Analisi della Buggyness nei Metodi Software: Un Approccio Data-Driven al Refactoring

Gaia Meola
Studentessa del Corso di Laurea Magistrale in Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Matricola 0369454, A.A. 2024/2025

1 ABSTRACT

Il legame tra la struttura del codice e l'insorgenza di bug rappresenta una sfida critica per la sostenibilità del software. Questo lavoro indaga tale correlazione attraverso l'analisi dei repository *Apache BookKeeper* e *Apache OpenJPA*, integrando metriche statico-dinamiche e algoritmi di machine learning per modellare la *defect proneness* a livello di metodo. A differenza degli approcci tradizionali, la ricerca si focalizza sull'efficacia del refactoring come strumento preventivo: mediante una simulazione *what-if*, è stato quantificato come la neutralizzazione sistematica dei *code smell* possa alterare drasticamente il profilo di rischio del software. I risultati evidenziano un abbattimento del fattore di rischio del 25,59% per *BookKeeper* e del 16,40% per *OpenJPA*. Tali evidenze confermano che una strategia di manutenzione *data-driven*, agendo sulla granularità del singolo metodo, non solo riduce il debito tecnico, ma costituisce una barriera concreta contro la propagazione dei difetti, ottimizzando l'affidabilità del sistema ben prima della fase di rilascio.

2 Introduzione

La manutenibilità rappresenta oggi una sfida critica nello sviluppo di sistemi complessi soggetti a evoluzione continua. L'incessante apporto di modifiche, volto a integrare nuove funzionalità o correggere errori, tende a degradare la struttura interna del software, accumulando **debito tecnico** e incrementando la propensione ai difetti. In questo scenario, la capacità di prevedere la *buggyness* a livello di singolo metodo emerge come un fattore determinante per ottimizzare le risorse di testing. Nonostante l'interesse scientifico per le metriche di qualità, sussiste una carenza di evidenze empiriche solide circa l'impatto reale della riduzione dei *code smells* sulla difettosità a granularità metodologica. Gran parte della letteratura si focalizza su classi o moduli, una scala troppo grossolana per guidare interventi mirati. Questa lacuna limita la possibilità per i team di sviluppo di definire strategie preventive basate sui dati e di quantificare il reale beneficio derivante dalla bonifica del codice. Il presente studio si propone di colmare questo divario attraverso un'analisi empirica condotta sui progetti **Apache BookKeeper** e **Apache OpenJPA**, gestiti attraverso due dataset paralleli. L'obiettivo principale è investigare l'efficacia del Machine Learning nella predizione dei guasti e valutare se interventi strutturali possano effettivamente prevenire l'insorgenza di bug. Nello specifico, il report intende rispondere ai seguenti quesiti di ricerca:

- **RQ1:** Quale tra i classificatori analizzati dimostra la migliore capacità predittiva nell'identificare correttamente i metodi affetti da bug nei due contesti analizzati?
- **RQ2:** Quanti bug potenziali avrebbero potuto essere evitati attraverso l'applicazione tempestiva di attività di *refactoring* volte alla rimozione delle anomalie strutturali?

Per fornire una visione completa del processo, il resto del documento è organizzato come segue: la **Sezione 3** approfondisce il protocollo sperimentale, descrivendo l'integrazione JIRA-Git e l'algoritmo *Proportion-based SZZ* per l'estrazione delle etichette. La **Sezione 4** costituisce il nucleo sperimentale del lavoro: partendo dalla selezione motivata di metodi critici, descrive l'attività di *refactoring* manuale e culmina nell'analisi quantitativa *What-if* (ispirata a Falessi et al., 2017) per stimare l'impatto della rimozione degli *smell* sulla riduzione della difettosità. Infine, la **Sezione 5** discute criticamente i risultati e le potenziali minacce alla validità, seguita dalle conclusioni e dagli sviluppi futuri nella **Sezione 6**. Il codice e le analisi sono disponibili pubblicamente sui repository di progetto[1,2].

3 Misura e Metodologia

3.1 Dataset ed Estrazione delle Etichette

La presente sezione descrive il protocollo sperimentale adottato per la costruzione di un dataset affidabile finalizzato alla *Software Defect Prediction*. L'intero workflow, automatizzato tramite un analizzatore custom sviluppato in Java, si articola in tre fasi logiche fondamentali: l'estrazione e sincronizzazione dai repository, la datazione dei bug tramite algoritmo SZZ e l'estrazione delle feature. L'analisi empirica si basa sull'integrazione dei dati provenienti dal repository *Git* e dall'issue tracker *JIRA* per i progetti **Apache BookKeeper** e **Apache OpenJPA**. Tramite la classe `GetReleaseInfo`, il sistema interroga le API REST di JIRA per ricostruire la timeline del progetto, considerando solo le release con stato *"released"*, *releaseDate* valida e naming convention semantica (`X.Y.Z`). Questa normalizzazione garantisce la coerenza tra i tag di Git e le versioni documentate in JIRA. Seguendo un approccio metodologico rigoroso, è stata operata una distinzione tra l'orizzonte di osservazione e le istanze di training: il calcolo delle metriche storiche e il

[1] `https://github.com/GaiaMeola/Progetto_ISW2`
[2] `https://sonarcloud.io/project/overview?id=GaiaMeola_Progetto_ISW2`

processo di *bug labelling* hanno interessato la totalità delle release estratte per permettere all'SZZ di identificare *fixing commits* cronologicamente distanti dall'iniezione. Tuttavia, come **basic stats** di filtraggio, il dataset finale include esclusivamente le istanze appartenenti al primo terzo (33%) delle release totali. Questa scelta è fondamentale per contrastare il fenomeno dello *snoring*, garantendo che ogni riga del dataset disponga di una storia futura (il restante 67%) sufficientemente ampia affinché i difetti abbiano il tempo di emergere ed essere risolti. Il sistema esegue una ricerca mirata tramite JQL per isolare i difetti reali (`issuetype = Bug`, `status` in `{Resolved, Closed}`, `resolution = Fixed`). Per ogni ticket, la classe `TicketParser` estrae la *Opening Version* (OV), la *Fix Version* (FV) e le *Affected Versions* (AV). Il collegamento tra JIRA e Git è affidato alla classe `BugLinker` tramite una strategia di *double-pass linking*: in prima istanza si cercano riferimenti espliciti agli ID ticket nei messaggi di commit; in seconda istanza si associano i commit mancanti basandosi su criteri temporali, di località (modifica di file già marcati come *fixed*) e di identità dell'autore. La determinazione del periodo di latenza $[IV, FV)$ è gestita da una versione estesa dell'algoritmo **SZZ**. Se la *Injected Version* (IV) è assente o incoerente, la classe `ProportionEstimator` ne stima l'indice tramite il modello *Proportion*:

$$Index(IV) = \max(0, \text{round}(Index(FV) - (Index(FV) - Index(OV)) \times P)) \quad (1)$$

Il fattore P deriva da una gerarchia di affidabilità: viene calcolato incrementalmente sul progetto corrente dopo 10 ticket validi o, in fase di *Cold Start*, tramite un'analisi *cross-project* su quattro progetti Apache affini (*Avro, ZooKeeper, Syncope, Tajo*). L'ultimo stadio del processo è gestito dalla classe `BugLabeler`: superando la classica etichettatura a livello di classe, il sistema utilizza la classe `MethodTouchAnalyzer` e la libreria *JGit* per identificare le specifiche righe di codice modificate nei *fixing commits*. L'etichetta *buggy* viene così propagata esclusivamente alle versioni del metodo esistenti nell'intervallo di latenza calcolato, riducendo drasticamente il rumore statistico del dataset finale.

3.2 Metriche e Definizioni

Al fine di analizzare la relazione tra la qualità del codice e la sua propensione ai difetti, sono state misurate diverse categorie di metriche a granularità di metodo, trattate come *features* per i modelli di classificazione e distinte in base alla loro natura operativa. Nello specifico, per isolare i fattori su cui è possibile intervenire attivamente rispetto a quelli legati alla storia del componente, le feature sono state distinte in:

- **Actionable Features** – Descrivono la struttura interna e il design del metodo, rappresentando i target primari per il refactoring: *NSmells* (violazioni identificate tramite PMD con i rule-set `design.xml` e `bestpractices.xml`), *Complessità Ciclomatica e Cognitiva*, *Dimensioni e Densità* (LOC, Statement e Local Variable Count) e parametri di *Interfaccia* (Parameter Count, Nesting Depth, Return Type Complexity).
- **Non-actionable Features** – Estratte tramite *JGit*, descrivono la storia evolutiva e non sono alterabili retroattivamente: *Churn e Modifiche di Riga* (volume di *StmtAdded* e *StmtDeleted*) e indicatori di *Instabilità e Collaborazione* (*Method Histories* e numero di *Distinct Authors*).

