POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Tesi di Laurea

Manutenibilità del codice sorgente scritto in Rust

Relatori
Luca Ardito
Riccardo Coppola

Candidato
Diego Verga
Student Number: s240703

Academic Year 2018-2019

Abstract

Contesto: Rust è un linguaggio di programmazione open-source, sviluppato dalla Community di Mozilla e progettato seguendo criteri di sicurezza, concorrenza e parallelismo. Ha velocità di esecuzione quasi paragonabile al C e al C++. Si vorrebbero avere informazioni riguardo alla manutenibilità dei softwares scritti in questo linguaggio, che sta guadagnando sempre maggiore popolarità e consensi.

La manutenibilità del codice è definita nello standard ISO 25010 come grado di efficacia ed efficienza con cui un prodotto o sistema può essere modificato dai manutentori previsti. È un aspetto sempre più importante del codice, che aumenta ancora di più di importanza man mano che i softwares diventano più complessi e pervasivi nella vita quotidiana. Tenere conto della manutenibilità di un codice porta diversi vantaggi, tra i quali una maggior facilità nella gestione del codice, una maggior adattabilità e una riduzione del technical debt. Stimare questo aspetto del codice, però, è piuttosto difficile e vi sono innumerevoli metriche atte a questo scopo.

Scopo: Gli obiettivi di questa tesi sono:
comprendere quali metriche sono effettivamente più utilizzate in letteratura e quali programmi vengono impiegati per computarle;
creare un programma in grado di calcolare il set di metriche più utilizzate;
utilizzare il lavoro svolto per portare le metriche più comuni in Rust e offrire così un modo per confrontare questo nuovo linguaggio con il C e il C++.

Metodo: Per rispondere al primo obiettivo si è svolta una SLR, le cui domande di ricerca sono: trovare le metriche usate in letteratura per valutare la manutenibilità di un codice; scoprire quali di queste siano le più popolari e creare una lista; trovare quali programmi siano stati utilizzati negli studi presi in considerazine; dare una selezione ideale di programmi in grado di calcolare le metriche più popolari per i linguaggi più supportati.

Per rispondere al secondo obiettivo si è creato un programma in Python che fosse open-source, modulare e facilmente estensibile.

Per svolgere il terzo punto si sono cercati dei tools per analizzare i codici scritti in Rust e si sono aggiunti al programma Python.

Risultati: Svolgendo la SLR sono stati selezionati una cinquantina di articoli, che in totale menzionavano 174 diverse metriche. Di queste, solo il 24.7% sono state citate da più di un studio. Abbiamo selezionato un set di

13 metriche e 2 suite di metriche. Abbiamo anche trovato un certo numero di programmi, sia proprietari che open-source.

Il programma creato è in grado sia di eseguire ed aggregare i risultati di altri analizzatori che di computare delle ulteriori metriche. I risultati ottenuti con l'aggregatore vengono presentati seguendo uno standard definito dall'autore della tesi.

Essendo questo un linguaggio abbastanza nuovo, al momento vi sono ben pochi analizzatori per i sorgenti scritti in Rust. Fra questi vi sono Tokei e Rust-code-analysis, entrambi in sviluppo.

Conclusioni: Svolgendo la SLR si è potuto constatare quanto sia utile trovare delle metriche efficaci per valutare la manutenibilità del codice, ma si è anche visto che non vi sono pareri unanimi sull'utilità delle varie metriche esistenti. Con il programma creato si spera di semplificare l'utilizzo delle metriche, aiutando l'utente finale ad ottenere quante più informazioni possibili sul codice analizzato.

Ringraziamenti

Desidero esprimere i miei più sinceri ringraziamenti a tutti coloro che, in un modo o nell'altro, mi hanno supportato in questo percorso.

In particolar modo voglio manifestare la mia gratitudine ad ognuno dei componenti della mia famiglia, per il sostegno e l'incoraggiamento mostratomi ogniqualvolta ce ne fosse stata la necessità.

Un grande ringraziamento va anche all'Assistente Professore Luca Ardito, all'Assistente ricercatore Riccardo Coppola ed a Luca Barbato, che mi hanno sempre dedicato il loro tempo, fornendomi aiuto con spiegazioni quanto più esaurienti possibili e rispondendo ai miei dubbi nel minor tempo possibile.

Un ringraziamento anche a Michele Valsesia che, sebbene sia entrato a far parte del team da relativamente poco tempo, in pochissimo è diventato un valido collaboratore ed un amico.

Indice

Abstract ii

Elenco delle tabelle VII

Elenco delle figure VIII

1 Introduzione 1
1.1 Manutenibilità 2
1.2 Il progetto "Code Clarity in Rust" 2
1.2.1 Fasi del progetto 3

2 Systematic Literature Review (SLR) 5
2.1 Research Questions 5
2.2 Strategie di ricerca 6
2.2.1 Librerie Digitali 6
2.2.2 Stringhe di ricerca 6
2.2.3 Criteri di Inclusione ed Esclusione 8
2.3 Ricerca degli Studi 8
2.4 Selezione degli Studi 9
2.5 Estrazione dei Dati 10
2.6 Sintesi e resoconto dei dati 11
2.7 Risultati 11
2.7.1 RQ1: le Metriche 13
2.7.2 RQ2: I Programmi 23

3 Metrics Aggregator 41
3.1 Interfaccia e Output dell'aggregatore 42
3.1.1 Interfaccia e Input 42
3.1.2 Output 44
3.2 Analizzatori 47

3.2.1 Programmi scelti 47
3.2.2 Modifiche effettuate agli analizzatori 48
3.3 Input per gli analizzatori 49
3.4 Processare output degli analizzatori 49
3.5 Unificazione dei singoli outputs e calcolo metriche secondarie . 50
3.5.1 Metriche secondarie 50

4 Porting in Rust e Confronto 53
4.1 Porting delle metriche 53
4.1.1 I problemi riscontrati 54
4.1.2 Rust Code Analysis 55
4.2 Confrontare i codici 55
4.2.1 Paragonare codici equivalenti 56
4.2.2 Confronto tramite script 57

5 Conclusioni e Lavori futuri 59
5.1 Conclusioni 59
5.1.1 SLR 59
5.1.2 L'aggregatore 61
5.1.3 Porting delle metriche in Rust 63
5.1.4 Come confrontare le metriche 63
5.2 Lavori futuri 63

Bibliografia 65

Appendix 73

Elenco delle tabelle

2.1 Pilot Studies 7
2.2 Stringhe di Ricerca 7
2.3 Numero di articoli trovati dalle librerie digitali selezionate . . 9
2.4 Metriche trovate negli studi primari, aventi numero di citazioni e punteggio almeno uguale a 2. 15
2.5 Metriche e suites con numero di citazioni e punteggi sopra il valore medio 17
2.6 Tutti i programmi trovati negli articoli scientifici 24
2.7 Programmi Proprietari - Linguaggi 30
2.8 Programmi O.S. - Linguaggi 31
2.9 Programmi Proprietari - Metriche 33
2.10 Programmi O.S. - Metriche 34
2.11 Programmi disponibili per calcolare le metriche più comuni per i linguaggi più comuni (in grassetto i tools open-source) . 36
2.12 Set ottimale di tools per i linguaggi di programmazione più supportati 38
2.13 Set ottimale di tools open-source per i linguaggi di programmazione più supportati 38
5.1 Studi Selezionati 73
5.2 Metriche - Studi (Completa) 76

Elenco delle figure

2.1 Distribuzione degli studi per anno, dal 2000 al 2019 12
2.2 Numero di autori per nazionalità di afferenza 12
2.3 Distribuzioni dei numeri totali di citazioni e dei punteggi delle metriche 14
2.4 Numero di programmi (proprietari ed open-source) per linguaggio 32
2.5 Numero di programmi (proprietari ed open-source) per metrica 35
3.1 Struttura dell'Aggregatore 43
3.2 Json output structure 46

Capitolo 1

Introduzione

Oggigiorno la sicurezza e la robustezza del software sono diventati sempre più importanti, vista la pervasività dei programmi. Si stanno compiendo sforzi importanti per avere strumenti in grado di riconoscere gli errori di programmazione il prima possibile e con sempre maggior accuratezza, oltre che a ridurre la frequenza con cui questi accadono. Si stanno sviluppando linguaggi di programmazione che rendano impossibile al programmatore fare una serie di errori comuni, con controlli effettuati durante la compilazione.
Fra le migliori pratiche che si possono seguire per ridurre gli errori in fase di scrittura del codice, si trovano la *continuous integration*, i test con misurazione della copertura del codice e, a volte, l'obbligo a rispettare uno stile di programmazione, che porta a migliorare la leggibilità del codice scritto.

Tale è l'importanza dell'argomento che negli anni si sono sviluppati degli standard internazionali per valutare la qualità del software. Lo standard ISO/IEC 9126 *"Software engineering - Product quality"*, la cui prima versione risale al 1991, si basava su lavori precedentemente svolti da studi pubblicati negli anni '70. Dopo successive revisioni, nel 2011 si è rimpiazzato il vecchio standard e si è passati al nuovo: ISO/IEC 25010 *"Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models"*, che è tuttora valido.
Il nuovo standard riprende e migliora il lavoro svolto in quello precedente, andando anche a sviluppare alcuni aspetti che erano stati prima tralasciati, come la sicurezza e la compatibilità del software. Il modello di qualità del software è diviso in 8 caratteristiche (functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability e portability), ognuna avente delle sotto-caratteristiche.

