Analisi della Buggyness nei Metodi Software

# Abstract

In questo progetto di ingegneria del software viene realizzato un dataset a partire da due progetti open source, finalizzato allo studio della predizione della buggyness a livello di metodo software. Il dataset comprende informazioni dettagliate sui metodi presenti in diverse release, con annotazioni sul loro stato buggy o non buggy. Su tale dataset vengono addestrati e valutati diversi modelli di machine learning con l’obiettivo di prevedere se, all’interno di una specifica release, un metodo risulti buggy. L’analisi consente di confrontare le prestazioni di modelli classici e avanzati, individuando le caratteristiche maggiormente rilevanti per la predizione degli errori. I risultati ottenuti evidenziano la possibilità di identificare con buona accuratezza i metodi problematici, fornendo un supporto significativo alle attività di testing e manutenzione. Il progetto contribuisce a migliorare la gestione della qualità del software tramite approcci data-driven, promuovendo processi di sviluppo più efficienti e affidabili.

# Introduzione

La manutenibilità del software rappresenta una delle sfide più rilevanti nello sviluppo di sistemi complessi e in continua evoluzione. Il codice sorgente, infatti, subisce modifiche frequenti per correggere errori, aggiungere funzionalità o adattarsi a nuovi requisiti, e tali interventi possono influire significativamente sulla qualità complessiva del software. In particolare, la presenza di difetti (bug) nei metodi software è un indicatore critico di rischio che può compromettere la stabilità e l’affidabilità del sistema, oltre a incrementare i costi e i tempi di manutenzione. Per questi motivi, comprendere e prevedere la propensione ai difetti a livello di singolo metodo diventa un elemento chiave per supportare processi di sviluppo più efficaci e orientati alla qualità.

Nonostante l’ampio interesse verso la qualità del codice, esiste una carenza di evidenze empiriche solide riguardo all’impatto della riduzione dei cosiddetti “code smells” e altre metriche di qualità sul livello di buggyness dei singoli metodi nelle release software. Gran parte della letteratura si concentra sull’analisi a livello di classi o moduli, mentre la granularità metodologica, cruciale per interventi mirati, è ancora poco esplorata. Questa lacuna limita la possibilità di definire strategie preventive e di ottimizzare le attività di testing focalizzandosi sulle unità di codice più a rischio.

Il presente studio si inserisce in questo contesto con l’obiettivo di colmare tale gap attraverso un’analisi empirica basata su due progetti open source. Viene creato un dataset dettagliato contenente informazioni sulle modifiche e sullo stato buggy dei metodi in più release, da cui vengono estratte caratteristiche significative per la costruzione di modelli predittivi. Le domande di ricerca fondamentali a cui si risponde sono: quale classificatore è in grado di prevedere con maggiore accuratezza se un metodo sarà buggy in una data release? Quanti metodi buggy si potrebbero potenzialmente evitare applicando tecniche di predizione efficaci? La risposta a queste domande permette di valutare l’efficacia dei modelli di machine learning nell’identificazione precoce di metodi problematici, offrendo così un contributo pratico e teorico alla gestione della qualità del software.

La struttura del lavoro è organizzata come segue: nella Sezione 2 vengono descritti i dati raccolti e le caratteristiche del dataset; nella Sezione 3 si illustrano i modelli predittivi adottati e le metodologie di valutazione; nella Sezione 4 vengono presentati e discussi i risultati ottenuti; infine, nella Sezione 5 si forniscono le conclusioni e le prospettive future di ricerca.

# Misurazioni e metodologia

## Dataset & Label Extraction

Il dataset utilizzato per l’analisi è stato creato a partire da due progetti open source, Bookkeeper e OpenJPA, entrambi con un numero significativo di release ottenute tramite il sistema di tracciamento Jira. Per ciascuna release sono stati raccolti i metodi presenti, insieme alle loro caratteristiche statiche e dinamiche. L’estrazione delle etichette “buggy” è stata eseguita collegando i commit ai ticket Jira utilizzando la tecnica Proportion per assegnare le versioni affette ai ticket privi di tale informazione esplicita. Questa metodologia ha consentito di associare in modo più completo i difetti ai metodi coinvolti nelle varie release. Il dataset finale comprende informazioni su metodi e release, con una varietà di metriche relative alla struttura del codice e alla sua evoluzione nel tempo.