Per validare la rilevanza delle feature, è stata calcolata la **correlazione di Spearman (ρ)** tra ogni feature e la variabile target (*buggyness*). Tale coefficiente permette di identificare relazioni non lineari, fornendo una misura della consistenza con cui il valore di una metrica cresce al variare della probabilità di presenza di un difetto. I risultati statistici dettagliati, inclusi i coefficienti ρ e i relativi p-value per ogni feature analizzata, sono consultabili in Appendice: la **Tabella 1** riporta i dati relativi al progetto *BookKeeper*, mentre la **Tabella 2** descrive i risultati per *OpenJPA*.

3.3 Protocollo Sperimentale

Il protocollo sperimentale definisce le procedure di addestramento e validazione per la selezione del *Best Classifier* tra **Random Forest**, **Naive Bayes** e **IBk**.

Per garantire robustezza statistica e minimizzare il *bias*, è stata adottata una strategia di tipo **10-times 10-fold Cross-Validation**. Sebbene l'approccio *Walk-Forward* (o *Sliding Window*) sia considerato il **gold standard** quando si opera su serie temporali software per gestire il *Concept Drift*, ovvero il cambiamento delle proprietà statistiche della variabile target nel tempo, in questo studio si è optato per una **Stratified Cross-Validation** poiché, limitando l'analisi al primo 33% delle release, una suddivisione cronologica avrebbe frammentato eccessivamente il dataset, impedendo l'apprendimento di pattern significativi. La ripetizione per dieci volte permette di esplorare l'intero spazio delle istanze, fornendo una stima affidabile della varianza e una baseline robusta per il confronto. L'analisi preliminare condotta tramite l'ambiente **Weka Explorer** ha rivelato una distribuzione delle classi estremamente coerente tra i due progetti, con una quota di istanze *Buggy* attestata al **20,8%** per BookKeeper e al **21,0%** per OpenJPA. Nel caso di BookKeeper, date le dimensioni contenute del dataset, si è scelto di operare sulle istanze originali senza applicare tecniche di bilanciamento artificiale, al fine di definire una baseline di performance che rifletta le reali condizioni operative del software. Al contrario, per gestire l'elevata mole di dati di **OpenJPA** e garantire la sostenibilità computazionale delle procedure di addestramento, è stato applicato un **campionamento casuale stratificato**. Tale tecnica ha permesso di estrarre un campione rappresentativo di **40.000 istanze** che, pur riducendo la complessità del calcolo, preserva rigorosamente la distribuzione originaria dei difetti osservata in fase di esplorazione. Per garantire la riproducibilità totale dei risultati, il workflow sperimentale adotta un **random-seed schedule** deterministico. È stato fissato un seme globale ($S = 42$), che viene incrementato linearmente ad ogni ripetizione della procedura ($S_i = S + i$). Tale seme viene utilizzato per inizializzare l'operazione di *shuffling* tramite il metodo `randomize()`, assicurando che il partizionamento dei dati in *fold* e le componenti stocastiche dei classificatori siano identici e verificabili tra diverse esecuzioni. Per ottimizzare l'efficienza dei modelli senza perdita di informazioni rilevanti, è stata integrata una fase di **Feature Selection** di tipo *filter* basata su **Information Gain** (con soglia impostata a $0,01$). Tale scelta è stata dettata dalla necessità di escludere metriche il cui potere discriminante rispetto alla variabile target risultava trascurabile. In questa sede, non sono stati adottati approcci di tipo *wrapper* (quali *Forward Selection* o *Backward Elimination*) data la dimensionalità gestibile del set di feature iniziale. L'esplorazione di strategie di selezione basate sull'interazione dinamica tra sottoinsiemi di feature e specifici classificatori viene pertan-

to demandata a possibili sviluppi e lavori futuri. Per ogni classificatore selezionato è stata eseguita una fase di ottimizzazione dei parametri volta a massimizzare la stabilità predittiva. Come implementato nella `ClassifierFactory`, per l'esperimento su **BookKeeper** è stata adottata una configurazione standard con $K = 3$ per l'algoritmo IBk e un set di 100 iterazioni per Random Forest, sfruttando il parallelismo computazionale tramite *execution slots* multipli. Al contrario, per il dataset **OpenJPA**, la configurazione è stata raffinata per garantire la sostenibilità dell'addestramento su un sample di 40.000 istanze. In questo contesto, l'algoritmo **Random Forest** è stato configurato con parametri specifici (`-I 30 -depth 12 -M 50`), riducendo il *BagSizePercent* al 40% per prevenire l'*overfitting* e ottimizzare i tempi di esecuzione. La valutazione dei classificatori precedentemente introdotti non si è limitata alla semplice accuratezza globale, parametro spesso fuorviante in presenza di dataset sbilanciati, ma ha impiegato un set di metriche orientate all'utilità pratica nel contesto del *Software Engineering*. Nello specifico: la **Precision** indica la frazione di metodi segnalati come *buggy* che risultano effettivamente difettosi, cruciale per minimizzare i falsi positivi; la **Recall** misura la capacità di intercettare i difetti realmente presenti, metrica prioritaria per evitare che i bug raggiungano la produzione; l'**AUC-ROC** esprime la probabilità che il modello assegni un rischio superiore a un metodo *buggy* rispetto a uno *clean*, valutando la capacità discriminante indipendentemente dalla soglia; il **Kappa Score** misura la coerenza delle predizioni al netto del caso (valori $> 0,40$ indicano un apprendimento solido, mentre tra $0,20$ e $0,40$ segnalano pattern significativi); infine, la metrica **NPofB20** indica la frazione di bug totali identificati ispezionando esclusivamente il top 20% dei metodi più rischiosi, parametro fondamentale per la prioritizzazione dei test. La scelta del modello ottimale, denominato *BestClassifier*, è stata guidata da una gerarchia di priorità volta a massimizzare l'efficacia delle attività di manutenzione preventiva:

1. **Massimizzazione della Recall:** privilegiando l'identificazione del maggior numero possibile di difetti anche a fronte di un ragionevole compromesso sulla *Precision*, seguendo un approccio orientato alla sicurezza (*safety-oriented*).

2. **Capacità di ranking:** espressa dall'**AUC-ROC**, parametro fondamentale per garantire che il modello sia in grado di ordinare correttamente i metodi in base al loro effettivo livello di criticità.

3. **Efficienza operativa:** misurata tramite l'indice **NPofB20**. Tale metrica identifica il modello che permette di rilevare la quota maggiore di bug con il minor sforzo di ispezione manuale, ottimizzando così le risorse del team di sviluppo.

La determinazione del **BestClassifier** per i due casi studio è stata effettuata sulla base dei criteri di selezione precedentemente definiti, analizzando le prestazioni medie riportate nelle **Tabelle 3** e **4**. Per il progetto **BookKeeper**, i risultati evidenziano come l'algoritmo **IBk** ($k = 3$) garantisca il miglior compromesso operativo. Al contrario, lo scenario relativo a **OpenJPA** favorisce la natura *ensemble* di **Random Forest**; quest'ultimo, pur eguagliando la *Recall* del classificatore IBk ($\approx 0,88$), si dimostra superiore nella capacità di discriminazione statistica, raggiungendo un valore di **AUC** pari a $0,9534$.

3.4 Assunzioni e Scelte Metodologiche

La validità dello studio poggia su scelte metodologiche mirate a bilanciare rigore e fattibilità tecnica. È stata assunta l'indipendenza statistica tra le osservazioni *metodo–release*, mantenendo costanti le metriche statiche per i componenti invariati tra versioni consecutive. Per garantire solidità e confrontabilità ai risultati, la sperimentazione è stata focalizzata su tre classificatori d'elezione nella letteratura della *defect prediction*. Infine, data l'assenza di strumenti nativi per l'analisi a granularità metodologica, l'estrazione dei dati è stata operata tramite parser custom basati sulla libreria **JavaParser**, permettendo una mappatura puntuale dell'Abstract Syntax Tree (AST) dei progetti analizzati.

4 Risultati

L'analisi qui presentata si fonda sui dati estratti dai repository di **BookKeeper** e **OpenJPA**, processati tramite la pipeline di *defect prediction* descritta nella Sezione 3.

