: modulo con correzione del segno
- **findNextPositivePowerOfTwo(int)**: calcolo della prossima potenza di 2

8.2 Category Partition

Ho definito le categorie focalizzandomi esclusivamente su boundary values ai bordi delle categorie: valori ai bordi come -1, 0, 1 per i long, e 1, 2, 3 per le potenze (1 è il valore minimo, 2 è una potenza esatta, 3 è la prima approssimazione) Senza utilizzare valori casuali nel range

Tabella 6: per l'implementazione guardare repository

Categoria	**Scelte Boundary**	**Descrizione**
Partizione 1: Segno dividendo	-1, 0, 1	limite del dividendo in signSafeMod
Partizione 2: Divisore minimo	divisore = 1	minimo valido per signSafeMod
Partizione 3: Potenze minime	value = 1, 2, 3	minimo, potenza, non potenza

9 Test LLM

Ho generato 4 test LLM con prompts di category partition:

- **LLM-1**: signSafeMod edge cases (divisor=1, large values)
- **LLM-2**: findNextPowerOfTwo boundary piccoli e grandi
- **LLM-3**: signSafeMod negativo stress test
- **LLM-4**: elapsedTime monotonicity verification

Prompt utilizzato (black-box su documentazione) "Considera la documentazione di MathUtils e genera test JUnit 4 black-box category partition, focalizzati sui boundary values"

Considerazioni personali sull'uso dell'LLM Anche in questo caso Chat-GPT ha preferito scrivere direttamente i test. L'LLM ha bisogno di prompt espliciti sui boundary per evitare valori casuali. Risulta molto utile per generare rapidamente scheletri di test black-box, ma va verificato che i casi rispettino le partizioni decise manualmente e non introducano assunzioni su dettagli interni

10 Analisi Jacoco: Prima dei Test di Control Flow

Ho eseguito i 7 test iniziali (3 boundary + 4 LLM) e analizzato la copertura su MathUtils:

Tabella 7: Jacoco Coverage PRIMA dei test CF (7 test: 3 boundary + 4 LLM)

Metrica	**Missed**	**Covered**	**Totale**	**Coverage**
Istruzioni	21	23	44	52%
Linee	5	5	10	50%
Branch	0	2	2	100%
Metodi	4	3	7	42.9%

Ho osservato che:

- istruzioni coperte al 52%: copertura limitata, 21 istruzioni rimangono non coperte
- branch coverage gia' al 100%: entrambi i rami sono coperti
- linee al 50%: meta' delle linee non sono visitate
- metodi al 42.9%: quattro metodi rimangono non testati (nowInNano(), elapsedMicroSec(), elapsedMSec(), elapsedNanos())

11 Identificazione di Test di Control Flow

Analizzando il report Jacoco, ho identificato metodi non coperti dai test boundary e LLM:

1. **nowInNano()**: metodo mai invocato che ritorna `System.nanoTime()`. Ho implementato CF-1 per aumentare la coverage
2. **elapsedMicroSec()**: metodo mai testato che converte nanosecondi in microsecondi tramite `TimeUnit.NANOSECONDS.toMicros()`. Ho creato CF-2 per testare questa conversione

12 Analisi Jacoco: Dopo i Test di Control Flow

Ho eseguito i 9 test completi (3 boundary + 4 LLM + 2 CF) e analizzato il miglioramento

Tabella 8: Jacoco Coverage DOPO i test CF (9 test: 3 boundary + 4 LLM + 2 CF)

Metrica	**Missed**	**Covered**	**Totale**	**Coverage**
Istruzioni	9	35	44	79%
Linee	2	8	10	80%
Branch	0	2	2	100%
Metodi	2	5	7	71.4%

Tabella 9: Confronto PRIMA vs DOPO i test CF

Metrica	**PRIMA (7 test)**	**DOPO (9 test CF)**	**Variazione**
Istruzioni	52% (23/44)	79% (35/44)	+27% (+12 istruzioni)
Linee	50% (5/10)	80% (8/10)	+30% (+3 linee)
Branch	100% (2/2)	100% (2/2)	Stabile
Metodi	42.9% (3/7)	71.4% (5/7)	+28.5% (+2 metodi)