È proprio su una di queste caratteristiche che verte questa tesi: la *Manutenibilità*.

1.1 Manutenibilità

Con manutenibilità del software si intende il grado di efficacia ed efficienza con cui un prodotto o un sistema può essere modificato dai manutentori preposti. Lo standard ISO 9126 definiva la manutenibilità del software come la facilità con cui un sistema o un componente software potesse essere modificato per correggere guasti, migliorare le prestazioni o altri attributi, oppure adattarsi ad un ambiente in evoluzione [48].

Un'applicazione progettata e scritta tenendo a mente la sua manutenibilità porterà ad avere un software successivamente più facile da gestire. Come vantaggi collaterali principali vi saranno una maggior velocità e facilità nell'adattamento del software a necessità future ed un risparmio di denaro [6], poiché contribuisce a mantenere basso il *technical debt*.

Come si può quindi intuire, la manutenibilità del codice è una caratteristica importante, da non sottovalutare per vari motivi. A tutt'oggi, però, stimare questa qualità non è affatto immediato. I motivi sono molteplici, i principali sono sicuramente la complessità dell'argomento ed una carenza di strumenti adeguati che automatizzino la valutazione del codice.

Negli anni sono state studiate, proposte e validate innumerevoli metriche, e set di metriche, per calcolare la manutenibilità del software. Infatti in letteratura si trovano studi risalenti a più di 40 anni fa. Tuttavia in tutti questi anni, nonostante il crescente interesse, nessuna metrica né strumento ha ottenuto un'ampia accettazione da parte del mondo accademico o da aziende del settore [26].

1.2 Il progetto "Code Clarity in Rust"

Mozilla[1], una delle più famose comunità di software libero, conosciuta soprattutto per il browser web "Firefox" e il client di posta elettronica "Thunderbird", nel 2010 ha iniziato a sviluppare un nuovo linguaggio di programmazione: *Rust*.

[1] https://www.mozilla.org

Rust[2] è un linguaggio di programmazione compilato, progettato tenendo a mente i criteri di sicurezza, concorrenza e parallelismo. Il software scritto in questo linguaggio tende ad essere molto veloce durante l'esecuzione, con velocità quasi paragonabili al C e al C++ ed utilizza la memoria disponibile in maniera efficiente, senza bisogno di un *garbage collector*. Un vantaggio rispetto a questi ultimi due linguaggi citati è che Rust è un linguaggio di alto livello, che mira a rendere più semplice la programmazione concorrente e che impedisce al programmatore di fare una certa serie di errori, con dei controlli effettuati al momento della compilazione. Non è possibile, ad esempio, avere dei *dangling pointers*, *null pointers* o *data races*.
Sebbene sia un linguaggio abbastanza nuovo e in forte sviluppo, sta guadagnando sempre maggiore popolarità ed apprezzamento, tanto da essere stato eletto negli ultimi 4 anni consecutivi come *linguaggio di programmazione più amato* nell'annuale sondaggio di Stack Overflow[3]

Avendo questo linguaggio di programmazione molto promettente, Mozilla ha voluto investigare sui suoi eventuali difetti, in modo da poterli infine correggere. Riconoscendo l'importanza della manutenibilità, e volendo mettere a paragone Rust con gli altri linguaggi sotto questo punto di vista, ha messo a punto un progetto col Politecnico di Torino, che ha portato a questa tesi. Il progetto è stato curato dal Professor Luca Ardito, del Politecnico di Torino, e da Luca Barbato, consulente esterno della Luminem S.r.l..

1.2.1 Fasi del progetto

Il progetto è stato diviso in 4 fasi principali e in questa tesi vengono trattate in modo particolare le prime 2, mentre la terza e la quarta fase sono ancora in via di sviluppo.

- Nella prima fase si è guardato nella letteratura quale fosse lo stato delle metriche atte a calcolare la manutenibilità del codice e quali softwares venissero utilizzati per calcolarle. Per essere sicuri di poter avere un'analisi accurata dello stato della letteratura, è stata svolta una Systematic Literature Review, che è stata riportata nel Capitolo 2.

[2] https://www.rust-lang.org

[3] https://insights.stackoverflow.com/survey/2019

- Nella seconda fase, svolta nel Capitolo 3, viene presentato il software che è stato progettato e sviluppato in risposta ai risultati ottenuti nella fase precedente. Non si è trovato, infatti, un programma che potesse calcolare tutte le metriche più discusse in letteratura. Utilizzare diversi softwares porta ad altre complicazioni, la più importante delle quali è la difficoltà nell'automatizzare lo svolgimento delle analisi sui vari codici da esaminare.
  L'aggregatore creato va a risolvere questi problemi, e definisce uno standard per gli output di questi analizzatori.

- Nella terza fase vi è la ricerca e creazione di analizzatori per i codici sorgenti scritti in Rust, che possano computare le metriche trovate nella fase iniziale e riportarle in modo conforme allo standard definito nel punto precedente.

- Come ultima fase del progetto si andrà a comparare la manutenibilità di codici sorgenti equivalenti. Nello specifico, si vuole confrontare il codice scritto in Rust con quello scritto in C / C++.

Capitolo 2

Systematic Literature Review (SLR)

2.1 Research Questions

Il lavoro descritto in questa parte della tesi ha come intento quello di rispondere alle seguenti domande di ricerca:

RQ1.1: *Quali sono, in letteratura, le metriche disponibili per valutare la manutenibilità del codice?*
Con questa domanda vogliamo determinare quali metriche sono presenti in letteratura e quanto sono comuni negli studi che parlano di manutenibilità del codice.

RQ1.2: *Quali, fra le metriche trovate, sono le più popolari in letteratura?*
Questa domanda mira a caratterizzare le diverse metriche ottenute dalle risposte alla RQ1.1, in base alla loro popolarità e utilizzo da parte dei ricercatori.

RQ2.1: *Quali programmi sono disponibili per eseguire una valutazione del codice?*
Da questa domanda di ricerca vogliamo ottenere una lista di programmi, sia open source che closed source, assieme alle metriche che questi possono calcolare.

RQ2.2: *Qual è la selezione ideale di programmi in grado di calcolare le metriche più popolari per i linguaggi di programmazione più supportati?*
Questa research question implica la misurazione della copertura fornita dall'insieme delle metriche più popolari per ogni linguaggio e la stesura del set ottimale di programmi che possono calcolare quelle metriche.

2.2 Strategie di ricerca

Le strategie di ricerca richiedono la selezione di fonti dalle quali attingere le risorse e l'identificazione dei termini di ricerca.

2.2.1 Librerie Digitali

Come librerie digitali sono state usate:

- ACM Digital Library;
- IEEE Xplore;
- Scopus;
- Web of Science.

2.2.2 Stringhe di ricerca

La formulazione delle stringhe di ricerca è fondamentale per la definizione delle strategie di ricerca della SLR. Secondo le linee guida definite da Kitchenham et al. [55] la prima operazione nel definire le stringhe di ricerca richiede una analisi delle principali parole chiave usate nelle domande di ricerca e una ricerca dei loro possibili sinonimi. In questa fase tutti i ricercatori coinvolti nella SLR hanno collaborativamente selezionato alcuni studi pilota relativi al dominio di ricerca. Gli studi scelti sono presentati nella Tabella 2.1. Questi articoli sono stati selezionati per verificare la bontà delle stringhe di ricerca: qualora questi articoli non fossero stati presenti nei risultati dopo la fase di raffinazione delle queries, si sarebbero dovute rivedere le stringhe scelte.

Le parole chiave identificate inizialmente erano: *software*, *maintainability* e *metrics*. Quindi è stata utilizzata la stringa "software maintainability metric" per eseguire la prima ricerca sulle librerie digitali scelte. Abbiamo incluso solo gli studi pubblicati fra il 2000 e il 2019.
Da queste prime ricerche ci si è resi conto che aggiungendo alla keyword

Tabella 2.1. Pilot Studies

ID	Authors	Title	Year
[14]	J. Ostberg, S. Wagner	On Automatically Collectable Metrics for Software Maintainability Evaluation.	2014
[11]	J. Ludwig, S. Xu, F. Webber	Compiling static software metrics for reliability and maintainability from GitHub repositories.	2017
[6]	A. Kaur, K. Kaur, K. Pathak	Software maintainability prediction by data mining of software code metrics.	2014
[26]	M. I. Sarwar, W. Tanveer, I. Sarwar, W. Mahmood	A comparative study of MI tools: Defining the Roadmap to MI tools standardization.	2008
[29]	H. Liu, X. Gong, L. Liao, B. Li	Evaluate How Cyclomatic Complexity changes in the context of software evolution.	2018

Tabella 2.2. Stringhe di Ricerca

Libreria	Query iniziale	Dopo il raffinamento
ACM D. L.	+("software" "code") +(metrics) +(maintainability)	+("software" "code") +(metrics) +(maintainability) -(defect) -(fault) -(co-change) -(policy-driven) -(design)
IEEE Xplore	(((code OR software) AND metrics) AND maintainability)	(((((((code OR software) AND metrics) AND maintainability) NOT defect) NOT fault) NOT co-change) NOT policy driven) NOT design)
Scopus	(code OR software) AND metrics AND maintainability	code OR software AND metrics AND maintainability AND NOT fault AND NOT defect AND NOT co-change AND NOT policy driven AND NOT design
Web of Science	(code OR software) AND metrics AND maintainability	code OR software AND metrics AND maintainability AND NOT fault AND NOT defect AND NOT co-change AND NOT policy driven AND NOT design

software il sinonimo *code* si ottenevano un numero maggiore di articoli nei risultati.
Sono poi stati aggiunti dei termini da escludere alle stringhe di ricerca, in quanto utili a ridurre il numero di articoli non pertinenti. Fra le keywords si trovano:

- *defect* e *fault*, per evitare gli articoli incentrati sui temi della validazione, verifica del software e tendenza agli errori, più che sulla manutenibilità;

- *co-change*, per evitare studi incentrati sull'evoluzione del codice;
- *policy-driven* e *design*, per evitare di considerare gli articoli che si concentrano sulle metriche usate per progettare il software.