4.1 Selezione del Metodo Target

La selezione dei metodi target per il *refactoring* è stata guidata dall'analisi delle correlazioni di Spearman (ρ). Tale coefficiente (dettagli in Appendice A) ha permesso di identificare le feature con il maggiore impatto statistico sulla variabile *buggy*, orientando l'intervento verso le criticità strutturali più correlate alla difettosità. L'analisi statistica indica nello **StatementCount** la *Feature Azionabile* ($AFeature$) con la maggiore correlazione assoluta per entrambi i progetti ($\rho_{BK} = 0,0855$; $\rho_{OJP} = 0,1785$). Tuttavia, per **BookKeeper** si è scelto il **Number of Smells** come *AFeatue*, al fine di validare empiricamente se la bonifica qualitativa sia sufficiente a invertire il verdetto del classificatore, ricollegandosi alla generalizzazione della *What-if Analysis* (Sez. 4.5). Per **OpenJPA**, la scelta è ricaduta coerentemente sullo **StatementCount**. Su queste basi, sono stati isolati i **Metodi Target** ($AFMethod$) con i valori massimi di $AFeature$ nell'ultima release:

- **BookKeeper (v4.2.1):** `BenchReadThroughputLatency.main()` (8 smell).
- **OpenJPA (v2.0.0):** `FieldMetaData.copy()` (38 statement).

La selezione di tali candidati è motivata da criteri di criticità strutturale e operativa: in primo luogo, l'elevata densità di difetti li rende *hotspot* ideali per testare la capacità predittiva del modello; secondariamente, la loro centralità architetturale, legata alla persistenza dei metadati in OpenJPA e all'orchestrazione dei benchmark in BookKeeper, garantisce che l'intervento impatti componenti vitali per l'integrità e le performance del sistema.

4.2 Analisi Critica dei Metodi Target

In questa sezione vengono esaminati i profili tecnici dei metodi selezionati, definendo lo stato di *baseline* qualitativa pre-ottimizzazione.

Caso Studio 1: `BenchReadThroughputLatency` **(BookKeeper)**

- **Signature:** `public static void main(String[] args) throws Exception`
- **Ruolo Operativo:** Orchestrazione del benchmark prestazionale, inclusa l'inizializzazione dell'infrastruttura ZooKeeper e la gestione del ciclo di vita dei ledger.

- **Analisi delle Criticità:** Il metodo rappresenta un *hotspot* qualitativo critico con **8 Code Smell** attivi.

Caso Studio 2: `FieldMetaData` **(OpenJPA)**

- **Signature:** `public void copy(FieldMetaData field)`
- **Ruolo Operativo:** Gestione della logica di duplicazione dei metadati di persistenza tra istanze della classe.
- **Analisi delle Criticità:** Sebbene caratterizzato da un numero di righe contenuto (55 LOC), il metodo esibisce una densità logica anomala con uno **Statement Count di 38**.

4.3 Criteri di Identificazione e Strategia di Refactoring

L'attività di *refactoring* è stata guidata da un approccio ibrido che combina l'analisi sistematica delle metriche con le capacità computazionali del modello **Gemini 1.5 Flash**. Attraverso una tecnica di *Zero-Shot Prompting*, si è operata una scomposizione logica del codice mirata alla neutralizzazione delle *Actionable Features* (*Number of Smells* per BookKeeper, *Statement Count* per OpenJPA), garantendo al contempo che le variabili correlate alla *bugginess* non subissero peggioramenti. Il protocollo ha seguito una regola di ottimizzazione vincolata: abbattere la complessità cognitiva e il *Nesting Depth* tramite *Extract Method* e *Functional Decomposition*, evitando di introdurre eccessiva frammentazione o nuovi *smell* strutturali. Il processo, articolato in step iterativi di raggruppamento per dominio e bilanciamento del carico, ha permesso di ridurre effettivamente il debito tecnico invece di riallocarlo nei sottometodi. Tale efficacia è confermata dal **ricalcolo sistematico delle metriche post-intervento** su ogni unità estratta, che certifica il raggiungimento di profili di complessità nominali. La mappatura tra criticità, tecniche applicate e il dettaglio delle nuove misurazioni è sintetizzata in Appendice C, D e E.

4.4 Risultati e Validazione

L'efficacia degli interventi è stata validata utilizzando i classificatori selezionati come oracoli di riferimento, ricalcolando la probabilità di *bugginess* sulla base della nuova topologia del codice post-refactoring. Questo approccio ha permesso di isolare l'impatto delle scomposizioni logiche, verificando se il miglioramento strutturale fosse di per sé sufficiente a spostare il verdetto del modello da *buggy* a *clean*. L'analisi quantitativa evidenzia un miglioramento strutturale drastico: in **BookKeeper**, la scomposizione del `main` attraverso tecniche di *Extract Method*, *Parameterize Method* e *Static Delegation* ha portato a una contrazione dei *code smell* del **75,0%** (da 8 a 2), mentre in **OpenJPA** l'isolamento delle logiche di *Fetch Groups* ha ridotto lo *Statement Count* dell'**81,6%** (da 38 a 7). I risultati confermano due diverse declinazioni del successo dell'intervento: nel caso **OpenJPA** si è ottenuta una de-classificazione totale, con la probabilità di errore crollata sotto la soglia critica del 50%; nel caso **BookKeeper**, invece, il *refactoring* ha permesso di isolare con precisione le componenti realmente critiche. Se nella *baseline* il rischio era distribuito in modo opaco in un unico monolite, la nuova struttura identifica chiaramente gli *hotspot* residui, permettendo di **concentrare le risorse di testing** e revisione manuale esclusivamente sui moduli a probabilità elevata. In definitiva, l'esperimento dimostra che il *refactoring* riesce a mappare la complessità intrinseca laddove non sia eliminabile, trasformando la qualità del codice in un *driver* operativo per i processi decisionali del ciclo di vita del software. L'integrità funzionale di tale processo è stata rigorosamente garantita dal superamento dell'intera suite di unit testing originale e dal successo delle build Maven, confermando la piena equivalenza semantica delle trasformazioni effettuate (le versioni del codice *baseline* e post-ottimizzazione sono disponibili per il confronto nell'**Appendice I**).

4.5 Analisi What-if

Metodi Buggy Evitabili

L'efficacia degli interventi è stata validata utilizzando i classificatori selezionati come oracoli di riferimento, interrogati per ricalcolare la probabilità di *bugginess* sulla base della nuova topologia del codice post-refactoring. Questo approccio ha permesso di isolare l'impatto delle scomposizioni logiche, verificando se il miglioramento strutturale fosse di per sé sufficiente a spostare il verdetto del modello da *buggy* a *clean*. I risultati, dettagliati nell'Appendice F, confermano tale transizione: per **BookKeeper**, il rischio predetto del metodo *main* è crollato al 33,33%, localizzando la criticità residua in *hotspot* isolati (Tab. 11), mentre per **OpenJPA** si è ottenuta una de-classificazione totale di tutti i componenti estratti (Tab. 12). In definitiva, l'analisi ha dimostrato che la rimozione di una *feature actionable* tramite refactoring manuale non è un atto isolato, ma innesca una reazione a catena che riduce drasticamente i valori di tutte le metriche strutturali e di complessità ad essa correlate. Questa evidenza empirica, confermando la sensibilità dei modelli ai miglioramenti qualitativi reali, costituisce il presupposto metodologico per la successiva *What-if Analysis*. L'indagine si sposta dunque dal singolo caso studio all'intero sistema, con l'obiettivo di determinare se la riduzione della *bugginess* osservata possa essere generalizzata attraverso la neutralizzazione sistematica degli *smell*.