## Metrics & Definitions

Nel progetto sono state utilizzate diverse caratteristiche per descrivere ogni metodo all’interno di una specifica release, includendo sia metriche strutturali, come linee di codice (LOC), numero di statement, complessità ciclomatica, complessità cognitiva, branch points e nesting depth, sia metriche storiche, come il numero di autori, il churn complessivo e medio, le linee aggiunte e rimosse, il numero di modifiche (MethodHistories) e l’età del metodo. Queste informazioni sono state estratte a livello di metodo sfruttando strumenti basati su JavaParser, garantendo così un’elevata precisione e granularità.

L’efficacia dei modelli predittivi è stata valutata attraverso quattro metriche principali: l’AUC (Area Under the Curve), che misura la capacità del classificatore di distinguere correttamente metodi buggy da quelli non buggy; la Kappa, che valuta la qualità della classificazione rispetto al caso casuale; la precision, che rappresenta la proporzione di veri positivi tra quelli predetti come buggy; e il recall, che indica la proporzione di metodi effettivamente buggy che sono stati correttamente identificati.

L’analisi delle correlazioni statistiche tra ciascuna metrica e la variabile target Buggy, condotta utilizzando i coefficienti di Pearson e Spearman, ha messo in evidenza una marcata differenza nella natura dei due progetti. In Bookkeeper, le metriche storiche si sono dimostrate più rilevanti rispetto a quelle strutturali. In particolare, si osservano forti correlazioni negative con la presenza di bug per metriche come AvgChurn (Pearson pari a -0.40), AddedLines (Spearman pari a -0.35), AvgAddedLines (Pearson pari a -0.36) e MethodHistories (Pearson pari a -0.31). Questo suggerisce che metodi frequentemente modificati ma con modifiche di entità contenuta tendono a stabilizzarsi nel tempo, risultando meno soggetti a introdurre bug.

Nel caso di OpenJPA, invece, le metriche strutturali risultano predominanti. Si osservano, ad esempio, correlazioni positive tra la presenza di bug e LOC (Pearson pari a 0.42), Statement (Pearson pari a 0.39), NestingDepth (Pearson pari a 0.36) e Cyclomatic Complexity (Pearson pari a 0.35). Ciò indica che la complessità interna del codice gioca un ruolo cruciale nella determinazione della qualità, e che metodi più lunghi, profondi o complessi sono maggiormente associati a difetti.

Queste differenze evidenziano due approcci progettuali profondamente diversi. In Bookkeeper la qualità sembra dipendere maggiormente dalla stabilità del metodo nel tempo, suggerendo un’organizzazione che privilegia l’evoluzione controllata e il mantenimento continuo. In OpenJPA, al contrario, la qualità è legata alla semplicità strutturale, suggerendo una progettazione più focalizzata sulla leggibilità e sulla manutenibilità del singolo metodo. Di conseguenza, i modelli predittivi per i due progetti devono tenere conto non solo delle metriche disponibili, ma anche del contesto di sviluppo che queste metriche riflettono.

## Experimental Protocol

Il protocollo sperimentale adottato si basa su uno schema temporale walk-forward, in cui ogni release viene predetta utilizzando esclusivamente i dati provenienti dalle versioni precedenti. Questa strategia, oltre a prevenire fenomeni di data leakage, simula realisticamente lo scenario in cui un modello predittivo viene impiegato per supportare decisioni su nuove versioni del software.

Per la selezione automatica delle feature è stato utilizzato l'information gain con soglia pari a 1. Questo approccio ha permesso di ridurre la dimensionalità del problema mantenendo prestazioni sostanzialmente invariate. In Bookkeeper, ad esempio, il numero di feature è passato da 22 a 13, mentre in OpenJPA si è ridotto da 22 a 18. Nonostante ciò, i modelli costruiti su queste sottoinsiemi hanno mostrato performance molto simili a quelli basati su tutte le feature. Nel caso di Bookkeeper, ad esempio, la AUC del classificatore Random Forest si è mantenuta stabile (da 0.74 con tutte le feature a 0.73 con info gain), così come la precisione (0.55 con tutte le feature e 0.54 con info gain). Questo risultato dimostra che, sebbene l'approccio none produca risultati validi, l’uso dell’information gain consente di semplificare il modello, riducendo la complessità computazionale e migliorando la comprensibilità, senza compromettere in modo significativo la capacità predittiva.