13 Mutation Testing con PITest

13.1 Introduzione al Mutation Testing

Il mutation testing è una tecnica per valutare la qualità dei test, generando versioni modificate (mutanti) del codice sorgente che introducono piccoli difetti (mutazioni), come:

- Modifica operatori matematici (+ diventa -, * diventa /)
- Negazione di condizioni (> diventa <=, == diventa !=)
- Modifica valori di ritorno (return 0 diventa return 1, return true diventa return false)
- Rimozione chiamate a metodi void

Un mutante si dice:

- **KILLED**: se almeno un test fallisce quando viene eseguito sul mutante
- **SURVIVED**: se tutti i test passano anche con la mutazione (indica debolezza nei test)
- **NO_COVERAGE**: se il mutante non è coperto da alcun test

Le metriche principali sono:

- **Mutation Coverage**: percentuale di mutanti killed sul totale mutanti
- **Test Strength**: percentuale di mutanti killed sui soli mutanti con coverage

13.2 Risultati Complessivi

Il **Test Strength del 95%** indica che i test sono molto efficaci nel rilevare difetti quando hanno coverage sul codice. Solo 1 mutante è sopravvissuto su 19 coperti

Tabella 10: PITest: Summary generale

Metrica	Valore
Classi mutate	2
Test eseguiti	63 test classes
Tempo esecuzione	1 secondo
Mutanti generati	23
Mutanti killed	18
Mutanti survived	1
Mutanti no coverage	4
Line Coverage	**86%** (19/22 linee)
Mutation Coverage	**78%** (18/23)
Test Strength	**95%** (18/19)

13.3 Analisi per RecyclableArrayList

13.3.1 Metriche

Tabella 11: PITest: RecyclableArrayList

Metrica	Valore
Line Coverage	92% (11/12 linee)
Mutation Coverage	83% (5/6 mutanti)
Test Strength	100% (5/5 mutanti coperti)
Mutanti generati	6
Mutanti killed	5
Mutanti survived	0
Mutanti no coverage	1

Test Strength 100% significa che ogni mutante coperto è stato ucciso

13.4 Analisi per MathUtils

13.4.1 Metriche

Test Strength 93% con solo 1 mutante sopravvissuto su 14 coperti

13.5 Interpretazione Mutatori

I mutatori PITest più attivi sono stati:

- **MathMutator** (16 mutanti): sostituisce operatori matematici (+, -, *, /, %)
- **PrimitiveReturnsMutator** (8 mutanti): modifica valori di ritorno primitivi (0→1, 1→0, etc.)

Tabella 12: PITest: MathUtils

Metrica	**Valore**
Line Coverage	80% (8/10 linee)
Mutation Coverage	76% (13/17 mutanti)
Test Strength	93% (13/14 mutanti coperti)
Mutanti generati	17
Mutanti killed	13
Mutanti survived	1
Mutanti no coverage	3

- **VoidMethodCallMutator** (2 mutanti): rimuove chiamate a metodi void (clear(), recycle())
- **ConditionalsBoundaryMutator** (1 mutante): modifica boundary di condizioni (¡ diventa ¡=)
- **NegateConditionalsMutator** (2 mutanti): inverte condizioni booleane
- **BooleanReturnValsMutator** (2 mutanti): inverte return true/false

Parte II
Syncope

Repository: https://github.com/Flaasks/syncope

14 Classe: MailTemplateLogic

La classe `MailTemplateLogic` è responsabile della gestione dei template email nel sistema, con operazioni CRUD e gestione dei formati HTML e TEXT

La classe `MailTemplateLogic` espone i seguenti metodi pubblici:

- `create(String key)`: crea un nuovo template
- `read(String key)`: legge un template esistente
- `delete(String key)`: elimina un template
- `list()`: elenca tutti i template
- `getFormat(String key, MailTemplateFormat format)`: recupera il contenuto di un formato specifico
- `setFormat(String key, MailTemplateFormat format, String template)`: imposta il contenuto di un formato

La classe interagisce con il database tramite `MailTemplateDAO` e verifica dipendenze tramite `NotificationDAO`

15 Category Partition

15.1 Metodologia

È stato adottato un approccio black-box basato sulle specifiche della classe. Le categorie sono state identificate analizzando la documentazione e il comportamento atteso dei metodi