Generalizzazione dei Risultati

Quest'ultima fase della sperimentazione mira a quantificare il beneficio teorico derivante dalla completa risoluzione dei *Code Smells*. Attraverso una tecnica di simulazione, l'impatto del *refactoring* viene proiettato sulle predizioni dei modelli di Machine Learning, permettendo di stimare su larga scala il guadagno in termini di affidabilità del software. Per garantire la massima affidabilità, il processo è affidato ai classificatori che hanno registrato le migliori performance globali nelle fasi preliminari: l'algoritmo **IBk** per *BookKeeper* e il **Random Forest** per *OpenJPA*. Tale selezione assicura che la valutazione dell'impatto strutturale sia condotta dai modelli dotati della maggiore capacità di generalizzazione. Una volta selezionato, il modello viene applicato senza alcun ri-addestramento ai tre sottoinsiemi partizionati sulla metrica *Number of Smells*: il **Dataset** B^+ (metodi *smelly*), il **Dataset** C (controllo *clean*) e il **Dataset** B (scenario *What-if*). In quest'ultimo, la feature di degradazione strutturale è forzata a zero, garantendo che ogni variazione nella predizione sia imputabile esclusivamente alla manipolazione della variabile d'interesse. Il cuore dell'analisi risiede nel calcolo del differenziale (Delta) tra le predizioni dello scenario reale (E_{B^+}) e quello simulato (E_B), operando una comparazione tra due stime generate. Il confronto tra le predizioni permette, infatti, di rispondere a un quesito cruciale: *«Qual*

è la quota di metodi che il modello aveva classificato come rischiosi e che, a parità di tutte le altre metriche, vengono riconsiderati sicuri una volta rimosso l'indicatore dello smell?». Una riduzione significativa nel dataset B conferma dunque la presenza di *smell* come **feature discriminante** nel processo decisionale. L'analisi quantitativa dell'impatto (i cui criteri di calcolo e definizioni formali sono riportati nell'**Appendice H**) ha prodotto i seguenti risultati, i cui dati estesi sono consultabili nell'**Appendice G**:

- **BookKeeper ($+25,59\%$):** La neutralizzazione degli *smell* ha prodotto la de-classificazione di **20 metodi**, con un beneficio sul dataset target (ΔB) dello **0,922%** e un impatto sul sistema totale (ΔA) dello **0,322%**.
- **OpenJPA ($+16,40\%$):** L'intervento ha generato la de-classificazione di **262 metodi**, corrispondenti a un beneficio target (ΔB) dell'**1,606%** e a una riduzione del rischio complessivo (ΔA) dello **0,595%**.

In entrambi i casi, il **Fattore Smell** evidenzia come un metodo degradato presenti una probabilità di errore superiore rispetto alla *baseline* del gruppo di controllo C. L'affidabilità delle stime prodotte dalla *What-if Analysis* è subordinata al rispetto di tre postulati metodologici che garantiscono l'isolamento dell'effetto del *refactoring*. In primo luogo, l'**invarianza dei parametri del classificatore** assicura che il modello operi come un oracolo imparziale; mantenendo pesi e confini decisionali costanti tra lo scenario reale e quello simulato, si garantisce che ogni variazione nella predizione sia funzione esclusiva della modifica delle feature. Parallelamente, si assume l'**indipendenza delle feature**, postulando che la manipolazione della variabile *Number of Smells* non alteri le altre dimensioni del sistema. Sebbene nel contesto operativo le metriche di qualità siano spesso interdipendenti e correlate alla densità degli *smell*, questa separazione metodologica è necessaria per isolare il contributo strutturale puro. Tale assunzione permette di rispondere con precisione alla domanda sulla sensibilità del modello, quantificando il vantaggio derivante dalla sola rimozione delle anomalie strutturali ed evitando che il beneficio stimato venga distorto dalla variazione concomitante di altre variabili del codice. Infine, l'accuratezza poggia sull'**equivalenza semantica del refactoring**: si ipotizza che il *refactoring* avvenga tramite trasformazioni (*behavior-preserving*) che incrementano la qualità del codice senza alterarne le funzionalità originali.

5 Discussione e Minacce alla validità

Al termine dello studio, l'incrocio dei dati sperimentali permette di rispondere in modo puntuale alle domande di ricerca iniziali.

- **Risposta alla RQ1**: L'analisi dimostra che le performance dei classificatori sono strettamente legate alla volumetria dei dati: per *BookKeeper* (dataset ridotto) l'algoritmo **IBk** eccelle nel bilanciamento della *Recall*, mentre per *OpenJPA* (dataset ampio) la natura ensemble di **Random Forest** garantisce una superiore capacità di ranking (AUC) ed efficienza operativa (*NPofB20*).
- **Risposta alla RQ2**: La *What-if Analysis* quantifica il valore del refactoring nella riduzione del debito tecnico, evidenziando che la rimozione dei *Code Smell* riduce la rischiosità relativa del **25,59%** per *BookKeeper* e del **16,40%** per *OpenJPA*, portando alla neutralizzazione preventiva di complessivi **282 metodi** potenzialmente difettosi.

Tali conclusioni devono tuttavia essere interpretate alla luce di alcune limitazioni metodologiche. In termini di **validità interna**, i risultati potrebbero risentire di errori sistematici nel tracciamento dei bug (*SZZ*) o del *refactorer bias* introdotto durante l'intervento manuale; inoltre, la parzialità del set di feature potrebbe aver indotto il modello a sovrastimare il peso degli smell in presenza di variabili latenti non tracciate. La **validità di costrutto** resta intrinsecamente legata alla capacità di generalizzazione del classificatore selezionato e alla precisione delle metriche estratte dai tool di analisi statica. Riguardo alla **validità esterna**, la generalizzabilità dei risultati è limitata all'ecosistema Apache e al linguaggio Java.

6 Conclusioni e Sviluppi futuri

I risultati dello studio evidenziano che la rimozione mirata dei *Code Smell* non solo migliora la qualità strutturale, ma riduce drasticamente la probabilità di errore futuro, neutralizzando potenzialmente centinaia di bug in sistemi complessi. L'implicazione principale dello studio risiede nella possibilità per i team di sviluppo di ottimizzare l'allocazione delle risorse di testing, intervenendo in modo mirato soltanto sulle componenti a maggior rischio e garantendo una superiore affidabilità del software lungo l'intero ciclo di vita. Le evidenze emerse aprono la strada a diverse opportunità di approfondimento per consolidare l'efficacia della manutenzione predittiva. In primo luogo, un'estensione naturale del lavoro riguarda l'ottimizzazione della **Feature Selection**: mentre in questa sede si è optato per un approccio di tipo *filter*, l'esplorazione di tecniche **wrapper** potrebbe rivelare se l'interazione dinamica tra sottoinsiemi di metriche possa ulteriormente affinare la sensibilità dei modelli. Parallelamente, l'approccio proposto potrebbe evolvere verso una **completa automazione del refactoring**, integrando tool di sviluppo in grado di suggerire trasformazioni di codice *smell-aware* ottimizzate per massimizzare la riduzione del rischio in tempo reale. Infine, la validità del miglioramento riscontrato merita di essere testata attraverso una **replicazione dello studio** su un numero più vasto di progetti, inclusi contesti industriali e linguaggi di programmazione differenti, unitamente a una **validazione longitudinale** sulle release future. Tale prospettiva permetterebbe di verificare la stabilità dei modelli in diversi ecosistemi organizzativi, trasformando la prevenzione dei bug da una stima statistica a una metrica operativa standardizzata per la gestione strategica del debito tecnico.

A Appendice: Risultati Analisi di Correlazione

In questa sezione vengono riportati i dettagli analitici relativi alla correlazione statistica di Spearman (ρ) calcolata per i dataset di *BookKeeper* e *OpenJPA*. I coefficienti indicano la forza e la direzione della relazione tra le metriche estratte (suddivise in Actionable e Non-Actionable) e la presenza di difetti nei metodi analizzati. Tutti i test statistici sono stati condotti con un livello di confidenza del 95%.

Feature	**Spearman (ρ)**	**p-value**	**Classe**
MethodHistories	0,6467	$<$ 0,0001	Non-Actionable
DistinctAuthors	0,6383	$<$ 0,0001	Non-Actionable
Churn	0,6174	$<$ 0,0001	Non-Actionable
StmtAdded	0,6171	$<$ 0,0001	Non-Actionable
StmtDeleted	0,3110	$<$ 0,0001	Non-Actionable
StatementCount	0,0855	$<$ 0,0001	Actionable
Number of Smells	0,0647	$<$ 0,0001	Actionable
CyclomaticComplexity	0,0552	$<$ 0,0001	Actionable
CognitiveComplexity	0,0517	$<$ 0,0001	Actionable
NestingDepth	0,0487	$<$ 0,0001	Actionable
LocalVariableCount	0,0270	$<$ 0,0001	Actionable
ReturnTypeComplexity	-0,0265	$<$ 0,0001	Actionable
LOC	0,0145	0,0122	Actionable
ParameterCount	0,0059	0,3119	Actionable

Tabella 1: Coefficienti di correlazione di Spearman per il progetto BookKeeper.