Sono state inoltre considerate strategie alternative di selezione automatica, come la forward search e la backward elimination. Tuttavia, la backward elimination si è dimostrata rapidamente impraticabile a causa dell’elevato costo computazionale necessario per esplorare tutte le combinazioni di feature. La forward search, invece, ha prodotto risultati deludenti in entrambi i progetti: nel caso di Bookkeeper si è rivelata inefficiente, generando insiemi di feature sub-ottimali e modelli con AUC inferiori a quelli ottenuti con info gain; nel caso di OpenJPA, la selezione si è interrotta prematuramente dopo pochissime iterazioni, restituendo modelli incapaci di superare una AUC di 0.65, ben al di sotto delle prestazioni ottenute con info gain (pari a 0.72).

I classificatori testati includono Naive Bayes, Random Forest e IBk (k-Nearest Neighbors). Tra questi, Random Forest si è confermato il modello più efficace per entrambi i progetti, garantendo un buon equilibrio tra accuratezza, robustezza e interpretabilità, specialmente se combinato con la riduzione delle feature tramite info gain. Tutti gli esperimenti sono stati eseguiti con seed fisso, garantendo la piena riproducibilità dei risultati.

## Assumptions / Shortcuts

Nell’analisi si è assunta l’indipendenza tra le coppie metodo–release, trattandole come unità distinte, anche se in realtà possono esistere dipendenze temporali tra versioni consecutive. La scelta di limitare la sperimentazione a tre classificatori noti è stata motivata dall’obiettivo di focalizzarsi su modelli ampiamente studiati e interpretabili. È stato considerato che, qualora un metodo non fosse stato modificato in una release, le sue metriche statiche (come LOC e numero di statement) restassero invariate rispetto alla release precedente. Il refactoring manuale rilevante è stato eseguito da un singolo sviluppatore, pertanto cambiamenti stilistici minori e non funzionali sono stati ignorati nel processo di analisi. Poiché non sono disponibili strumenti di analisi statica specifici a livello di metodo, sono stati implementati estrattori di metriche personalizzati basati sul parsing del codice sorgente con l’ausilio di JavaParser. Questa soluzione ha consentito di calcolare in modo affidabile le metriche di complessità e struttura a livello metodologico, riducendo potenziali errori di attribuzione e garantendo coerenza nell’estrazione delle feature.

# Risultati

Questo capitolo presenta i risultati ottenuti dall’analisi dei progetti Bookkeeper e OpenJPA, con l’obiettivo di individuare metodi candidati al refactoring sulla base della correlazione tra caratteristiche del codice e presenza di difetti.

## Rationale for Selecting the Target Method

L’analisi della correlazione tra le feature raccolte e la bugginess ha guidato la selezione delle feature actionable da usare come criterio prioritario di refactoring. Per il progetto Bookkeeper, la metrica con maggiore correlazione assoluta è risultata essere la complessità ciclomatica (ρ = 0.059 Spearman), sebbene di valore basso. È stata comunque scelta in quanto metrica interpretabile, ben supportata in letteratura, e direttamente mitigabile tramite refactoring. È stato quindi selezionato il metodo con massima complessità ciclomatica nella release più recente, ovvero processPacket nella classe BookieServer, versione 4.2.1, con un valore di 34.

Per OpenJPA, invece, la metrica con la più alta correlazione è risultata il numero di statement (ρ = 0.253 Spearman). Anche in questo caso, si tratta di una caratteristica significativa e correggibile. È stato dunque selezionato il metodo eval della classe JPQLExpressionBuilder, versione 1.2.0, con 175 statement.

Questi due metodi risultano buoni candidati al refactoring: processPacket è al centro del flusso di comunicazione tra client e bookie in Bookkeeper, quindi ha alta criticità architetturale e impatto sul core del sistema; eval in OpenJPA rappresenta la logica di parsing e valutazione delle query JPQL, ed è pertanto centrale per l’intero sistema di persistenza.