La lista delle search queries utilizzate per ognuna delle librerie digitali scelte sono riportate in Tabella 2.2, sia nelle versioni iniziali, prima del raffinamento, che nelle versioni finali.

2.2.3 Criteri di Inclusione ed Esclusione

La fase finale della selezione degli studi si è compiuta seguendo i criteri di inclusione ed esclusione elencati di seguito. Lo scopo di questi criteri è quello di selezionare solo gli studi inerenti all'argomento voluto e di scartare gli altri.

Criteri di inclusione:

IC1: Studi scritti in una lingua comprensibile dagli autori.

IC2: Studi che descrivano accuratamente una nuova metrica.

IC3: Studi che presentino, analizzino o comparino metriche conosciute o programmi.

IC4: Studi primari dettagliati.

Criteri di esclusione:

EC1: Studi scritti in una lingua non direttamente comprensibile dagli autori, ovvero non scritto in inglese, italiano, portoghese o spagnolo.

EC2: Studi che presentino una nuova metrica, ma non la descrivano accuratamente.

EC3: Studi che non descrivano né usino metriche o programmi per calcolarle.

EC4: Studi secondari (es.: systematic literature reviews, surveys, mappings).

2.3 Ricerca degli Studi

Questa fase consiste nella raccolta di tutti gli studi trovati applicando alle librerie digitali selezionate le stringhe di ricerca formulate e discusse nella sottosezione 2.2.2, sfruttando anche il tool Publish or Perish [2]. Dopo

l'applicazione delle *queries*, i risultati delle diverse librerie digitali sono stati riuniti in un'unica lista. I risultati duplicati sono stati rimossi e, alla fine, sono stati raccolti un totale di 801 articoli. Si possono vedere i numeri in Tabella 2.3. Per semplificare e velocizzare l'analisi dei singoli articoli, le in-

Tabella 2.3. Numero di articoli trovati dalle librerie digitali selezionate

Libreria	**Con query iniziali**	**Post raffinamento**	**Senza duplicati**
ACM D. L.	497	152	-
IEEE Xplore	443	215	-
Scopus	848	381	-
Web of Science	599	300	-
Totali	2387	1048	801

formazioni principali di tutti i papers sono state aggregate in un unico file CSV.

Il risultato di questa fase è una lista di possibili studi, che sono poi stati controllati seguendo i criteri di inclusione ed esclusione. Gli articoli che hanno passato questa ultima selezione sono diventati gli studi primari per il nostro studio.

2.4 Selezione degli Studi

Il processo della selezione degli studi è stata affrontata da ogni autore di questa SLR in maniera indipendente dagli altri. Per analizzare gli articoli, al posto di dividerli da subito in due categorie (adatti e non adatti), si è deciso di utilizzare una scala Likert a 5 punti, seguendo i seguenti criteri:

- Un punto agli studi che rispettavano i criteri esclusione, ma nessuno dei criteri di inclusione;
- Due punti agli articoli che soddisfacevano alcuni criteri di esclusione ed alcuni criteri di inclusione;
- Tre punti ai papers che non rispettavano nessuno dei criteri, oppure che erano in dubbio;
- Quattro punti agli studi che rispettavano la maggior parte dei criteri di inclusione, ma non tutti;
- Cinque punti a quelli che soddisfacevano tutti i criteri di inclusione.

Gli studi sono stati analizzati in due passaggi: dapprima basandosi sul titolo e sull'abstract, per avere un riscontro immediato dell'articolo con i criteri di inclusione ed esclusione; successivamente gli articoli che hanno ricevuto 3 punti sono stati letti per intero e rivalutati. Per questo motivo sono poi stati presi in considerazione solo gli articoli valutati almeno 4 punti.

Durante questa fase abbiamo anche applicato il processo di *snowballing*, che in questa specifica SLR non ha portato all'aggiunta di altri studi. Con il processo di snowballing [1] si intende l'utilizzo delle liste di articoli citati dagli studi precedentemente individuati per ottenere articoli addizionali a quelli acquisiti con le domande di ricerca.

2.5 Estrazione dei Dati

In questa fase sono stati letti nuovamente gli studi primari, per raccogliere i dati utili a rispondere alle domande di ricerca formulate precedentemente (Sezione 2.1). È stato creato un foglio di calcolo con un modulo da compilare per ogni articolo da analizzare, con lo scopo di ottenere metodicamente, per ogni domanda di ricerca, i dati di interesse. L'estrazione dei dati è stata svolta, anche questa, da tutti gli autori, in modo indipendente.

Per ogni articolo si sono raccolte alcune informazioni di contesto:

- Anno di pubblicazione;
- Numero di volte in cui lo studio è stato visionato per intero;
- Numero di citazioni;
- Autori e ubicazione degli autori.

Per rispondere alla RQ1.1 è stato ispezionato il set degli studi primari per capire quali metriche essi definissero o menzionassero. Quindi, per ogni articolo, sono stati estratti i seguenti dati:

- Le *liste* di *metriche* e di *set di metriche* utilizzati per ogni paper;
- I *linguaggi di programmazione* e le famiglie di linguaggi per i quali le metriche proposte o utilizzate possano essere calcolate;

Per rispondere alla RQ1.2 si è data una classificazione addizionale delle metriche, oltre al numero di citazioni: abbiamo classificato ogni metrica più o meno positivamente analizzando le opinioni degli autori dei vari studi.

Abbiamo così potuto prendere in considerazione la popolarità delle metriche, e dei set di metriche, contando la differenza fra le citazioni positive e negative date dagli autori.

Per rispondere alla RQ2.1 abbiamo dovuto esaminare gli studi primari per capire quali programmi fossero presentati o usati per calcolare le metriche che erano state adottate. Per ogni articolo che menzionava dei tools sono state quindi prese queste informazioni:

- La *lista dei programmi* descritti, utilizzati o citati;
- Ove possibile, la *lista di metriche* che possono essere calcolate da ogni programma;
- La *lista dei linguaggi di programmazione* supportati da ogni tool;
- Alcune *informazioni generiche* sui programmi usati, come il fatto che fossero open-source o meno.

Per rispondere alla RQ2.2 invece non serviva recuperare dei nuovi dati, ma si dovevano rielaborare quelli già presi per rispondere alle domande di ricerca precedenti.

2.6 Sintesi e resoconto dei dati

In questa fase i dati precedentemente estratti sono stati elaborati in modo da ottenere una risposta per ognuna delle domande di ricerca. Avendo tutti i dati necessari sotto forma di un modulo per ogni articolo analizzato abbiamo potuto procedere con la sintesi dei dati.

Tutte le metriche e i set di metriche trovati sono stati raccolti in una tabella, tenendo traccia degli studi che li menzionavano, e sono state calcolate delle misure aggregate sulla popolarità di ogni metrica.

2.7 Risultati

Questa sezione descrive i risultati ottenuti per rispondere alle domande di ricerca descritte nella sottosezione 2.2.2. Le tabelle complete, con tutti i dati estratti, sono state riportate in appendice, mentre di seguito sono state riportati solo i risultati più significativi, per migliorare la leggibilità e navigabilità del testo.