15.2 Categorie Identificate

15.2.1 Categoria 1: Stato della Chiave

Questa categoria rappresenta lo stato di esistenza di una chiave nel database

Partizioni:

- P1 (Boundary): Chiave non esistente nel DB
- P2 (Boundary): Chiave esistente nel DB

Valori di test ai bordi:

- Chiave completamente nuova (non esistente)
- Chiave già presente nel database

15.2.2 Categoria 2: Formato Template (MailTemplateFormat)

Rappresenta il tipo di formato del template email

Partizioni:

- P1 (Boundary): HTML
- P2 (Boundary): TEXT

Valori di test ai bordi:

- MailTemplateFormat.HTML
- MailTemplateFormat.TEXT

15.2.3 Categoria 3: Contenuto Template

Rappresenta lo stato del contenuto memorizzato nel template

Partizioni:

- P1 (Boundary): Template vuoto (blank o null)
- P2 (Boundary): Template con contenuto valido

Valori di test ai bordi:

- Stringa vuota o null per il contenuto
- Stringa con contenuto HTML/TEXT valido

15.2.4 Categoria 4: Tipo di Operazione

Rappresenta le diverse operazioni disponibili sulla classe

Partizioni:

- P1: create(key)
- P2: read(key)
- P3: delete(key)
- P4: getFormat(key, format)
- P5: setFormat(key, format, template)
- P6: list()

I test sono stati progettati combinando le partizioni per coprire i casi più significativi, con particolare attenzione ai valori di boundary. La scelta delle combinazioni si è concentrata su:

- Transizioni di stato (non esistente $\rightarrow$ esistente)
- Condizioni di errore (chiave non trovata, duplicata)
- Varianti di formato (HTML vs TEXT)
- Contenuto vuoto vs contenuto valido

16 Test Category Partition Implementati

Ho implementato 16 test seguendo l'approccio Category Partition, utilizzando JUnit 5 e Mockito per l'isolamento delle dipendenze

16.1 Test CREATE

CP1 - create() con chiave già esistente: Testa la partizione P2 della Categoria 1. Verifica che venga lanciata `DuplicateException` quando si tenta di creare un template con chiave già presente

CP2 - create() con chiave nuova: Testa la partizione P1 della Categoria 1. Verifica il successo della creazione di un nuovo template con chiave non esistente

16.2 Test READ

CP3 - read() con chiave non esistente: Testa la partizione P1 della Categoria 1. Verifica che venga lanciata `NotFoundException` per chiave inesistente

CP4 - read() con chiave esistente: Testa la partizione P2 della Categoria 1. Verifica il successo della lettura di un template esistente

16.3 Test DELETE

CP5 - delete() con chiave esistente e nessuna notifica: Testa le partizioni P2 (Categoria 1) e P1 (Categoria 4). Verifica il successo dell'eliminazione quando il template non è referenziato

CP6 - delete() con chiave non esistente: Testa la partizione P1 della Categoria 1. Verifica che venga lanciata `NotFoundException`

16.4 Test GETFORMAT

CP7 - getFormat() HTML con template HTML vuoto: Testa le combinazioni P1 (Categoria 2) e P1 (Categoria 3). Verifica che venga lanciata `NotFoundException` quando il formato richiesto è vuoto

CP8 - getFormat() TEXT con template TEXT vuoto: Testa le combinazioni P2 (Categoria 2) e P1 (Categoria 3). Analogo a CP7 per formato TEXT

CP9 - getFormat() HTML con contenuto valido: Testa le combinazioni P1 (Categoria 2) e P2 (Categoria 3). Verifica il recupero corretto del contenuto HTML

CP10 - getFormat() TEXT con contenuto valido: Testa le combinazioni P2 (Categoria 2) e P2 (Categoria 3). Verifica il recupero corretto del contenuto TEXT

CP11 - getFormat() con chiave non esistente: Testa la partizione P1 della Categoria 1. Verifica l'eccezione per chiave inesistente

16.5 Test SETFORMAT

CP12 - setFormat() HTML: Testa la partizione P1 della Categoria 2. Verifica l'aggiornamento corretto del formato HTML

CP13 - setFormat() TEXT: Testa la partizione P2 della Categoria 2. Verifica l'aggiornamento corretto del formato TEXT