## Description of the Selected Method

Il metodo processPacket in BookieServer si occupa di gestire pacchetti di richiesta provenienti dal client, instradandoli a seconda del tipo (read, write, addEntry). Il metodo contiene 140 righe di codice e 118 statement, con una complessità ciclomatica pari a 34 e una profondità di annidamento di 10. Le sue caratteristiche suggeriscono elevata ramificazione logica, confermata dalla presenza di 33 branch points e una complessità cognitiva di 98. Non è presente una storia di modifiche rilevante (1 modifica nella release corrente) e le operazioni di churn sono nulle, indicando una recente stabilità. Tuttavia, la struttura attuale suggerisce code smells come “Long Method”, “God Method” e “Feature Envy”.

Per OpenJPA, il metodo eval della classe JPQLExpressionBuilder ha 261 linee di codice, 175 statement, una complessità ciclomatica di 82 e una cognitiva di 171. Il metodo non ha subito modifiche recenti ma presenta una struttura fortemente complessa, probabilmente associata a un singolo costrutto monolitico per la valutazione delle query. Anche in questo caso si osservano code smells simili, soprattutto “Long Method” e “Complex Conditional”.

Questi dati confermano la complessità strutturale dei due metodi selezionati e giustificano la scelta di intervenire su di essi [Snasphot dati].

## Criteria for Identifying Refactoring Targets within the Method

L’individuazione dei target di refactoring è stata guidata dalla necessità di affrontare code smells quali metodi troppo lunghi e complessi, che indicavano l’opportunità di applicare la tecnica di Extract Method. Questa scelta ha permesso di frammentare il codice monolitico in componenti più piccoli e semanticamente coerenti, riducendo così la complessità e facilitando la manutenzione.

La regola di prioritizzazione ha previsto la riduzione della metrica principale — complessità ciclomatica per processPacket e numero di statement per eval — senza aumentare le altre feature correlate positivamente alla bugginess, come la complessità cognitiva, i branch points o la profondità di annidamento. Questa strategia ha assicurato un miglioramento complessivo della qualità del codice, applicando l’estrazione di metodi solo quando si poteva garantire un beneficio misurabile e sostenibile.

## Resulting Refactored Method

Il refactoring ha interessato entrambi i metodi con l’estrazione di sottocomponenti funzionali, portando a una drastica riduzione delle metriche critiche. Per processPacket, il metodo originario di 140 righe, 118 statement e complessità ciclomatica 34 è stato suddiviso in tre metodi. Il metodo principale è ora ridotto a 30 righe con complessità ciclomatica pari a 7, mentre i metodi handleReadEntry e handleAddEntry contano rispettivamente 78 e 23 righe. Per eval, originariamente con 261 righe e 175 statement, il refactoring ha prodotto quattro metodi più piccoli, con il principale ridotto a 37 righe e 13 statement, mentre la complessità ciclomatica è passata da 82 a 24.

La compilazione e i test unitari hanno confermato la corretta funzionalità del codice refattorizzato. Nel caso di OpenJPA, durante la fase di testing sono stati rilevati alcuni errori, tuttavia si è verificato che tali errori erano già presenti prima del refactoring e non coinvolgevano il metodo oggetto dell’intervento. Probabilmente queste problematiche sono dovute a incompatibilità tra le versioni di Java utilizzate, dato che il progetto è strutturato per Java 5, una versione ormai obsoleta e difficilmente installabile nei moderni ambienti di sviluppo. Le metriche mostrano una riduzione significativa, con la complessità ciclomatica di processPacket diminuita di circa il 79% e il numero di statement di eval ridotto di oltre il 90%.

## What-If Analysis of Avoidable Buggy Methods

Per valutare l’impatto potenziale di interventi mirati sui metodi difettosi, è stata condotta una simulazione di tipo what-if utilizzando i modelli precedentemente addestrati. L’obiettivo è stimare quanti difetti potrebbero essere evitati ipotizzando una rimozione completa delle feature correlate a cattive pratiche — come, ad esempio, l’elevata complessità — nei metodi identificati come problematici. L’esperimento è basato sulla riduzione a zero delle feature actionable identificate tramite l’analisi di correlazione descritta nei paragrafi precedenti. Questa simulazione permette di quantificare il miglioramento della manutenibilità del software in termini di riduzione dei difetti previsti dal modello.