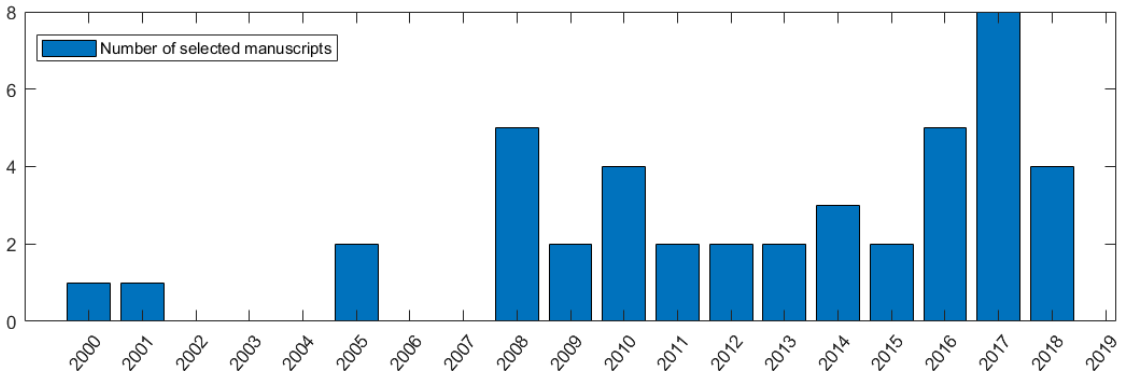

Figura 2.1. Distribuzione degli studi per anno, dal 2000 al 2019

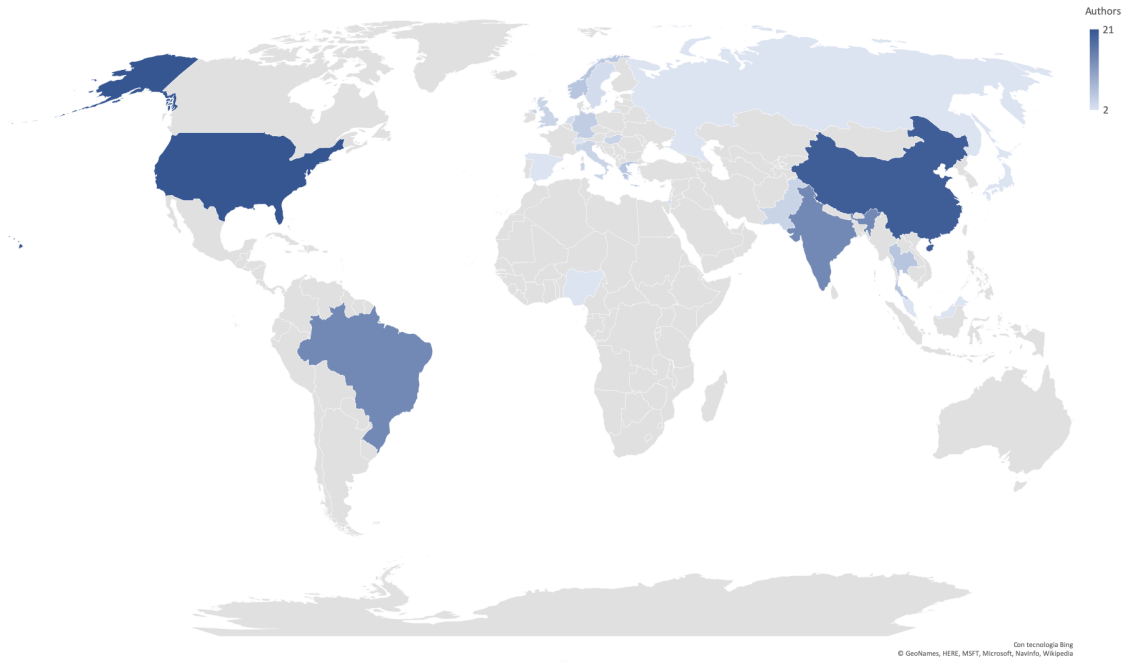

Figura 2.2. Numero di autori per nazionalità di afferenza

Alla fine di questa fase abbiamo collezionato una lista di 43 studi primari per la fase successiva della SLR. La Figura 2.1 mostra la distribuzione degli articoli selezionati nel lasso di tempo considerato, dal primo all'ultimo anno di pubblicazione degli studi selezionati, che sono riportati nella Tabella 5.1, che è riportata in appendice.

Dalla figura emerge che, col passare del tempo, sembra esserci un progressivo aumento di interesse verso le metriche di manutenibilità del codice, visto il numero di studi effettuati negli ultimi anni.

2.7.1 RQ1: le Metriche

Dagli articoli selezionati come studi primari per la SLR abbiamo trovato un totale di 174 diverse metriche, che abbiamo riportato in appendice nella Tabella 5.2. La tabella riporta:

- il *nome della suite* di cui fa parte la metrica considerata (vuoto se la metrica non fa parte di un particolare set);
- il *nome della metrica* e, se esiste, un suo acronimo;
- la *lista di articoli* che menzionano la metrica;
- il *numero degli studi* che menzionano la metrica (ovvero il numero di studi nella terza colonna);
- il *punteggio* che abbiamo dato a ogni metrica.

Per calcolare il punteggio dato a ogni metrica abbiamo assegnato:

- +1 se lo studio semplicemente usava o definiva la metrica, oppure se gli autori dell'articolo hanno espresso un parere positivo sulla stessa;
- -1 se lo studio criticava la metrica.

Esaminando le ultime due colonne della tabella delle metriche si può notare subito che la maggioranza delle volte, per ogni metrica, i due valori sono identici. Questo perché la maggior parte degli studi che abbiamo trovato utilizza le metriche senza commentarle.

È subito evidente come alcune *suites* di metriche, e metriche singole, siano prese in considerazione molto più di frequente rispetto ad altre. Più del 75% delle metriche sono citate da soltanto uno degli articoli che abbiamo selezionato. I boxplots in Figura 2.3 mostrano le distribuzioni del *numero di citazioni* (Mentions) e dei *punteggi* (Scores); i boxplot in rosso sono quelli che prendono in considerazione tutte le 174 metriche trovate, mentre quelli in blu si riferiscono solo alle metriche che hanno ricevuto almeno due citazioni. È evidente come nei boxplot che si riferiscono a tutte le metriche (quelli rossi) la differenza fra le due distribuzioni sia piuttosto limitata, il che è dato

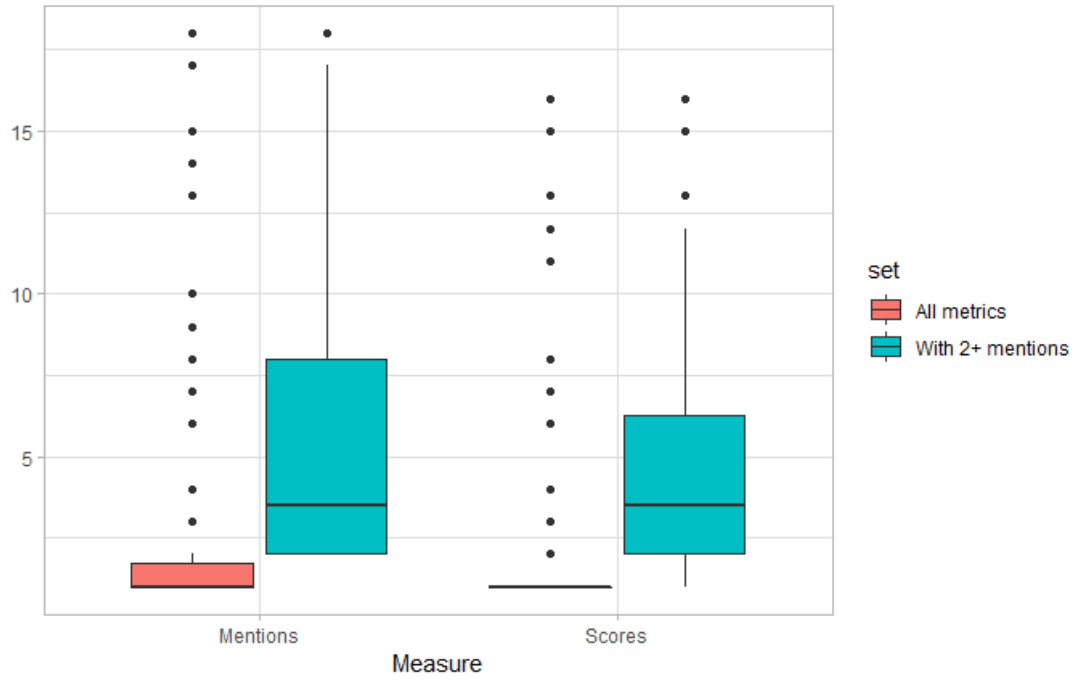

Figura 2.3. Distribuzioni dei numeri totali di citazioni e dei punteggi delle metriche

dalla grande maggioranza di opinioni positive o neutrali sulle metriche, negli articoli che le citano.
Poiché solo il 24.7% delle metriche sono utilizzate da più di uno degli articoli da noi selezionati, quando si prende in considerazione l'intera lista di metriche in entrambi i casi il valore medio è 1.

In generale, comunque, è bene sottolineare come un *punteggio* basso non necessariamente significhi che la metrica considerata sia di bassa qualità, ma che potrebbe semplicemente essere meno conosciuta nella letteratura e, in questo caso specifico, negli studi da noi considerati. Un'altra cosa degna di nota è che non abbiamo trovato una particolare metrica che abbia ricevuto un numero considerevole di opinioni negative.

Le metriche più citate

Visto che la nostra analisi è finalizzata al trovare le metriche più popolari, per poter poi estrarre un set di metriche da utilizzare con diversi linguaggi, si era interessati a trovare le metriche citate da più articoli.

Tabella 2.4: Metriche trovate negli studi primari, aventi numero di citazioni e punteggio almeno uguale a 2.