CP14 - setFormat() con chiave non esistente: Testa la partizione P1 della Categoria 1. Verifica l'eccezione per chiave inesistente

16.6 Test LIST

CP15 - list() con database vuoto: Testa il boundary inferiore (0 elementi). Verifica che venga restituita lista vuota

CP16 - list() con template presenti: Testa il caso con 1 elemento. Verifica la corretta restituzione della lista di template

17 Test Generati da LLM

17.1 Processo di Generazione

Sono stati generati 4 test utilizzando Google Gemini, fornendo come prompt la documentazione della classe e l'approccio Category Partition

"[LINK DOCUMENTAZIONE] Considerando la documentazione di questa classe, ho bisogno della definizione di N test in JUnit5 secondo metodologia category partition. è importante definire solo la classe minima di test più significativi, quindi andando a prendere come input dei test solamente i valori ai bordi delle category partition definite"

Rispetto a Bookkeeper con Chat-GPT, in questo caso l'IA è stata in grado di creare una category partition valida, per quanto povera

17.2 Test Generati

LLM1 - create() con nuova chiave: Test di creazione di un template con chiave non esistente

LLM2 - create() con chiave esistente: Test di creazione con chiave duplicata. Verifica il lancio di `DuplicateException`

LLM3 - read() con chiave mancante: Test di lettura con chiave inesistente. Verifica il lancio di `NotFoundException`

LLM4 - delete() con chiave mancante: Test di eliminazione con chiave inesistente. Verifica il lancio di `NotFoundException`

17.3 Discussione sui Test LLM

17.3.1 Punti di Forza

1. **Copertura dei casi base:** L'LLM ha identificato correttamente i casi fondamentali (chiave esistente/non esistente, success/error path)

2. **Test generati corretti:** I test generati hanno funzionato al primo tentativo, nessun errore di compilazione

17.3.2 Problematiche Rilevate

1. **Mancanza di verifiche dettagliate:** I test generati verificavano solo il tipo di eccezione, senza controllare il messaggio o lo stato finale del sistema
2. **Copertura limitata:** L'LLM si è concentrato sui casi più semplici, tralasciando scenari complessi come i formati multipli o le dipendenze tra entità

18 Analisi Coverage Pre-Control Flow

18.1 Esecuzione dei Test

Dopo l'implementazione dei 16 test Category Partition e dei 4 test LLM, è stata effettuata l'analisi della coverage utilizzando JaCoCo

18.2 Risultati Complessivi

La coverage ottenuta sulla classe `MailTemplateLogic` è stata:

- **Instructions Coverage:** 71% (187 su 262 istruzioni)
- **Branch Coverage:** 40% (9 su 22 branch)
- **Methods Coverage:** 93% (13 su 14 metodi)
- **Lines Coverage:** 78% (45 su 58 linee)

18.3 Analisi per Metodo

Metodi con Coverage 100%:

- `create()`: 100% istruzioni, 100% branch
- `read()`: 100% istruzioni
- `list()`: 100% istruzioni
- `getFormat()`: 100% istruzioni, 100% branch
- `setFormat()`: 100% istruzioni, 100% branch

Metodi con Coverage Parziale:

- `delete()`: 65% istruzioni, 50% branch

Metodi non Coperti:

- `resolveReference()`: 0% istruzioni, 0% branch

18.4 Identificazione dei Branch Mancanti

L'analisi dettagliata del metodo `delete()` ha evidenziato che il branch non coperto corrisponde alla condizione:

```
if (!notifications.isEmpty()) {
    SyncopeClientException sce =
        SyncopeClientException.build(ClientExceptionType.InUse);
    sce.getElements().addAll(...);
    throw sce;
}
```

Questo branch gestisce il caso in cui un template è referenziato da una o più notifiche, impedendone l'eliminazione. I test Category Partition iniziali avevano testato solo il caso con zero notifiche

18.5 Considerazioni

Il metodo `resolveReference()` non è stato coperto perché si tratta di un metodo di supporto interno utilizzato dal framework per la risoluzione delle dipendenze. Non è direttamente invocato dai test e la sua copertura richiederebbe test di integrazione più complessi, che per motivi di tempo non sono riuscito ad ideare ed implementare