Nel caso di Bookkeeper, come mostrato nella Tabella x, il dataset iniziale (A) contiene 7132 metodi effettivamente buggy. La porzione B+ rappresenta i metodi buggy in cui è presente almeno una delle feature actionable; in questo sottoinsieme si contano 7032 metodi. La porzione C, al contrario, include i soli metodi buggy in cui tali feature sono già assenti (solo 100 metodi). Quando le feature actionable in B+ vengono simulate come rimosse (dataset B), il numero di metodi predetti come buggy dal classificatore si riduce da 7032 a 6985. Si osserva quindi una diminuzione di 47 metodi buggy previsti, corrispondente a una riduzione del 6.7% rispetto al sottoinsieme B+ e di circa lo 0.66% rispetto all’intero dataset A. Sebbene l’effetto assoluto sembri contenuto, va sottolineato che la rimozione ha riguardato solo un numero selezionato di feature, e che il modello mantiene una soglia di cautela elevata nella classificazione.

Tabella : risultati analisi waht-if su Bookkeeper

|  |  |  |
| --- | --- | --- |
| Dataset | Presenti | Predetti |
| A | 7132 | 7132 |
| Bplus | 7032 | 7032 |
| C | 100 | 100 |
| B | - | 6985 |

Per quanto riguarda OpenJPA, i risultati mostrati in Tabella y mostrano come il dataset A contiene 14912 metodi effettivamente buggy, mentre il modello ne predice 15511. Il sottoinsieme B+ rappresenta i 14334 metodi in cui le feature actionable sono presenti, e il sottoinsieme C include i rimanenti 578. Dopo aver impostato a zero le feature actionable in B+, ottenendo il dataset B, il numero di metodi predetti come buggy scende da 14938 a 14452. La riduzione è quindi di 486 metodi, pari a un miglioramento relativo del 3.3% su B+ e del 3.1% su tutto il dataset A. A differenza di Bookkeeper, qui l’effetto della simulazione è più marcato, riflettendo probabilmente il ruolo più centrale delle metriche strutturali nella predizione dei difetti nel progetto OpenJPA.

Tabella : risultati analisi what-if su OpenJPA

|  |  |  |
| --- | --- | --- |
| Dataset | Presenti | Predetti |
| A | 14912 | 15511 |
| Bplus | 14334 | 14938 |
| C | 578 | 573 |
| B | - | 14452 |

Questa analisi si basa sull’assunzione di indipendenza condizionata, ovvero si assume che la modifica delle feature actionable non alteri in modo significativo la distribuzione delle altre variabili. In pratica, si ipotizza che la riduzione della complessità non comporti effetti collaterali negativi su altri aspetti del metodo. Sebbene tale assunzione sia idealizzata, permette di ottenere una stima conservativa del potenziale beneficio: i risultati indicano che, con semplici refactoring mirati alla riduzione di specifici fattori strutturali, sarebbe possibile evitare fino a 486 metodi buggy in OpenJPA e 47 in Bookkeeper secondo le previsioni del classificatore.

# Discussioni e minacce

I risultati ottenuti permettono di rispondere in modo diretto alle due domande di ricerca. Per la RQ1, il classificatore più efficace è risultato essere Random Forest, che ha ottenuto le migliori prestazioni in termini di AUC e recall, dimostrandosi il più adatto a identificare metodi buggy sia in Bookkeeper che in OpenJPA. Il suo comportamento stabile e la capacità di gestire feature rumore confermano l’efficacia di modelli ensemble in contesti ad alta dimensionalità.

Per quanto riguarda la RQ2, l’analisi what-if mostra che riducendo a zero le feature più correlate ai difetti — come i code smells — si otterrebbe una riduzione attesa di 47 metodi buggy in Bookkeeper e 486 in OpenJPA. Questo indica un miglioramento più marcato della manutenibilità per quest’ultimo progetto, dove le correlazioni tra qualità interna e presenza di bug sono più forti. Bookkeeper, al contrario, presenta una struttura meno sensibile alle metriche di qualità, suggerendo una maggiore robustezza o una diversa origine dei difetti.