Inizio della Tabella:Metriche - Studi (Riass)

Suite	Metrica	Studi che la usano	Cit.	Score
-	LOC, Lines of Code	[5], [8], [9], [11], [14], [15], [17], [18], [19], [25], [26], [43], [44], [42]	14	10
-	NOM, No. of Methods	[5], [12], [34], [43]	4	4
-	CHANGE, No. of Lines Changed in the Class	[5], [6], [8], [12]	4	4
-	STAT, No. of Statements	[6], [43], [8], [42]	4	4
-	No. of Query	[6], [8]	2	2
Chidamber & Kemerer	WMC, Weighted Methods per Class	[5], [6], [12], [17], [19], [20], [25], [29], [33], [35], [39], [44], [47]	13	11
Chidamber & Kemerer	DIT, Depth of Inheritance Tree	[5], [6], [12], [14], [17], [19], [20], [25], [33], [34], [35], [39], [44], [47], [42]	15	13
Chidamber & Kemerer	NOC, Number of Children	[5], [6], [12], [14], [17], [20], [25], [33], [35], [39], [44], [47], [42]	13	11
Chidamber & Kemerer	CBO,Coupling Between Objects	[5], [6], [8], [11], [14], [17], [19], [20], [25], [33], [34], [35], [36], [39], [44], [47], [42]	17	15
Chidamber & Kemerer	RFC, Response for Class	[5], [6], [10], [11], [12], [14], [17], [19], [20], [25], [33], [35], [39], [41], [44], [47]	16	14
Chidamber & Kemerer	LCOM, Lack of Cohesion in Methods	[5], [6], [12], [14], [17], [20], [25], [33], [34], [35], [36], [41], [44], [47]	14	12
Chidamber & Kemerer	LCOM2, Lack of Cohesion in Methods	[19], [39], [42]	3	3
Chidamber & Kemerer	NPM, Number of Public Methods	[6], [19], [34], [39]	4	4
Li & Henry (L&H)	MPC, Message Passing Coupling	[8], [12], [35], [41]	4	4
Li & Henry (L&H)	DAC	[12], [35]	2	2
Halstead	Halstead Vocabulary (n)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
Halstead	Halstead Volume (V)	[6], [8], [14], [15], [17], [18], [21], [26], [44], [47]	10	8
Halstead	Halstead Bugs (B)	[6], [8], [14], [21], [44], [47]	6	4
Halstead	Halstead Length (N)	[6], [8], [9], [14], [15], [18], [21], [44], [47]	9	7
Halstead	Halstead Difficulty (D)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
Halstead	Halstead Effort (E)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
-	Ca, Afferent Coupling	[6], [19]	2	2
-	Ce, Efferent Coupling	[6], [8], [19]	3	3
-	Number of Commands	[6], [8]	2	2
-	CLOC, Comments Lines of Code	[6], [8], [9], [17], [34], [42]	6	6
-	JLOC, JavaDoc lines of Code	[6], [8], [34]	3	3
-	NOAC, Number of Operations added	[6], [8]	2	2
-	PDcy, Number of package dependencies	[6], [8]	2	2
-	CONS, Number of Constructors	[6], [8]	3	2
-	CSOA, Class Size (Operations +Attributes)	[6], [8]	2	2
-	CSA, Class Size(Attributes)	[6], [8]	2	2

Continuazione della Tabella: Metriche - Studi (Riass)

Suite	Metrica	Studi che la usano	Cit.	Score
-	CSO, Class Size (operations)	[6], [8]	2	2
-	Avg cc, Average cyclomatic complexity	[6]	2	2
-	CC, McCabe's Cyclomatic Complexity	[9], [14], [15], [17], [18], [21], [25], [26], [29], [39], [43], [44] [47], [42]	14	12
-	WMC, McCabe's Weighted Method Count	[8], [11], [14], [16], [34], [36]	6	6
-	Code to Comment Ratio	[11], [14]	2	2
-	MI, Maintainability Index	[8], [14], [15], [17], [26]	5	3
Structural Measures	TCC, Tight class Cohesion	[17], [42]	2	2
-	Number of classes (including nested classes, interfaces, enums and annotations)	[43], [42]	2	2
-	Number of files	[43], [45]	2	2
-	MOA, Measure Of Aggregation	[19], [34]	2	2
-	DAM, Data Access Metric (Card Metric)	[19], [47]	2	2
-	LCOM3, Lack of Cohesion of Methods	[19], [42]	2	2

Fine della Tabella: Metriche - Studi (Riass)

Nella Tabella 2.4 abbiamo riportato solo le metriche che sono state utilizzate da almeno due studi fra quelli da noi selezionati. Questo ci permette di ridurre il rumore causato da tutte le metriche citate solo una volta (spesso nell'articolo dove era stata definita in origine). Dopo aver applicato questo filtro, sono rimaste solo 43 metriche (il 24.7% delle 174 originali). Guardando i boxplot azzurri in Figura 2.3, che si riferiscono a questa lista di metriche, si può notare come in questo caso il valore medio per il numero di citazioni sia 3, così come per il punteggio.

Siccome lo scopo finale per rispondere alla RQ1.2 è quello di trovare il set delle metriche più popolari per stimare la manutenibilità del codice, abbiamo selezionato, fra le metriche trovate, quelle aventi un punteggio uguale o maggiore al valore medio, ovvero sia 3.

Con questa ulteriore condizione abbiamo ottenuto un set di 13 *metriche* e 2 *suite di metriche*, che abbiamo riportato nella Tabella 2.5. Due suites sono state incluse per intero (Chidamber & Kemerer e Halstead) poiché tutte le loro metriche hanno ricevuto un numero di citazioni e un punteggio superiori o uguali al valore medio trovato. Per loro, la tabella riporta il numero di citazioni e il punteggio minimi fra quelli ottenuti dalle metriche che compongono la suite.
Per la suite di Li & Henry, invece, è stata presa in considerazione solo la metrica MPC (Message Passing Coupling), in quanto era l'unica ad aver ottenuto dei valori superiori ai valori medi.

Tabella 2.5. Metriche e suites con numero di citazioni e punteggi sopra il valore medio

Metrica	Citazioni totali	Punteggio
CC - McCabe's Cyclomatic Complexity	14	12
Ce - Efferent Coupling	3	3
CHANGE - Number of Lines Changed in Class	4	4
C&K - Chidamber & Kemerer Suite	13+	11+
CLOC - Comments Lines of Code	6	6
Halstead's Suite	6+	4+
JLOC - JavaDoc Lines of Code	3	3
LOC - Lines of Code	14	11
LCOM2 - Lack of Cohesion in Methods	3	3
MI - Maintainability Index	6	4
MPC - Message Passing Coupling	4	4
NOM - Number of Methods	4	4
NPM - Number of Public Methods	4	4
STAT - Number of Statements	4	4
WMC - McCabe's Weighted Method Count	7	7

Di seguito abbiamo riportato, in ordine alfabetico, una breve descrizione delle metriche selezionate.

- **CC - McCabe's Cyclomatic Complexity**, - Complessità Ciclomatica di McCabe - messa a punto da T. J. McCabe nel 1976 [49] è una metrica intesa per calcolare la complessità del codice esaminando il grafico del flusso di controllo di un programma; ovvero contando i suoi percorsi di esecuzione indipendenti [14]. L'assunzione è che la complessità del codice è correlata al numero di percorsi di esecuzione del suo diagramma di flusso.
  È stato provato da Jay et al. [58] che esiste una correlazione lineare fra la CC e le metriche di tipo LOC (introdotte poco sotto); inoltre, questa correlazione è indipendente dai linguaggi di programmazione scelti e dai paradigmi del codice.

  Ogni *nodo* nel diagramma di flusso corrisponde a un blocco di codice nel programma, dove il flusso è sequenziale. Gli *archi* corrispondono ai vari rami che possono essere presi dal flusso di controllo durante l'esecuzione del programma. Basandosi su questi archi e nodi, la CC di un codice sorgente è definita come $M = e - n + 2p$, dove:

  n è il numero di *nodi* del grafo,

  e è il numero di *archi* del grafo,

p è il numero di componenti connessi, ovvero il numero di uscite dalla logica del programma [26].

- **CE - Efferent Coupling.** Questa metrica misura quanti tipi di dato vengono utilizzati dalla classe analizzata (a parte la classe stessa). La metrica prende in considerazione l'eredità dei tipi conosciuti, le interfacce implementate dalla classe, i tipi dei parametri dei suoi metodi, i tipi degli attributi dichiarati e i tipi delle eccezioni utilizzate.
- **CHANGE - Number of Lines Changed in the Class**, numero di linee cambiate in una classe. È una metrica che misura il cambiamento di un codice, infatti misura quante linee di codice sono cambiate fra due versioni della stessa classe. Questa metrica, quindi, è differente dalle altre che abbiamo elencato qui perché non si riferisce ad una singola versione del codice, bensì è fatta per analizzare l'*evoluzione* del codice sorgente. Se una classe è continuamente modificata, può essere un segno che essa è dispendiosa da manutenere.
  Generalmente, vi sono tre tipi di modifiche che possono essere effettuate ad una linea di codice: aggiunte, cancellazioni o modifiche. In letteratura, di solito, vi è un'uniformità su come contare le operazioni di modifica, che vengono trattate come se fossero una cancellazione, seguita da una aggiunta di codice (andando, di fatto, a valere il doppio di una cancellazione o di una aggiunta). La maggior parte delle volte i commenti e le linee vuote non sono considerate.
- **C&K - Chidamber and Kemerer's suite.** Introdotta nel 1994, è fra i set di metriche più conosciuti [50]. Questa suite è stata progettata tenendo a mente un approccio orientato ad oggetti. È composta da 6 metriche, elencate di seguito:
  - *WMC, Weighted Method per Class*, definita allo stesso modo di Weighted Method Count (WMC) di McCabe, ma applicato a una classe. Si ottiene calcolando e sommando assieme i valori delle CC di tutte le funzioni della classe considerata [50].
  - *DIT, Depth of Inheritance Tree*, definito come la lunghezza del percorso più lungo fra il nodo foglia e il nodo radice dell'albero dell'ereditarietà delle classi del software analizzato.
    L'ereditarietà aiuta la riusabilità del codice, ovvero aumenta la manutenibilità, però è altresì vero che più la gerarchia delle classi aumenta, più il comportamento del codice diventa sempre più complesso, rendendo il software sempre più difficile da mantenere.

Avere uno, due o tre livelli di ereditarietà può aiutare la manutenibilità del codice, ma aumentarli ulteriormente è ritenuto controproducente.