Feature	**Spearman (ρ)**	**p-value**	**Classe**
MethodHistories	0,5040	$<$ 0,0001	Non-Actionable
StmtAdded	0,4345	$<$ 0,0001	Non-Actionable
Churn	0,4278	$<$ 0,0001	Non-Actionable
DistinctAuthors	0,3987	$<$ 0,0001	Non-Actionable
StmtDeleted	0,3779	$<$ 0,0001	Non-Actionable
StatementCount	0,1785	$<$ 0,0001	Actionable
LocalVariableCount	0,1726	$<$ 0,0001	Actionable
NestingDepth	0,1696	$<$ 0,0001	Actionable
CyclomaticComplexity	0,1635	$<$ 0,0001	Actionable
CognitiveComplexity	0,1624	$<$ 0,0001	Actionable
Number of Smells	0,1263	$<$ 0,0001	Actionable
LOC	0,1245	$<$ 0,0001	Actionable
ParameterCount	0,0102	$<$ 0,0001	Actionable
ReturnTypeComplexity	0,0065	0,0028	Actionable

Tabella 2: Coefficienti di correlazione di Spearman per il progetto OpenJPA.

B Appendice: Risultati Dettagliati dei Modelli Predittivi

In questa sezione sono riportati i dati analitici completi e il materiale grafico di supporto relativi alla valutazione dei modelli predittivi.

B.1 Tabelle Comparative delle Performance

I valori riportati nelle tabelle seguenti rappresentano le medie ottenute durante la procedura di *10-times 10-fold Cross-Validation*, utilizzate come base per la selezione dei *BestClassifier*.

Classificatore	Precision	Recall	AUC	Kappa	NPofB20
Naive Bayes	0,6636	0,6692	0,7704	0,2036	0,3426
Random Forest	0,6758	0,6820	0,7537	0,3067	0,3316
IBk	0,6844	0,6927	0,7648	0,3186	0,3900

Tabella 3: Performance medie per il progetto **BookKeeper**.

Classificatore	Precision	Recall	AUC	Kappa	NPofB20
Naive Bayes	0,7340	0,6707	0,8816	0,3415	0,3471
Random Forest	0,8810	0,8809	0,9534	0,7617	0,3942
IBk	0,8831	0,8829	0,9456	0,7658	0,3768

Tabella 4: Performance medie per il progetto **OpenJPA**.

B.2 Distribuzioni delle Performance

I boxplot raggruppati permettono di analizzare la variabilità e la robustezza di ciascun classificatore attraverso i diversi fold. Ogni *box* delimita l'intervallo interquartile, offrendo una misura visiva della stabilità delle predizioni per le metriche analizzate.

Figura 1: Distribuzione delle performance per il progetto **BookKeeper**.

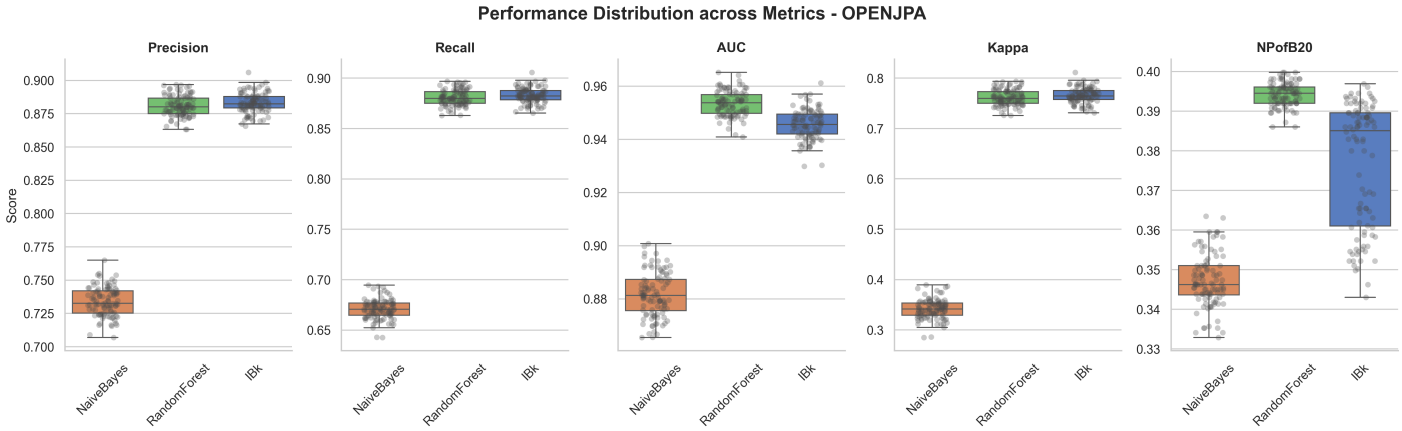

Figura 2: Distribuzione delle performance per il progetto **OpenJPA**.

C Appendice: Dettaglio della Scomposizione Funzionale

In questa sezione viene riportato il dettaglio delle metriche per i singoli sotto-metodi estratti durante l'attività di *refactoring*. Tale scomposizione illustra come la complessità del metodo originale sia stata ridistribuita in unità funzionali più piccole, facilitando l'isolamento degli *smell* residui e migliorando la leggibilità complessiva del codice.

Metodo (BookKeeper)	LOC	Cyclo	Cogn	Stmt	Smell	Smell Type
`main`	16	3	3	2	2	*Signature; GuardLog*
`startBenchmark`	29	2	1	13	2	*Signature; LawOfDemeter*
`handleZkEvent`	11	5	7	1	3	*LawOfDemeter; CatchGen; GuardLog*
`handleChildrenChanged`	24	7	7	4	0	*Nessuno*
`initializeZkState`	9	3	3	1	1	*Signature*
`validateArgs`	9	3	2	2	0	*Nessuno*

Tabella 5: Distribuzione finale del carico logico e Smell per metodo (BookKeeper)

Metodo (OpenJPA)	Stmt	Cyclo	Cogn	Nest	LOC	Smell	Smell Type
`copy()`	8	1	0	0	13	0	*Nessuno*
`copyBasicState()`	25	1	0	0	26	1	*LawOfDemeter*
`copyFetchGroupInfo()`	4	8	8	2	11	0	*Nessuno*
`copyStrategyInfo()`	3	5	4	1	7	0	*Nessuno*
`copySequenceInfo()`	2	3	2	1	6	0	*Nessuno*

Tabella 6: Distribuzione finale post-refactoring: isolamento della complessità (OpenJPA)

D Analisi Dettagliata degli Interventi di Refactoring

In questa sezione viene fornita una sintesi sistematica delle trasformazioni apportate al codice sorgente per i due casi studio. L'analisi si focalizza sul mapping tra le criticità strutturali rilevate e le tecniche di rifattorizzazione adottate, giustificando la persistenza di eventuali indicatori residui sulla base di vincoli tecnologici o architetturali inamovibili.

Code Smell	**Azione Correttiva**	**Esito**
Cognitive Complexity	**Decomposizione Funzionale**: Estratti i metodi `getOptions`, `validateArgs` e `setLedgerParams`.	**Risolto**: Logica delegata dal `main` ai metodi estratti.
Statement Count	**Extract Method**: Riduzione della logica procedurale nel corpo del `main`.	**Risolto**: Variabili locali ridotte da 20 a 3.
Nesting Depth	**Static Delegation**: Logica dei *Watcher* delegata a `handleZkEvent` e `handleChildrenChanged`.	**Risolto**: Valore ridotto da 7 a 2 (classi anonime rimosse).
Redundant Boxing	**Type Migration**: Sostituzione di `valueOf()` con i primitivi `parseInt/parseLong`.	**Risolto**: Eliminato l'overhead di creazione oggetti *wrapper*.
AccessorMethodGen.	**Visibility Refactoring**: Modifica visibilità da `private` a *package-private*.	**Risolto**: Impedita la generazione di metodi sintetici.
AvoidCatchGeneric	**Exception Specialization**: Sostituzione catch generici con blocchi mirati (`KeeperException`).	**Parziale**: Mantenuto nei *Watcher* per la resilienza di rete.
SignatureThrows	*Nessuna (Vincolo architetturale).*	**Residuo**: Mantenuto nel `main` per gestire errori critici.
GuardLogStatement	**Sintassi parametrizzata**.	**Risolto**: Uso di `{}` di SLF4J per efficienza computazionale.
Law of Demeter	*Nessuna (Vincolo API).*	**Residuo**: Ispezione catene ZooKeeper intrinseca al progetto.