19 Test Control Flow

19.1 Motivazione

Basandosi sui risultati dell'analisi JaCoCo, ho deciso di implementare test specifici per coprire il branch mancante nel metodo `delete()`, utilizzando un approccio Control Flow guidato dalla coverage

19.2 Definizione della Categoria

È stata definita una nuova categoria specifica per i test Control Flow:
Categoria: Numero di Notifiche Associate (In-Use State)
Partizioni con Boundary Values:

- P1 (Boundary): 0 notifiche - già coperto dai test CP
- P2 (Boundary): 1 notifica - valore minimo per generare InUseException

19.3 Test Implementati

19.3.1 CF1 - delete() con 1 notifica associata

Obiettivo: Testare il boundary minimo (1 notifica) che genera l'eccezione InUse
Configurazione:

- Template esistente nel database

- 1 notifica che referenzia il template
- Tentativo di eliminazione del template

Risultato Atteso:

- `SyncopeClientException` con tipo `ClientExceptionType.InUse`
- Lista elementi eccezione contiene esattamente 1 chiave di notifica
- Metodo `deleteById()` non viene invocato

20 Analisi Coverage Post-Control Flow

20.1 Esecuzione dei Test

Dopo l'aggiunta del test Control Flow ho rieseguito l'analisi di coverage con Jacoco

20.2 Risultati Complessivi

La coverage finale sulla classe `MailTemplateLogic` è:

- **Instructions Coverage:** 76% (201 su 262 istruzioni)
- **Branch Coverage:** 45% (10 su 22 branch)
- **Methods Coverage:** 93% (13 su 14 metodi)
- **Lines Coverage:** 78% (45 su 58 linee)

20.3 Miglioramento del Metodo delete()

Il metodo `delete()` ha raggiunto la copertura completa:

- **Prima:** 65% istruzioni, 50% branch
- **Dopo:** 100% istruzioni, 100% branch
- **Miglioramento:** +35% istruzioni, +50% branch

20.4 Confronto Pre/Post Control Flow

Incremento Totale:

- Instructions: da 71% a 76% (+5 punti percentuali)
- Branch: da 40% a 45% (+5 punti percentuali)
- 14 istruzioni aggiuntive coperte
- 1 branch aggiuntivo coperto

Efficienza dei Test: Con un solo test mirat, circa il 5% del totale, ho ottenuto un miglioramento significativo della coverage

21 Classe: DefaultPasswordGenerator

Ho definito la category partition identificando un buon numero di categorie principali basandomi sull'analisi della documentazione della classe

Le costanti chiave della classe sono:

- VERY_MIN_LENGTH = 0
- VERY_MAX_LENGTH = 64
- MIN_LENGTH_IF_ZERO = 8

21.1 Categorie Identificate

Ho suddiviso le categorie in quattro gruppi logici principali:

Gruppo 1: Lunghezze Password

Categoria 1 - Numero di policy: 0 policy, 1 policy. Ho testato il comportamento con lista vuota e con una singola policy per verificare il caso base e il caso minimo significativo

Categoria 2 - minLength: 0, 64. Il valore 0 viene convertito a 8 tramite MIN_LENGTH_IF_ZERO, mentre 64 rappresenta la lunghezza massima possibile

Categoria 3 - maxLength: 0, 64. Il valore 0 viene ignorato e sostituito con VERY_MAX_LENGTH (64), mentre 64 e il limite massimo

Gruppo 2: Requisiti Caratteri

Categoria 4 - uppercase: 0, 1. Zero indica nessun requisito, 1 e il boundary minimo per richiedere almeno un carattere maiuscolo

Categoria 5 - lowercase: 0, 1. Stesso ragionamento di uppercase

Categoria 6 - digit: 0, 1. Zero per nessun requisito di cifre, 1 per richiederne almeno una

Categoria 7 - special: 0, 1. Zero per nessun carattere speciale richiesto, 1 per richiederne almeno uno

Categoria 8 - alphabetical: 0, 1. Caratteri alfabetici generici

Categoria 9 - repeatSame: 0, 1. Zero per nessun limite di ripetizione, 1 e il limite piu restrittivo possibile ma causa IllegalArgumentException perche richiede sequenze di lunghezza almeno 2