- *NOC, Number of Children*, è il numero di sottocolassi dirette della classe analizzata. All'aumentare del numero, aumenta la manutenibilità.
- *CBO, Coupling Between Objects*, è il numero di classi con le quali la classe analizzata è collegata. Due classi sono considerate collegate quando i metodi dichiarati in una classe usano metodi o variabili di istanza definiti dall'altra classe.
  Quindi questa metrica ci dà un'idea di quanto siano interlacciate le varie classi fra di loro, ovvero di quanto il mantenimento di una classe influisca sulle altre.
- *RFC, Response for Class*, è definita come un set di metodi che può essere potenzialmente eseguito in risposta ad un messaggio ricevuto da un oggetto di quella classe. In questo caso, maggiore è il valore ritornato da questa metrica, maggiore è la complessità della classe.
- *LCOM, Lack of Cohesion in Methods*, è definita come la sottrazione fra il numero di coppie di metodi non aventi attributi in comune e il numero di coppie di metodi aventi attributi in comune. Un valore alto di questa metrica fornisce una misura della relativa differenza fra i metodi nella classe.
  In letteratura vi sono diverse versioni di questa metrica.

- **CLOC - Comments Line of Code** La metrica dà il numero di linee di codice contenenti commenti. Non vengono contate le linee di commenti vuote. In contrasto con la metrica LOC, qui più il valore ritornato è alto, maggiori sono i commenti nel codice analizzato, quindi il codice dovrebbe essere più leggibile e mantenibile.
  In letteratura è stata anche proposta una metrica che mette in relazione CLOC e LOC, chiamata *Code to Comment Ratio*.

- **Halstead's suite.** Introdotta nel 1977 [52], è un set di metriche che tenta di calcolare la quantità di lavoro richiesto per mantenere il codice in analisi, la sua qualità e il numero di errori in quella implementazione. Per calcolare le metriche di questa suite bisogna prima conteggiare i seguenti indicatori:

  n_1 **:** il numero di *operatori* distinti

n_2 **:** il numero di *operandi* distinti

N_1 **:** il numero totale di *operatori*

N_2 **:** il numero totale di *operandi*

Nel codice, gli *operandi* sono gli oggetti che sono manipolati; gli *operatori* sono tutti i simboli che rappresentano azioni specifiche.

Gli operatori e gli operandi sono i due tipi di componenti che formano tutte le metriche di questa suite, che sono:

- *Length (N):* $N = N_1 + N_2$
  È la somma di operatori e operandi nel codice analizzato.
- *Vocabulary (n):* $n = n_1 + n_2$
  È la somma del numero totale di operatori e operandi nel codice.
- *Volume (V):* $V = N \cdot \log_2 n$
  Rappresenta la dimensione, in bits, dello spazio su disco usato dal programma per archiviarlo.Varia in base alla specifica implementazione.
- *Difficulty (D):* $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$
  Dà una stima di quanto sia difficile comprendere il codice.
- *Effort (E):* $E = D \cdot V$
  Tenta di quantificare lo sforzo necessario per comprendere una classe.
- *Bugs (B):* $B = \frac{E^{\frac{2}{3}}}{3000}$
  Prova a dare una stima del numero di errori presenti durante l'implementazione del codice.
- *Time to implement (T):* $T = \frac{E}{18}$
  Tenta di stimare la quantità di tempo necessario per implementare quel codice.

- **JLOC, JavaDoc Lines of Code.** È una metrica specifica per Java, definita come il numero di linee di codice contenenti commenti JavaDoc. È simile alle altre metriche che misurano il numero di commenti nel codice sorgente. In generale, un valore alto per questa metrica è ritenuto positivo, poiché suggerisce una migliore documentazione del codice, che porta a una manutenibilità maggiore. Come precedentemente detto, questa è una metrica specifica per un certo linguaggio di programmazione, il Java, ma esistono generatori di documentazione simili in altri

linguaggi, come JSDoc e PHPDocumentor. Quindi si potrebbero creare delle versioni similari di questa metrica per tutti quei linguaggi di programmazione che hanno questi generatori di documentazione.

- **LOC, Lines of Code.** Una metrica largamente usata, spesso per via della sua semplicità. Dà una misura immediata della dimensione del codice sorgente. Fra le metriche più popolari, questa è stata l'unica avente due menzioni negative negli articoli da noi selezionati. Questi pareri negativi provengono dal fatto che non esiste una definizione unica di come la metrica LOC debba essere computata [14]: in alcuni lavori vengono contate tutte le linee in un file (non molto comune), ma la maggioranza delle volte vengono contate solo le linee non vuote. Non viene nemmeno specificato come contare una linea se questa contiene più di una istruzione: si può sia contare come singola linea (si conteggiano le linee fisiche), oppure si può conteggiare il numero di istruzioni che la compongono (si parla quindi di conteggio di linee logiche).
  Si può quindi vedere come sia fondamentale che i programmi usati per computare le metriche specifichino esattamente come calcolano i valori che ritornano. Nel caso di tools open-source queste informazioni si possono sia derivare che verificare direttamente dal codice, dando una certezza in più all'utilizzatore.
  Sebbene LOC sembri essere poco correlata con lo sforzo necessario per il mantenimento del codice [14] e ci siano più modi per calcolarla, questa metrica è utilizzata da un'altra metrica, il Maintainability Index, e sembra essere correlata con molte altre [56]. Maggiore è il valore ritornato da questa metrica, meno mantenibile è il codice analizzato.

- **LCOM2, Lack of Cohesion in Methods.** È una evoluzione della metrica LCOM, che è parte della suite di C&K. Equivale alla percentuale di metodi che non fanno accesso a uno specifico attributo, mediato su tutti gli attributi della classe. Se il numero di metodi o attributi è zero, LCOM2 viene posta a zero. Un valore di LCOM2 o LCOM3 basso indica un'alta coesione della classe.

- **MI, Maintainability Index.** È una metrica composita, che mira a dare una stima sulla manutenibilità di un software. Ci sono diverse definizioni di questa metrica, che è stata introdotta per la prima volta da Oman e Hagemeister nel 1992 [53]. Ci sono due diverse formule per calcolare l'indice di mantenibilità: una utilizza solo tre metriche (e per questo motivo chiamata *3-metrics equation*), il Volume di Halstead (V),

la Complessità Ciclomatica (CC) e la LOC; l'altra (*4-metrics equation*) utilizza anche il numero di commenti (CLOC).
Nonostante sia abbastanza popolare come metrica, Ostberg e Wagner hanno espresso dei dubbi riguardo la sua adeguatezza, asserendo che, siccome il M.I. si basa su metriche non idonee a calcolare la manutenibilità del codice, non si può ottenere una indicazione utile nemmeno da quest'ultima. In più, contestano il fatto che il risultato della metrica stessa sia poco intuitiva [14].
Di differente opinione sono Sarwar et al. che ritengono questa metrica piuttosto utile per migliorare sia la manutenibilità del software che per abbassare il costo della sua manutenzione [26].
3-metrics equation:

$$\mathbf{MI} = 171 - 5.2 \cdot \ln(\overline{H_V}) - 0.23 \cdot \overline{CC} - 16.2 \cdot \ln(\overline{LOC})$$

4-metrics equation:

$$\mathbf{MI} = 171 - 5.2 \cdot \ln(\overline{H_V}) - 0.23 \cdot \overline{CC} - 16.2 \cdot \ln(\overline{LOC}) + 50 \cdot \sin(\sqrt{2.4 \cdot \overline{Com_{\%}}})$$

Essendo in entrambe le equazioni:

$\overline{H_V}$**:** la media dei Volumi di Hastead,
$\overline{LOC}$**:** la media dei valori LOC,
$\overline{CC}$**:** la media dei valori delle CC,
$\overline{Com_{\%}}$**:** la media delle percentuali di CLOC.

Per questa metrica un valore di ritorno sopra l'85 indica che il codice è facilmente manutenibile; un valore fra l'85 e il 65 denota un codice non così semplice da mantenere; sotto il 65 il codice risulta difficile da mantenere. Il valore ritornato può essere anche negativo, in modo particolare per progetti di grandi dimensioni.

- **MPC, Message Passing Coupling.** Fa parte della suite di Li & Henry ed è l'unica metrica di quel set ad aver ricevuto un punteggio sufficiente per essere considerata in questo elenco. È definita come il numero di chiamate di metodo definite nei metodi di una classe a metodi di un'altra classe [51].
- **NOM, Number of Methods.** Ritorna il numero di metodi in una data classe o file contenente codice sorgente. Maggiore è il numero di metriche, minore sarà la manutenibilità di quella parte di codice.

- **NPM, Number of Public Methods.** Ritorna il numero di tutti i metodi pubblici di una classe.
- **STAT, Number of Statements** o numero di espressioni in un metodo. Bisogna dire se si contano le espressioni nelle interfacce, nelle classi interne anonime, e in quelle interne con nome. Kaue et al. nel loro studio contano solo il numero di espressioni presenti nelle classi interne anonime [6].
- **WMC, McCabe's Weighted Method Count.** È una misura di complessità che somma le complessità di tutti i metodi implementati nel codice analizzato. La complessità di ogni metodo è calcolata utilizzando la Complessità Ciclomatica di McCabe.
  Una variante semplificata di questa metrica, chiamata WMC-Unweighted, conta ogni metodo come se avesse complessità unitaria (non pesata, appunto), ritornando, di fatto, il numero di metodi presenti nel codice.