Tabella 7: Mappatura Code Smells, Azioni Correttive e Analisi dei Residui (BookKeeper)

Unità Logica	**Azione di Refactoring Effettuata**
`copy()` (Originale)	Identificazione del metodo monolitico e analisi delle responsabilità eterogenee per la gestione dei metadati.
`copy()` (Refactored)	Applicazione del pattern *Delegation*: conversione del corpo del metodo in un orchestratore di chiamate specializzate.
`copyBasicState`	*Extract Method*: isolamento dei flag di base, dei membri interni e delle configurazioni di persistenza.
`copyFetchAndStrategy`	*Extract Method*: incapsulamento della logica condizionale relativa ai gruppi di fetch e alle strategie di aggiornamento.
`copySeqInverseInfo`	*Extract Method*: separazione della gestione delle sequenze e delle relazioni inverse (metadati relazionali).
Risultato Finale	Consolidamento dell'architettura modulare e rimozione delle ridondanze logiche nel punto di ingresso.

Tabella 8: Analisi delle azioni di refactoring applicate al metodo copy(): OpenJPA

E Appendice: Metriche Pre/Post Refactoring

In questa sezione vengono riportate le misurazioni di baseline per i metodi selezionati nei progetti **BookKeeper** e **OpenJPA**.

Feature (BookKeeper)	**Pre**	**Post**	**Δ%**
Line of Code (LOC)	89	16	-82,0%
Cyclomatic Complexity	17	3	-82,3%
Cognitive Complexity	22	3	-86,4%
Nesting Depth	7	1	-85,7%
Statement Count	24	2	-91,7%
Local Variable Count	20	2	-90,0%
Total Code Smells	**8**	**2**	**-75,0%**

Tabella 9: Confronto Metriche Pre/Post Refactoring: BookKeeper.

Smell rilevati (PMD): `CognitiveComplexity`, `NPathComplexity`, `SignatureDeclareThrowsException`, `LawOfDemeter`, `CyclomaticComplexity`, `MissingOverride`, `AvoidCatchingGenericException`, `GuardLogStatement`.

Feature (OpenJPA)	**Pre**	**Post**	**Δ%**
Statement Count	**38**	**7**	**-81,6%**
Line of Code (LOC)	55	13	-76,4%
Cyclomatic Complexity	14	1	-92,8%
Cognitive Complexity	14	0	-100,0%
Nesting Depth	2	0	-100,0%
Local Variable Count	0	0	0%
Total Code Smells	4	0	-100,0%

Tabella 10: Confronto Metriche Pre/Post Refactoring: OpenJPA.

Smell rilevati (PMD): `CognitiveComplexity`, `CyclomaticComplexity`, `NPathComplexity`, `LawOfDemeter`.

F Appendice: Validazione tramite Classificatore ML

In questa sezione vengono riportati i verdetti emessi dal classificatore di *defect prediction* sui metodi ottenuti a seguito del *refactoring* manuale. I risultati evidenziano la probabilità stimata dal modello che un componente sia affetto da bug (*YES*) o meno (*NO*), fornendo una validazione quantitativa dell'efficacia dell'intervento.

Metodo (BookKeeper)	**Verdetto ML**	**Probabilità YES**
`main()`	**NO**	33,33%
`startBenchmark()`	**YES**	**100,00%**
`handleZkEvent()`	**NO**	50,00%
`handleChildrenChanged()`	**YES**	**75,00%**
`findLatestLedgerId()`	**NO**	40,00%
`initializeZkState()`	**NO**	50,00%
`validateArgs()`	**NO**	0,00%

Tabella 11: Risultati della predizione post-refactoring manuale per BookKeeper.

Metodo (OpenJPA)	Verdetto ML	Probabilità YES
`copy()`	**NO**	41,56%
`copyBasicState()`	**NO**	31,22%
`copyFetchGroupInfo()`	**NO**	26,03%
`copyStrategyInfo()`	**NO**	22,73%
`copySequenceAndInverseInfo()`	**NO**	27,29%

Tabella 12: Risultati della predizione post-refactoring manuale per OpenJPA.

G Appendice: Risultati Analisi What-if

In questa sezione vengono riportati i dati aggregati relativi alla simulazione *What-if*. Il confronto evidenzia la variazione delle istanze predette come *buggy* dal classificatore passando dal dataset originale (B^+) a quello rifattorizzato ipoteticamente (B), permettendo di stimare il numero di bug evitabili attraverso la rimozione sistematica dei *code smell*.

Scenario (BookKeeper)	Istanze Attuali (A)	Istanze Predette (E)
Dataset A (Totale)	6.206	5.366
Dataset B^+ (Baseline - Smelly)	2.169	2.162
Dataset B (What-if - Refactored)	—	**2.142**
Dataset C (Controllo - Clean)	4.037	3.204

Tabella 13: Confronto delle predizioni What-if su BookKeeper.

Scenario (OpenJPA)	Istanze Attuali (A)	Istanze Predette (E)
Dataset A (Totale)	44.013	36.183
Dataset B^+ (Baseline - Smelly)	16.309	14.712
Dataset B (What-if - Refactored)	—	**14.450**
Dataset C (Controllo - Clean)	27.704	21.471

Tabella 14: Confronto delle predizioni What-if su OpenJPA.

H Formalizzazione Matematica What-if Analysis

In questa appendice vengono riportati i protocolli di calcolo e le definizioni formali utilizzate per la valutazione del beneficio del refactoring nei progetti BookKeeper e OpenJPA.

H.1 Indicatori di Beneficio e Fattore Smell

L'impatto della neutralizzazione degli *smell* viene quantificato tramite i seguenti indicatori:

1. **Efficacia sul Dataset Target (ΔB):** Percentuale di metodi *smelly* (B^+) riclassificati come *clean* a seguito della trasformazione.
$$\Delta B = \frac{E(B^+) - E(B)}{A(B^+)} \times 100 \tag{2}$$

2. **Impatto sul Sistema Totale (ΔA):** Riduzione della difettosità predetta rispetto alla totalità delle istanze del progetto (A Totale).
$$\Delta A = \frac{E(B^+) - E(B)}{A(Total)} \times 100 \tag{3}$$

3. **Fattore Smell (Risk Increment):** Incremento di rischio relativo indotto dalle anomalie strutturali rispetto alla baseline di controllo (Dataset C).
$$\text{Fattore Smell} = \frac{\text{Densità}_{B^+} - \text{Densità}_C}{\text{Densità}_C} \times 100 \tag{4}$$

H.2 Calcoli Analitici: BookKeeper

- **Istanze rimosse** $(E(B^+) - E(B))$**:** $2.162 - 2.142 = 20$
- **Densità Baseline (C):** $3.204/4.037 \approx 0,7937$
- **Densità Smelly (B+):** $2.162/2.169 \approx 0,9968$

$$\text{Fattore Smell} = \frac{0,9968 - 0,7937}{0,7937} \approx \mathbf{+25,59\%}$$

$$\Delta B = \frac{20}{2.169} \times 100 \approx \mathbf{0,922\%}$$

$$\Delta A = \frac{20}{6.206} \times 100 \approx \mathbf{0,322\%}$$

H.3 Calcoli Analitici: OpenJPA

- **Istanze rimosse** $(E(B^+) - E(B))$**:** $14.712 - 14.450 = 262$
- **Densità Baseline (C):** $21.471/27.704 \approx 0,7750$
- **Densità Smelly (B+):** $14.712/16.309 \approx 0,9021$

$$\text{Fattore Smell} = \frac{0,9021 - 0,7750}{0,7750} \approx \mathbf{+16,40\%}$$

$$\Delta B = \frac{262}{16.309} \times 100 \approx \mathbf{1,606\%}$$

$$\Delta A = \frac{262}{44.013} \times 100 \approx \mathbf{0,595\%}$$

I Appendice: Refactoring Manuale

In questa sezione viene riportato il confronto per il metodo `main()` della classe `BenchReadThroughputLatency` nel progetto Bookkeeper.