L’analisi è soggetta a minacce alla validità interna, tra cui errori nella misura automatica delle metriche e assunzioni semplificative nell’esperimento what-if, che presume indipendenza tra le feature e la possibilità di portare a 0 qualsiasi feature actionable. Inoltre, la selezione delle metriche modificabili può essere influenzata da bias soggettivi. Nonostante questi limiti, i risultati forniscono evidenza concreta del legame tra metriche riducibili e difettosità, sostenendo l’uso di approcci predittivi per guidare il refactoring.

# Conclusioni e lavori futuri

Questo studio ha esplorato la possibilità di identificare metodi buggy all’interno di due progetti Java open source attraverso l’analisi di metriche statiche e la costruzione di modelli predittivi. Il risultato principale è che il classificatore Random Forest ha mostrato la maggiore efficacia, e che la presenza di feature correlabili ha un impatto significativo sulla probabilità di difettosità. Questi risultati suggeriscono che un’attenzione mirata alla riduzione delle feature problematiche può portare a un miglioramento concreto della manutenibilità del software.

Prossimi passi futuri potrebbero includere:

* analisi comparativa su progetti con architetture non object-oriented (es. funzionali o ibridi),
* integrazione delle metriche con dati runtime e di performance per rafforzare la predizione,
* sviluppo di strumenti interattivi per ingegneri del software che evidenzino metodi critici in tempo reale durante la codifica.

# Appendice

|  |  |  |
| --- | --- | --- |
| ***Codice*** | ***Nome*** | ***Significato*** |
| Project | Progetto | Nome del progetto software |
| Package | Pacchetto | Nome del package in cui il metodo è contenuto |
| Class | Classe | Nome della classe contenente il metodo |
| Method | Metodo | Nome del metodo analizzato |
| Version | Versione | Release software di riferimento |
| LOC | Lines of Code | Numero di linee di codice nel metodo |
| Statement | Statement | Numero di istruzioni eseguibili nel metodo |
| Cyclomatic | Complessità Ciclomatica | Misura della complessità logica del metodo (numero di percorsi indipendenti) |
| Cognitive | Complessità Cognitiva | Stima della difficoltà cognitiva nella comprensione del metodo |
| MethodHistories | Storico Metodo | Numero di modifiche storiche subite dal metodo |
| AddedLines | Righe Aggiunte | Numero di righe di codice aggiunte nel metodo rispetto alla release precedente |
| MaxAddedLines | Max Righe Aggiunte | Massimo numero di righe aggiunte in una singola modifica |
| AvgAddedLines | Media Righe Aggiunte | Media delle righe aggiunte per modifica |
| DeletedLines | Righe Eliminate | Numero di righe di codice rimosse dal metodo |
| MaxDeletedLines | Max Righe Eliminate | Massimo numero di righe eliminate in una singola modifica |
| AvgDeletedLines | Media Righe Eliminate | Media delle righe eliminate per modifica |
| Churn | Churn | Somma di righe aggiunte e eliminate (modifiche totali) |
| MaxChurn | Max Churn | Massimo churn registrato in una singola modifica |
| AvgChurn | Media Churn | Media del churn per modifica |
| BranchPoints | Punti di Branch | Numero di punti di decisione (if, switch, etc.) nel metodo |
| NestingDepth | Profondità di Annidamento | Massimo livello di annidamento di strutture di controllo |
| ParametersCount | Numero di Parametri | Numero di parametri formali del metodo |