2.7.2 RQ2: I Programmi

Le domande di ricerca 2.1 e 2.2 richiedono di raccogliere informazioni su tutti i programmi, detti di seguito anche *tools*, che sono stati utilizzati negli studi primari ricavati nelle fasi precedenti della nostra SLR. Nella Tabella 2.6 abbiamo riportato tutti i programmi che sono stati identificati leggendo gli articoli. Le colonne riportano rispettivamente: il nome del tool, come presentato negli studi; gli studi che lo hanno utilizzato; una fonte web da dove il programma può essere scaricato. Nella prima parte della tabella abbiamo citato i programmi che non si riescono a trovare in rete e gli studi che menzionavano dei tools senza però dare dei riferimenti per trovarli. Per questi ultimi abbiamo indicato i programmi con i nomi degli autori.
Nella seconda e terza sezione abbiamo diviso i programmi in base al fatto che fossero tools commerciali oppure open-source.
Nella tabella vi sono riportati un totale di 38 programmi, dei quali: 19 non sono stati trovati, 6 sono closed-source e 13 open-source.

La maggioranza dei programmi che abbiamo trovato sono menzionati in soltanto uno degli studi selezionati; tre sono citati da due studi e solo uno, CKJM, è citato da cinque articoli.

Si può subito vedere come i tools open-source siano più del doppio, in numero, rispetto a quelli closed-source. Questo risultato non è certamente correlato con la qualità dei vari programmi, ma è giustificato dal fatto che i programmi rilasciati sotto una licenza open-source siano più adatti

Tabella 2.6. Tutti i programmi trovati negli articoli scientifici

Nome del programma	**Studi**	**Dove trovarlo**
Columbus Quality Model	[11]	Non trovato
Analyst4j	[26]	Non trovato
Lachnesis	[26]	Non trovato
Metrics	[26]	Non trovato
CCEvaluator	[29]	Non trovato
Lagrein	[37]	Non trovato
Code Crawler	[37]	Non è chiaro quale programma abbiano usato
RAMOOS - Reconfigurable Automated Metrics for Object-Oriented Software	[41]	Non trovato
Baker (Baker 1993)	[45], [46]	Non è chiaro quale programma abbiano usato
Kamiya (Kamiya et al. 2002)	[45]	Non è chiaro quale programma abbiano usato
Li (Li and Thompson 2010)	[45]	Non è chiaro quale programma abbiano usato
Baxter (Baxter et al. 1998)	[45]	Non è chiaro quale programma abbiano usato
Brown (Brown and Thompson 2010)	[45]	Non è chiaro quale programma abbiano usato
Koschke (Koschke et al. 2006)	[45]	Non è chiaro quale programma abbiano usato
Higo (Higo and Kusumoto 2009)	[45]	Non è chiaro quale programma abbiano usato
Mayrand (Mayrand et al. 1996)	[45]	Non è chiaro quale programma abbiano usato
Elva (Elva and Leavens 2012)	[45]	Non è chiaro quale programma abbiano usato
Murakami (Murakami et al. 2014)	[45]	Non è chiaro quale programma abbiano usato
Higo (Higo et al. 2007)	[45]	Non è chiaro quale programma abbiano usato
Closed Source Tools		
Codacy	[11]	https://www.codacy.com
Visual Studio	[21]	https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values
Understand	[26], [11]	https://scitools.com/feature/metrics
Jhawk	[26]	http://www.virtualmachinery.com/jhawkprod.htm
CMT++/CMTJava	[26]	https://www.verifysoft.com/en_cmtx.html
CAST's Application Intelligence Platform	[40]	https://www.castsoftware.com/products/application-intelligence-platform
Open Source Tools		
CKJM	[5], [6], [8], [12], [19]	https://www.spinellis.gr/sw/ckjm
metricsReloaded (Intellij IDEA Plugin)	[6]	https://github.com/BasLeijdekkers/MetricsReloaded
codeMetrics (Intellij IDEA Plugin)	[6]	https://github.com/kisstkondoros/codemetrics-idea
Ref-Finder (Eclipse plug-in)	[10], [16]	https://sites.google.com/site/reffindertool
Squale	[11]	http://www.squale.org
Quamoco Benchmark for Software Quality	[11]	https://github.com/wagnerst/quamoco
CBR Insight	[11]	https://github.com/StottlerHenkeAssociates
Halstead Metrics Tool	[16]	https://sourceforge.net/p/halsteadmetricstool
SonarQube and CodeAnalyzers, by SonarSource	[11], [26], [43]	https://www.sonarsource.com
jsinspect	[32]	https://www.npmjs.com/package/jsinspect
escomplex	[32]	https://github.com/escomplex/escomplex
eslint	[32]	https://eslint.org
CCFinder, now called CCFinderX	[45]	http://www.ccfinder.net/ccfinderxos.html

all'uso accademico, poiché danno la possibilità di controllare gli algoritmi usati, di modificare e/o integrare nuove funzioni e di analizzare facilmente le performances del tool in analisi.

Per ognuno dei programmi che siamo stati in grado di identificare abbiamo qui di seguito riportato una breve descrizione. I dettagli sui linguaggi e le metriche supportati da ogni tool si possono trovare dopo le descrizioni.

Programmi Proprietari

Negli studi primari analizzati abbiamo trovato sei programmi closed source, tre dei quali sono citati dallo stesso studio. I tools, descritti qui di seguito, sono presentati in ordine alfabetico.

- **Application Intelligent Platform, di CAST.** Questo programma analizza tutti i codici sorgenti di una applicazione per misurare un elenco di proprietà non funzionali come la performance, la robustezza, la sicurezza e la variabilità (che è strettamente legata alla manutenibilità). Quest'ultima proprietà è misurata basandosi sulla complessità ciclomatica, sul coupling, sul codice duplicato e sulle modifiche degli indici nei gruppi [40].

- **CMT++/CMTJava.** CMT è un programma fatto specificamente per stimare la manutenibilità di un codice scritto in C, C++ or Java e per identificare le parti meno mantenibili. È possibile calcolare molte delle metriche discusse con questo tool, come la complessità ciclomatica di McCabe, le metriche di Halstead, LOC, MI e altre ancora. Il programma funziona sia da linea di comando che tramite interfaccia grafica.

- **Codacy.** È un tool gratuito se usato per analizzare progetti open-source, altrimenti va acquistata una licenza. Uno dei suoi vantaggi è che si può anche hostare una sua istanza su un server privato. Questo tool mira a migliorare la qualità del codice, ad aumentare la *code coverage* e a prevenire problemi di sicurezza. Più che sul calcolare delle metriche, questo programma si focalizza sull'identificazione di bugs e comportamenti non definiti. Fornisce in output una lista di statistiche sul codice analizzato, contenente fra le altre cose indicazioni sulla propensione agli errori, sullo stile del codice, sulla complessità del codice, sulle parti di codice non usato e sulla sicurezza.

- **Jhawk.** Il tool è fatto per analizzare solo codice scritto in Java, ma può calcolare un vasto numero di metriche. Non è un programma appena

comparso, visto che è presente sul mercato da più di dieci anni. L'ultima versione disponibile quando si è scritta questa tesi è la v. 6.1.3, del 2017. Jhawk favorisce la valutazione empirica delle metriche, offrendo anche la possibilità di riportare in output le metriche calcolate in vari formati, fra cui XML e CSV. Ha anche un'interfaccia da terminale, per facilitarne l'utilizzo in script automatizzati.

- **Understand.** Sviluppato da SciTools, può calcolare diverse metriche, e i risultati possono essere estratti automaticamente tramite CLI, GUI o la loro API. La maggior parte delle metriche supportate da questo tool sono metriche atte a misurare la Complessità del codice (come la CC di McCabe), metriche di Volume (come LOC), e metriche per linguaggi orientati a Oggetti. La correlazione fra le metriche supportate e la dedotta manutenibilità del software non è però spiegata esplicitamente nella documentazione di questo tool.

- **Visual Studio.** È un IDE ben conosciuto, sviluppato da Microsoft, che ha al suo interno, fra tutte le sue altre funzioni, dei moduli per calcolare delle metriche riguardanti la qualità del codice. Fra le metriche listate nella sezione precedente questo IDE supporta il MI, la CC, DIT, LOC e il Class Coupling.
  La principale limitazione di questo tool è che le metriche possono essere calcolate solo per progetti scritti in C e C++, mentre tutti gli altri linguaggi supportati da Visual Studio ne restano esclusi.
  In più, nella documentazione di Visual Studio si può notare come questo IDE faccia delle assunzioni sulle metriche che sono differenti da quelle standard. Ad esempio, il valore ritornato dal MI in Visual Studio è un numero intero fra 0 e 100, che è differente dai valori standard definiti per quella metrica: per questo tool MI > 20 indica un codice facilmente mantenibile, $10 <$ MI < 19 indica un codice mediamente mantenibile e MI < 10 indica una bassa manutenibilità.

Programmi Open-Source

Negli studi primari analizzati sono stati trovati tredici tools open-source, anche se la maggior parte dei quali richiedono una licenza per essere usati per analizzare progetti non open-source, oppure per essere utilizzati senza limitazioni. Anche di questi programmi abbiamo riportato qui di seguito le loro descrizioni, sempre in ordine alfabetico, e senza nessun ordine di importanza.

- **CBR Insight.** È un programma che si basa su un altro tool, Understand (listato nella sezione precedente - Programmi Closed-Source), e si basa su quest'ultimo per calcolare le metriche. CBR Insight calcola, insieme a Understand, delle metriche che sono altamente correlate all'affidabilità e manutenibilità del codice, oltre che alla prevenzione del cosiddetto *debito tecnico*. Fornisce una Dashboard per presentare in modo più immediato i dati agli sviluppatori del sistema. Bisogna sottolineare che questo tool, sebbene open-source, necessiti di una licenza per il programma Understand, senza il quale non può funzionare.