I.1 Versione Originale (Pre-Refactoring)

Listing 1: Metodo main() originale in Bookkeeper

```
public static void main(String[] args) throws Exception {
    Options options = new Options();
    options.addOption("ledger", true, "Ledger␣to␣read...");
    options.addOption("listen", true, "Listen␣for␣creation...");
    options.addOption("password", true, "Password␣used...");
    options.addOption("zookeeper", true, "Zookeeper␣ensemble...");
    options.addOption("sockettimeout", true, "Socket␣timeout...");
    options.addOption("help", false, "This␣message");

    CommandLineParser parser = new PosixParser();
    CommandLine cmd = parser.parse(options, args);

    if (cmd.hasOption("help")) {
        usage(options);
        System.exit(-1);
    }

    final String servers = cmd.getOptionValue("zookeeper", "localhost:2181");
    final byte[] passwd = cmd.getOptionValue("password", "benchPasswd").getBytes();
    final int sockTimeout = Integer.valueOf(cmd.getOptionValue("sockettimeout", "5"));

    if (cmd.hasOption("ledger") && cmd.hasOption("listen")) {
        LOG.error("Cannot␣used␣-ledger␣and␣-listen␣together");
        usage(options);
        System.exit(-1);
    }

    final AtomicInteger ledger = new AtomicInteger(0);
    final AtomicInteger numLedgers = new AtomicInteger(0);
    if (cmd.hasOption("ledger")) {
        ledger.set(Integer.valueOf(cmd.getOptionValue("ledger")));
```

```
} else if (cmd.hasOption("listen")) {
    numLedgers.set(Integer.valueOf(cmd.getOptionValue("listen")));
} else {
    LOG.error("You must use -ledger or -listen");
    usage(options);
    System.exit(-1);
}

final CountDownLatch shutdownLatch = new CountDownLatch(1);
final CountDownLatch connectedLatch = new CountDownLatch(1);
final String nodepath = String.format("/ledgers/L%010d", ledger.get());

final ClientConfiguration conf = new ClientConfiguration();
conf.setReadTimeout(sockTimeout).setZkServers(servers);

final ZooKeeper zk = new ZooKeeper(servers, 3000, new Watcher() {
    public void process(WatchedEvent event) {
        if (event.getState() == Event.KeeperState.SyncConnected
                && event.getType() == Event.EventType.None) {
            connectedLatch.countDown();
        }
    }
});
try {
    zk.register(new Watcher() {
        public void process(WatchedEvent event) {
            try {
                if (event.getState() == Event.KeeperState.SyncConnected
                        && event.getType() == Event.EventType.None) {
                    connectedLatch.countDown();
                } else if (event.getType() == Event.EventType.NodeCreated
                        && event.getPath().equals(nodepath)) {
                    readLedger(conf, ledger.get(), passwd);
                    shutdownLatch.countDown();
                } else if (event.getType() == Event.EventType.NodeChildrenChanged) {
                    if (numLedgers.get() < 0) return;
                    List<String> children = zk.getChildren("/ledgers", true);
                    List<String> ledgers = new ArrayList<String>();
                    for (String child : children) {
                        if (LEDGER_PATTERN.matcher(child).find()) ledgers.add(child);
                    }
                    Collections.sort(ledgers, ZK_LEDGER_COMPARE);
                    String last = ledgers.get(ledgers.size() - 1);
                    final Matcher m = LEDGER_PATTERN.matcher(last);
                    if (m.find()) {
                        int ledgersLeft = numLedgers.decrementAndGet();
                        Thread t = new Thread() {
                            public void run() {
                                readLedger(conf, Long.valueOf(m.group(1)), passwd);
                            }
                        };
                        t.start();
                        if (ledgersLeft <= 0) shutdownLatch.countDown();
                    } else {
                        LOG.error("Cant file ledger id in {}", last);
                    }
                } else {
                    LOG.warn("Unknown event {}", event);
                }
            } catch (Exception e) {
                LOG.error("Exception in watcher", e);
            }
        }
    });
    connectedLatch.await();
    if (ledger.get() != 0) {
        if (zk.exists(nodepath, true) != null) {
            readLedger(conf, ledger.get(), passwd);
            shutdownLatch.countDown();
        } else {
```

```
                LOG.info("Watching␣for␣creation␣of" + nodepath);
            }
        } else {
            zk.getChildren("/ledgers", true);
        }
        shutdownLatch.await();
    } finally {
        zk.close();
    }
}
```

I.2 Versione Finale (Post-Refactoring)

Listing 2: Metodo main() rifattorizzato in BookKeeper

```
package org.apache.bookkeeper.benchmark;

    public static void main(String[] args) throws Exception {
        Options options = getOptions();
        try {
            CommandLine cmd = new PosixParser().parse(options, args);
            if (cmd.hasOption("help")) {
                usage(options);
                return;
            }
            validateArgs(cmd, options);
            startBenchmark(cmd);
        } catch (ParseException e) {
            LOG.error("Parsing␣error:␣" + e.getMessage());
            usage(options);
        }
    }

    private static void startBenchmark(CommandLine cmd) throws Exception {
        final String servers = cmd.getOptionValue("zookeeper", "localhost:2181");
        final byte[] passwd = cmd.getOptionValue("password", "benchPasswd").getBytes();
        final int sockTimeout = Integer.parseInt(cmd.getOptionValue("sockettimeout", "5"));

        final AtomicInteger ledger = new AtomicInteger(0);
        final AtomicInteger numLedgers = new AtomicInteger(0);
        setLedgerParams(cmd, ledger, numLedgers);

        final CountDownLatch shutdownLatch = new CountDownLatch(1);
        final CountDownLatch connectedLatch = new CountDownLatch(1);
        final String nodepath = String.format("/ledgers/L%010d", ledger.get());

        final ClientConfiguration conf = new ClientConfiguration();
        conf.setReadTimeout(sockTimeout).setZkServers(servers);

        final ZooKeeper zk = new ZooKeeper(servers, 3000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                if (event.getState() == Event.KeeperState.SyncConnected) connectedLatch.
                    countDown();
            }
        });

        try {
            zk.register(new Watcher() {
                @Override
                public void process(WatchedEvent event) {
                    handleZkEvent(event, zk, conf, ledger, numLedgers, passwd, nodepath,
                        connectedLatch, shutdownLatch);
                }
            });
            connectedLatch.await();
            initializeZkState(zk, ledger, nodepath, conf, passwd, shutdownLatch);
            shutdownLatch.await();
        } finally {
            zk.close();
```

```
        }
    }

   // Visibilit  package-private per evitare AccessorMethodGeneration
static void handleZkEvent(WatchedEvent event, ZooKeeper zk, ClientConfiguration conf,
                                AtomicInteger ledger, AtomicInteger numLedgers, byte[]
                                    passwd,
                                String nodepath, CountDownLatch connectedLatch,
                                    CountDownLatch shutdownLatch) {
    // Rimosso catch generico per evitare AvoidCatchingGenericException
    if (event.getState() == Event.KeeperState.SyncConnected && event.getType() == Event.
        EventType.None) {
        connectedLatch.countDown();
    } else if (event.getType() == Event.EventType.NodeCreated && event.getPath().equals(
        nodepath)) {
        readLedger(conf, ledger.get(), passwd);
        shutdownLatch.countDown();
    } else if (event.getType() == Event.EventType.NodeChildrenChanged) {
        try {
            handleChildrenChanged(zk, conf, numLedgers, passwd, shutdownLatch);
        } catch (Exception e) { // Qui il catch    pi  giustificato ma meglio se specifico
            LOG.error("Errore nel recupero figli: {}", e.getMessage());
        }
    }
}

static void handleChildrenChanged(ZooKeeper zk, ClientConfiguration conf, AtomicInteger
    numLedgers,
                                      final byte[] passwd, final CountDownLatch
                                          shutdownLatch) throws KeeperException,
                                          InterruptedException {
    if (numLedgers.get() < 0) return;

    List<String> children = zk.getChildren("/ledgers", true);
    Long latestId = findLatestLedgerId(children); // Estrazione logica (Decomposizione)

    if (latestId != null) {
        final long ledgerId = latestId;
        final ClientConfiguration fConf = conf;
        if (numLedgers.decrementAndGet() <= 0) shutdownLatch.countDown();

        new Thread(new Runnable() {
            @Override public void run() { readLedger(fConf, ledgerId, passwd); }
        }).start();
    }
}

// Nuovo metodo per abbassare la complessit  di handleChildrenChanged
private static Long findLatestLedgerId(List<String> children) {
    List<String> ledgers = new ArrayList<String>();
    for (String child : children) {
        if (LEDGER_PATTERN.matcher(child).find()) ledgers.add(child);
    }
    if (ledgers.isEmpty()) return null;
    Collections.sort(ledgers, ZK_LEDGER_COMPARE);
    Matcher m = LEDGER_PATTERN.matcher(ledgers.get(ledgers.size() - 1));
    return m.find() ? Long.parseLong(m.group(1)) : null;
}

    private static void initializeZkState(ZooKeeper zk, AtomicInteger l, String p,
        ClientConfiguration c, byte[] pw, CountDownLatch s) throws Exception {
        if (l.get() != 0) {
            if (zk.exists(p, true) != null) {
                readLedger(c, l.get(), pw);
                s.countDown();
            }
        } else {
            zk.getChildren("/ledgers", true);
        }
    }
```

```
    private static void setLedgerParams(CommandLine cmd, AtomicInteger l, AtomicInteger n) {
        if (cmd.hasOption("ledger")) l.set(Integer.parseInt(cmd.getOptionValue("ledger")));
        else n.set(Integer.parseInt(cmd.getOptionValue("listen")));
    }

    private static Options getOptions() {
        Options opts = new Options();
        opts.addOption("ledger", true, "Ledger to read");
        opts.addOption("listen", true, "Listen count");
        opts.addOption("password", true, "Password");
        opts.addOption("zookeeper", true, "ZK ensemble");
        opts.addOption("sockettimeout", true, "Timeout");
        opts.addOption("help", false, "Help");
        return opts;
    }

    private static void validateArgs(CommandLine cmd, Options opts) {
        if (cmd.hasOption("ledger") && cmd.hasOption("listen") || (!cmd.hasOption("ledger")
            && !cmd.hasOption("listen"))) {
            LOG.error("Must use -ledger OR -listen");
            System.exit(-1);
        }
    }

    private static void usage(Options options) {
        new HelpFormatter().printHelp("BenchReadThroughputLatency <options>", options);
    }
}
```