Gruppo 3: Liste di Caratteri

Categoria 10 - specialChars: vuota, un elemento. Lista vuota usa i caratteri speciali di default, un elemento definisce caratteri custom

Categoria 11 - illegalChars: vuota, un elemento. Lista vuota non esclude caratteri, un elemento esclude specifici caratteri

Categoria 12 - wordsNotPermitted: vuota, un elemento. Lista vuota non esclude parole, un elemento esclude parole specifiche

Gruppo 4: Configurazioni Booleane e Null Cases

Categoria 13 - usernameAllowed: false, true. Flag booleano per permettere o meno username nella password

Categoria 14 - policies parameter: null. Test del comportamento con parametro null

Categoria 15 - resource parameter: null. Test del comportamento con risorsa null

Categoria 16 - realms parameter: null. Test del comportamento con realms null

Categoria 17 - policy in list: null. Policy null all'interno di una lista di policy

Categoria 18 - resource passwordPolicy: null. Policy null nella risorsa

Categoria 19 - realm passwordPolicy: null. Policy null nel realm

Gruppo 5: Casi Invalidi

Categoria 20 - minLength maggiore di maxLength: minLength=16, maxLength=8. Testa la correzione automatica

Categoria 21 - Requisiti che eccedono minLength: uppercase=5, lowercase=5, digit=5, minLength=8. Verifica che la lunghezza venga adattata

Categoria 22 - Configurazione vuota: tutti i valori a default. Testa il fallback al charset di default

21.2 Implementazione dei Test

Ho implementato 33 test Category Partition. Ogni test è stato progettato per essere atomico

Qui ho avuto diversi problemi lato implementazione, quindi per riuscire ad accedere ai metodi protected merge() e generate(DefaultPasswordRuleConf) della classe sotto test, ho creato (aiutandomi con AI) una classe interna TestableDefaultPasswordGenerator che estende DefaultPasswordGenerator ed espone questi metodi attraverso metodi wrapper pubblici mergeForTest() e generateForTest()

22 Test Generati da LLM

Ho utilizzato Google Gemini per generare 4 test aggiuntivi con l'obiettivo di verificare se un LLM potesse identificare scenari non coperti dalla mia Category Partition. Ho fornito a Gemini la documentazione della classe DefaultPasswordGenerator e chiesto di generare test significativi:

Test LLM1 - Valid Range: testa una configurazione con minLength=8, maxLength=16 e vari requisiti di caratteri

Test LLM2 - Fixed Length: testa una configurazione con minLength=maxLength=10

Test LLM3 - Impossible Constraints: il test originale assumeva che requisiti impossibili lanciassero un'eccezione, ma in realta Passay genera semplicemente una password piu lunga

Test LLM4 - Illegal Characters: testava l'esclusione di caratteri illegali verificando che non fossero presenti nella password

22.1 Considerazioni Personali

I test generati dall'LLM sono stati interessanti ma non particolarmente sofisticati. Gemini ha proposto principalmente variazioni sui test gia coperti dalla Category Partition, senza identificare casi mancanti. Questo suggerisce che un approccio sistematico come la Category Partition con boundary values è, per il momento, superiore alla generazione casuale o pseudo-intelligente di test

23 Analisi Jacoco Pre-Control Flow

Dopo aver eseguito i category partition test e gli LLM, ho generato il report Jacoco per analizzare la coverage e identificare branch non coperti

La coverage complessiva prima dei test Control Flow era:

- Line coverage: 61/72 linee coperte (85 percento)
- Branch coverage: molti branch parzialmente coperti

23.1 Branch Non Coperti Identificati

Analizzando il report HTML ho identificato i seguenti branch non coperti:

Branch 1 - Linea 68: filter(p -> !policies.contains(p)). Questo branch verifica se una policy è gia presente nella lista quando si processano i realms. Il branch non era coperto perche nessun test aveva creato uno scenario con policy duplicate tra resource e realms

Branch 2 - Linea 117: if (ruleConf.getAlphabetical() > result.getAlphabetical()). Questo branch del metodo merge() viene eseguito quando si fa merge di configurazioni con requisiti di caratteri alfabetici