Tabella - Features

|  |  |  |
| --- | --- | --- |
| Attributo | Pearson | Spearman |
| Version | 0,275376 | 0,259299 |
| Cyclomatic | 0,038954 | 0,059774 |
| LOC | 0,026111 | 0,049495 |
| Statement | 0,023831 | 0,049024 |
| NestingDepth | 0,008952 | 0,040925 |
| BranchPoints | 0,029815 | 0,034353 |
| Cognitive | 0,023181 | 0,032887 |
| Project | 0 | 0 |
| ParametersCount | 0,024358 | -0,03285 |
| MaxDeletedLines | 0,015392 | -0,04067 |
| DeletedLines | 0,01182 | -0,04114 |
| AvgDeletedLines | 0,041126 | -0,04506 |
| Package | 0,142781 | -0,05193 |
| Class | 0,035863 | -0,05622 |
| Method | 0,013903 | -0,07425 |
| AvgAddedLines | 0,127379 | -0,15491 |
| AddedLines | 0,019363 | -0,15898 |
| MaxChurn | 0,010166 | -0,1595 |
| MaxAddedLines | 0,027326 | -0,15985 |
| Churn | 0,009939 | -0,16039 |
| AvgChurn | 0,129992 | -0,16192 |
| MethodHistories | 0,066044 | -0,20177 |

Tabella – Correlazione Bookkeeper

|  |  |  |
| --- | --- | --- |
| Project | BOOKKEEPER | OPENJPA |
| Package | org.apache.bookkeeper.proto | org.apache.openjpa.kernel.jpql |
| Class | BookieServer | JPQLExpressionBuilder |
| Method | processPacket | eval |
| Version | 4.2.1 | 1.2.0 |
| LOC | 140 | 261 |
| Statement | 118 | 175 |
| Cyclomatic | 34 | 82 |
| Cognitive | 98 | 171 |
| MethodHistories | 1 | 0 |
| AddedLines | 5 | 0 |
| MaxAddedLines | 5 | 0 |
| AvgAddedLines | 5 | 0 |
| DeletedLines | 0 | 0 |
| MaxDeletedLines | 0 | 0 |
| AvgDeletedLines | 0 | 0 |
| Churn | 5 | 0 |
| MaxChurn | 5 | 0 |
| AvgChurn | 5 | 0 |
| BranchPoints | 33 | 81 |
| NestingDepth | 10 | 5 |
| ParametersCount | 2 | 1 |
| Buggy | false | false |

|  |  |  |
| --- | --- | --- |
| Attributo | Pearson | Spearman |
| Statement | 0,246323 | 0,253107 |
| LOC | 0,240339 | 0,252977 |
| NestingDepth | 0,25947 | 0,240619 |
| BranchPoints | 0,21581 | 0,232049 |
| Cognitive | 0,175891 | 0,231096 |
| Cyclomatic | 0,216127 | 0,228938 |
| MethodHistories | 0,178689 | 0,193495 |
| AvgAddedLines | 0,08858 | 0,122974 |
| ParametersCount | 0,116932 | 0,120317 |
| MaxDeletedLines | 0,081415 | 0,120098 |
| DeletedLines | 0,078781 | 0,119689 |
| MaxChurn | 0,138471 | 0,112985 |
| Churn | 0,117077 | 0,112401 |
| AddedLines | 0,12447 | 0,109376 |
| MaxAddedLines | 0,139279 | 0,109153 |
| AvgChurn | 0,089277 | 0,106131 |
| AvgDeletedLines | 0,113773 | 0,071031 |
| Package | 0,04389 | 0,068831 |
| Method | 0,00839 | 0,045839 |
| Project | 0 | 0 |
| Class | 0,018911 | -0,01528 |
| Version | 0,017677 | -0,05573 |

Tabella - Correlazione Openjpa

|  |  |  |
| --- | --- | --- |
| Project | BOOKKEEPER | OPENJPA |
| Package | org.apache.bookkeeper.proto | org.apache.openjpa.kernel.jpql |
| Class | BookieServer | JPQLExpressionBuilder |
| Method | processPacket | eval |
| Version | 4.2.1 | 1.2.0 |
| LOC | 30 | 37 |
| Statement | 25 | 13 |
| Cyclomatic | 7 | 24 |
| Cognitive | 14 | 49 |
| MethodHistories | 1 | 1 |
| AddedLines | 41 | 23 |
| MaxAddedLines | 41 | 23 |
| AvgAddedLines | 41 | 23 |
| DeletedLines | 43 | 29 |
| MaxDeletedLines | 43 | 29 |
| AvgDeletedLines | 43 | 29 |
| Churn | 84 | 52 |
| MaxChurn | 84 | 52 |
| AvgChurn | 84 | 52 |
| BranchPoints | 6 | 23 |
| NestingDepth | 3 | 2 |