I.3 Versione Originale (Pre-Refactoring)

Listing 3: Metodo copy() originale in OpenJPA

```
/**
 * Copy state from the given field to this one. Do not copy mapping
 * information.
 */
public void copy(FieldMetaData field) {
    super.copy(field);

    _intermediate = field.usesIntermediate();
    _implData = field.usesImplData();

    // copy field-level info; use get methods to force resolution of lazy data
    _proxyClass = field.getProxyType();
    _initializer = field.getInitializer();
    _transient = field.isTransient();
    _nullValue = field.getNullValue();
    _manage = field.getManagement();
    _explicit = field.isExplicit();
    _extName = field.getExternalizer();
    _extMethod = DEFAULT_METHOD;
    _factName = field.getFactory();
    _factMethod = DEFAULT_METHOD;
    _extString = field.getExternalValues();
    _extValues = Collections.EMPTY_MAP;
    _fieldValues = Collections.EMPTY_MAP;
    _primKey = field.isPrimaryKey();
    _backingMember = field._backingMember;
    _enumField = field._enumField;
    _lobField = field._lobField;
    _serializableField = field._serializableField;
    _generated = field._generated;
    _mappedByIdValue = field._mappedByIdValue;
    _isElementCollection = field._isElementCollection;
    _access = field._access;
    _orderDec = field._orderDec;

    // embedded fields can't be versions
```

```
    if (_owner.getEmbeddingMetaData() == null && _version == null)
        _version = (field.isVersion()) ? Boolean.TRUE : Boolean.FALSE;

    // only copy this data if not already set explicitly in this instance
    if (_dfg == 0) {
        _dfg = (field.isInDefaultFetchGroup()) ? DFG_TRUE : DFG_FALSE;
        if (field.isDefaultFetchGroupExplicit())
            _dfg |= DFG_EXPLICIT;
    }
    if (_fgSet == null && field._fgSet != null)
        _fgSet = new HashSet(field._fgSet);
    if (_lfg == null)
        _lfg = field.getLoadFetchGroup();
    if (_lrs == null)
        _lrs = (field.isLRS()) ? Boolean.TRUE : Boolean.FALSE;
    if (_valStrategy == -1)
        _valStrategy = field.getValueStrategy();
    if (_upStrategy == -1)
        _upStrategy = field.getUpdateStrategy();
    if (ClassMetaData.DEFAULT_STRING.equals(_seqName)) {
        _seqName = field.getValueSequenceName();
        _seqMeta = null;
    }
    if (ClassMetaData.DEFAULT_STRING.equals(_inverse))
        _inverse = field.getInverse();

    // copy value metadata
    _val.copy(field);
    _key.copy(field.getKey());
    _elem.copy(field.getElement());
}
```

I.4 Versione Finale (Post-Refactoring)

Listing 4: Metodo copy() rifattorizzato in OpenJPA

```
/**
 * Metodo orchestratore post-refactoring.
 * La logica e stata delegata a metodi specializzati per ridurre
 * lo StatementCount.
 */
public void copy(FieldMetaData field) {
    super.copy(field);

    copyBasicState(field);
    copyFetchGroupInfo(field);
    copyStrategyInfo(field);
    copySequenceAndInverseInfo(field);

    _val.copy(field);
    _key.copy(field.getKey());
    _elem.copy(field.getElement());
}

private void copyBasicState(FieldMetaData field) {
    _intermediate = field.usesIntermediate();
    _implData = field.usesImplData();
    _proxyClass = field.getProxyType();
    _initializer = field.getInitializer();
    _transient = field.isTransient();
    _nullValue = field.getNullValue();
    _manage = field.getManagement();
    _explicit = field.isExplicit();
    _extName = field.getExternalizer();
    _extMethod = DEFAULT_METHOD;
    _factName = field.getFactory();
    _factMethod = DEFAULT_METHOD;
    _extString = field.getExternalValues();
    _extValues = Collections.EMPTY_MAP;
    _fieldValues = Collections.EMPTY_MAP;
```

```
    _primKey = field.isPrimaryKey();
    _backingMember = field._backingMember;
    _enumField = field._enumField;
    _lobField = field._lobField;
    _serializableField = field._serializableField;
    _generated = field._generated;
    _mappedByIdValue = field._mappedByIdValue;
    _isElementCollection = field._isElementCollection;
    _access = field._access;
    _orderDec = field._orderDec;
}

private void copyFetchGroupInfo(FieldMetaData field) {
    if (_dfg == 0) {
        _dfg = (field.isInDefaultFetchGroup()) ? DFG_TRUE : DFG_FALSE;
        if (field.isDefaultFetchGroupExplicit())
            _dfg |= DFG_EXPLICIT;
    }
    if (_fgSet == null && field._fgSet != null)
        _fgSet = new HashSet(field._fgSet);
    if (_lfg == null)
        _lfg = field.getLoadFetchGroup();
    if (_lrs == null)
        _lrs = (field.isLRS()) ? Boolean.TRUE : Boolean.FALSE;
}

private void copyStrategyInfo(FieldMetaData field) {
    if (_owner.getEmbeddingMetaData() == null && _version == null)
        _version = (field.isVersion()) ? Boolean.TRUE : Boolean.FALSE;

    if (_valStrategy == -1)
        _valStrategy = field.getValueStrategy();
    if (_upStrategy == -1)
        _upStrategy = field.getUpdateStrategy();
}

private void copySequenceAndInverseInfo(FieldMetaData field) {
    if (ClassMetaData.DEFAULT_STRING.equals(_seqName)) {
        _seqName = field.getValueSequenceName();
        _seqMeta = null;
    }
    if (ClassMetaData.DEFAULT_STRING.equals(_inverse))
        _inverse = field.getInverse();
}
```

Riferimenti bibliografici

[1] B. Vandehei, D. A. Da Costa e D. Falessi, «Leveraging the Defects Life Cycle to Label Affected Versions,» *ACM Transactions on Software Engineering and Methodology*, vol. 30, p. 35, 2021.

[2] D. Falessi, S. Mesiano Laureani, J. Çarka, M. Esposito e D. A. Da Costa, «Enhancing the defectiveness prediction of methods and classes via JIT,» *Empirical Software Engineering*, vol. 28, n. 37, p. 43, 2023.

[3] J. Çarka, D. Falessi e M. Esposito, «On effort-aware metrics for defect prediction,» *Empirical Software Engineering*, vol. 27, n. 152, p. 38, 2022.

[4] D. Falessi, L. Narayana, J. F. Thai e B. Turhan, «Preserving Order of Data When Validating Defect Prediction Models,» p. 20.

[5] D. Falessi, B. Russo e M. Kathleen, «What if I had no smells?,» *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 17, 2017.

Disponibilità del materiale

Il codice sorgente, i dataset generati e i risultati completi delle analisi descritte in questo lavoro sono consultabili pubblicamente ai seguenti indirizzi:

GitHub Repository: `https://github.com/GaiaMeola/Progetto_ISW2`

SonarCloud Dashboard: `https://sonarcloud.io/project/overview?id=GaiaMeola_Progetto_ISW2`