Altri branch parzialmente coperti erano alle linee 139 e 144, relativi al filtraggio di caratteri speciali e illegali duplicati durante il merge, ma richiedevano scenari piu complessi che non sono riuscito ad analizzare

24 Test Control Flow

Ho quindi definito 2 test Control Flow mirati a coprire i branch identificati, utilizzando sempre la Category Partition ai boundary values

24.1 CF1: Multiple Realms with Duplicate Policy

Obiettivo: coprire il branch alla linea 68 che filtra policy duplicate

Category Partition:

- realms: 2 realms
- policy: stessa policy per entrambi i realms
- resource: policy diversa

Ho creato uno scenario con ExternalResource avente una propria PasswordPolicy, e due Realm che puntano entrambi alla stessa PasswordPolicy. Questo forza l'esecuzione del filtro alla linea 68 perche la seconda policy del realm2 e gia presente nella lista

Ho utilizzato Mockito per creare mock di PasswordPolicy, ExternalResource e Realm, configurando le relazioni con when().thenReturn(). Il test verifica semplicemente che venga generata una password senza errori

24.2 CF2: Alphabetical Character Requirement

Obiettivo: coprire il branch alla linea 117 nel metodo merge()
Category Partition:

- alphabetical: 1
- minLength: 8
- maxLength: 64

Ho creato una DefaultPasswordRuleConf con alphabetical=1, che forza il branch if (ruleConf.getAlphabetical() > result.getAlphabetical()) perche result parte da 0. Il test verifica che il merge preservi il valore alphabetical=1 e che la password generata rispetti la lunghezza minima

25 Analisi Jacoco Post-Control Flow

Dopo aver aggiunto i 2 test Control Flow ho rieseguito tutti i test e rigenerato il report Jacoc

25.1 Risultati Coverage Finale

Branch linea 117 (alphabetical): precedentemente non coperto, ora coperto al 100 percento grazie al test CF2. Il report JaCoCo mostra "All 2 branches covered" per questa linea

Branch linea 68 (filter duplicate): nonostante il test CF1, questo branch risulta ancora "All 2 branches missed" nel report. Questo e dovuto al fatto che TestableDefaultPasswordGenerator sovrascrive getPasswordRules() restituendo sempre una lista vuota, quindi il flusso del codice non processa realmente le policy ma genera direttamente la password con configurazione di default

26 Mutation Testing con PiTest

Ho eseguito PiTest per valutare la qualita dei test attraverso mutation testing
Statistiche Generali:

- Mutazioni generate: 86
- Mutazioni killed: 37 (43 percento)
- Mutazioni survived: 44
- Mutazioni senza coverage: 5
- Test strength: 46 percento
- Test eseguiti: 1142 (13.28 test per mutazione)
- Line coverage: 61/72 (85 percento)

27 Utilizzo dei Mock nei Test

Ho utilizzato Mockito estensivamente per isolare la classe DefaultPasswordGenerator dalle sue dipendenze esterne

Ho creato mock di tutte le entity del persistence layer:

- PasswordPolicy: rappresenta una policy di password
- ExternalResource: rappresenta una risorsa esterna con policy
- Realm: rappresenta un realm con policy
- Implementation: rappresenta un'implementazione di regola

Questi mock sono stati annotati con @Mock e iniettati automaticamente da MockitoExtension. Li ho utilizzati principalmente nei test che chiamano generate(ExternalResource, List<Realm>) per fornire oggetti senza dover creare entità reali del database

27.1 Stubbing dei Comportamenti

Ho utilizzato when().thenReturn() per definire il comportamento dei mock, ad esempio:

when(resource.getPasswordPolicy()).thenReturn(resourcePolicy) configura il mock resource per restituire resourcePolicy quando viene chiamato getPasswordPolicy()

when(realm1.getPasswordPolicy()).thenReturn(sharedPolicy) configura realm1 per restituire una policy specifica

Questo mi ha permesso di definire scenari controllati come policy duplicate, policy null, e le combinazioni di policy tra resource e realms

27.2 Verifica delle Interazioni

Non ho utilizzato verify() di Mockito perche i test si concentravano sul risultato finale (la password generata) piuttosto che sulle interazioni interne. In un approccio piu rigoroso potrei verificare quante volte getPasswordPolicy() viene chiamato, se viene chiamato nell'ordine corretto, etc