- **CCfinderX - Code Clones Finder.** Questo tool, prima conosciuto col nome di CCFinder, individua i frammenti di codice duplicato in sorgenti scritti in Java, C, C++, C#, COBOL e VB. Al momento della stesura di questa SLR il progetto sembra essere non più sviluppato e la sua ultima versione risale a maggio 2010.

- **CKJM.** Il programma [57], citato in cinque degli studi da noi selezionati, supporta solo il Java. Può calcolare le sei metriche della suite di C&K, la metrica *afferent coupling* (Ca) e il numero di metodi pubblici (NPM). I risultati possono essere esportati in un formato XML, e il programma può essere integrato con *ant*.
  Al momento della stesura di questo testo lo sviluppo del programma sembra essere stato interrotto, poiché la sua ultima versione (v1.9) risale al 2008.

- **CodeMetrics.** È un plug-in dell' IDE IntelliJ IDEA, rilasciato sotto licenza MIT. Può calcolare la complessità di ogni metodo e il totale per ogni classe del codice sorgente analizzato, anche se non calcola la Complessità Ciclomatica standard, ma una sua approssimazione. Al momento della stesura di questo articolo il progetto è ancora attivamente mantenuto.

- **Escomplex.** È un programma che esegue una analisi della complessità del software sull'albero della sintassi astratta di un codice JavaScript. Può computare diverse metriche fra quelle identificate per rispondere alla RQ1.2, tra cui il MI, la suite di Halstead, la CC di McCabe e LOC. I risultati sono ritornati in formato Json, così possono essere facilmente parsificati e utilizzati da programmi di terze parti. Al momento della scrittura di questo testo l'ultima versione di questo programma risale alla fine del 2015.

- **Eslint.** Questo tool è un linter (un programma che analizza il codice per verificare in automatico la presenza di potenziali errori di programmazione, errori stilistici, o costrutti sospetti) per JavaScript. Il programma permette sia l'utilizzo delle regole integrate, che l'aggiunta di regole come plug-ins, caricati dinamicamente. Supporta anche la correzione automatica di alcune problematiche che vengono rilevate. Al momento della scrittura della SLR il progetto è attivamente sviluppato, con la sua ultima versione (v6.5.1) rilasciata a settembre del 2019.

- **Halstead Metrics Tool.** Un analizzatore per C, C++ e Java. Calcola solo le metriche della suite di Halstead. È scritto in Java e può esportare i risultati in formato HTML o PDF. Al momento della scrittura di questo testo, possiamo dire che il tool non è più mantenuto dallo sviluppatore, visto che non ci sono stati più aggiornamenti da dopo il 2016.

- **Jsinspect.** È un programma per analizzare codice JavaScript in cerca di *code smells*, come pezzi di codice duplicato e logica ripetuta. Il tool mira ad identificare nel progetto analizzato le porzioni di codice con strutture simili, basandosi sui tipi dei nodi nell'albero della sintassi astratta (AST). Al momento della scrittura il tool sembra non essere più sviluppato, visto che l'ultimo commit sul repository risale all'agosto del 2017.

- **MetricsReloaded.** Il programma, che è un plug-in per l'IDE IntelliJ IDEA, può anche essere utilizzato da solo, dalla linea di comando. Il progetto sembra essere stato abbandonato dal settembre 2017.

- **Quamoco Benchmark for Software Quality.** È un tool scritto in Java per analizzare codice Java. Si basa sul modello Quamoco, che mira a integrare gli attributi astratti sulla qualità del codice con valutazioni più concrete [3]. Il programma è menzionato in diversi degli studi accademici selezionati in questa SLR e il suo codice è disponibile su GitHub. Dal repository si può vedere che il suo sviluppo è stato interrotto , con l'ultimo commit risalente a luglio 2013.

- **Ref-Finder.** È un plug-in del famoso IDE open-source Eclipse. Questo tool punta ad individuare il refactoring del codice avvenuto fra due differenti versioni dello stesso programma, aiutando così gli sviluppatori a capire meglio i cambiamenti avvenuti al codice. Il plug-in può riconoscere con accuratezza anche i refactoring complessi e supporta 65 dei 72 tipi di refactoring nel catalogo di Fowler [4].

- **SonarQube.** Insieme a CodeAnalizer, è un prodotto della compagnia SonarSource. Entrambi i programmi sono forniti in due edizioni differenti: una *community edition*, open-source, e una versione commerciale. La prima edizione supporta meno metriche, meno linguaggi di programmazione e non fornisce rapporti sulla sicurezza, che sono una delle caratteristiche principali della versione commerciale. Supportano più di 25 linguaggi di programmazione (15 nella versine open-source) e centinaia di regole, fra le quali code smells e metriche di manutenibilità.

- **Squale.** Software QUALity Enhancment, è rilasciato su licenza LGPLv3. Si basa su tecnologie di terze parti, sia open-source che commerciali, che producono dati grezzi di qualità e usa dei modelli ben definiti per aggregare e analizzare i dati in un formato di più alto livello. È un programma che aiuta a valutare la qualità del software, dando come risultato delle informazioni che possono essere usate sia dai teams di sviluppo che da quelli di management, poiché tratta sia gli aspetti tecnici che quelli economici della qualità del software.
  Supporta più linguaggi di programmazione (fra cui Java, C, C++, .NET, PHP, Cobol) e utilizza metriche e modelli di qualità per stimare il livello del codice. Al momento della scrittura della SLR, il tool sembra non essere più attivamente sviluppato, con l'ultima versione del programma, la 7.1, risalente a maggio 2011.

Corrispondenza fra programmi e linguaggi

Le Tabelle 2.7 e 2.8 mostrano quali linguaggi sono supportati da ognuno dei programmi trovati. Alcuni dei tools considerati supportano una ampia varietà di linguaggi, come Understand, Codacy, e tools di SonarSource (SonarQube e CodeAnalyzer). Poiché CBR Insight, come detto prima, si basa su Understand, supporta la stessa lista di linguaggi. La maggioranza dei programmi, tuttavia, supporta un numero piuttosto esiguo di linguaggi di programmazione, se non uno solo. Jhawk, CKJM, Code metrics e Ref-finder, ad esempio, supportano solo Java. Jsinspect, Escomplex ed Eslint lavorano solo su codici JavaScript.

Dalla tabella è possibile vedere come i programmi closed source in generale supportino più linguaggi rispetto a quelli open-source (con una media di 10.5 linguaggi supportati per tool, rispetto a una media di 4.85).

Analizzando gli studi primari selezionati per questa SLR è inoltre emerso che i programmi closed source tendono a supportare alcune metriche meglio dei corrispettivi tool open-source: per esempio, uno studio comparativo

Tabella 2.7. Programmi Proprietari - Linguaggi

	Cast's AIP	Codacy	CMT++/CMTJava	Jhawk	Understand	Visual Studio
Ada					X	
APAB						
Apex		X			X	
Assembly					X	
C	X	X	X		X	X
C++	X	X	X		X	X
C#	X	X	X		X	
Cobol	X				X	
CSS						
Fortran	X				X	
Go						
HTML						
Java	X	X	X	X	X	
JavaScript	X	X			X	
Jovial					X	
Json		X			X	
JSP	X	X			X	
Kotlin		X			X	
Markdown		X			X	
Objective C						
PHP	X	X			X	
Python	X	X			X	
RPG						
Ruby		X			X	
Scala		X			X	
SQL						
Swift						
T-SQL						
TypeScript						
VB6	X	X			X	
VB.NET	X	X			X	
Velocity		X			X	
VisualForce		X			X	
XML		X			X	

fra diversi programmi capaci di calcolare il MI ha riportato una affidabilità maggiore per i risultati ottenuti tramite tools proprietari [23].

L'Immagine 2.4 mostra quanti tools closed source e open-source sono stati trovati per ogni linguaggio. Da questo diagramma è immediato vedere come alcuni linguaggi di programmazione siano meglio supportati di altri. Java, C, C++, seguiti da JavaScript e C#, sono supportati da almeno la metà dei programmi che abbiamo considerato nel nostro studio. Più nello specifico,

Tabella 2.8. Programmi O.S. - Linguaggi

	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco Benchmark	CBR Insight	Halstead Metrics Tool	SonarQube/CodeAnalyzer	jsinspect	escomplex	eslint	CCFinderX	Ref-Finder tool
Ada						X							
APAB								X					
Apex						X		X					
Assembly						X							
C		X		X		X	X	X				X	
C++				X		X	X	X				X	
C#						X		X				X	
Cobol				X		X		X				X	
CSS								X					
Fortran						X							
Go								X					
HTML								X					
Java	X	X	X		X	X	X	X				X	X
JavaScript						X		X	X	X	X		
Jovial						X							
Json						X							
JSP						X							
Kotlin						X		X					
Markdown						X							
Objective C								X					
PHP						X		X					
Python						X		X					
RPG								X					
Ruby						X		X					
Scala						X		X					
SQL								X					
Swift								X					
T-SQL								X					
TypeScript								X					
VB6						X							
VB.NET						X							
Velocity						X							
VisualForce						X							
XML						X		X					

Java, C, C++ e C# sono supportati da quasi tutti i programmi closed source che abbiamo trovato, mentre alcuni linguaggi meno popolari (come APAB, RPG, T-SQL) sono invece supportati, fra tutti i programmi che abbiamo trovato negli studi primari selezionati, solo da tools open-source.

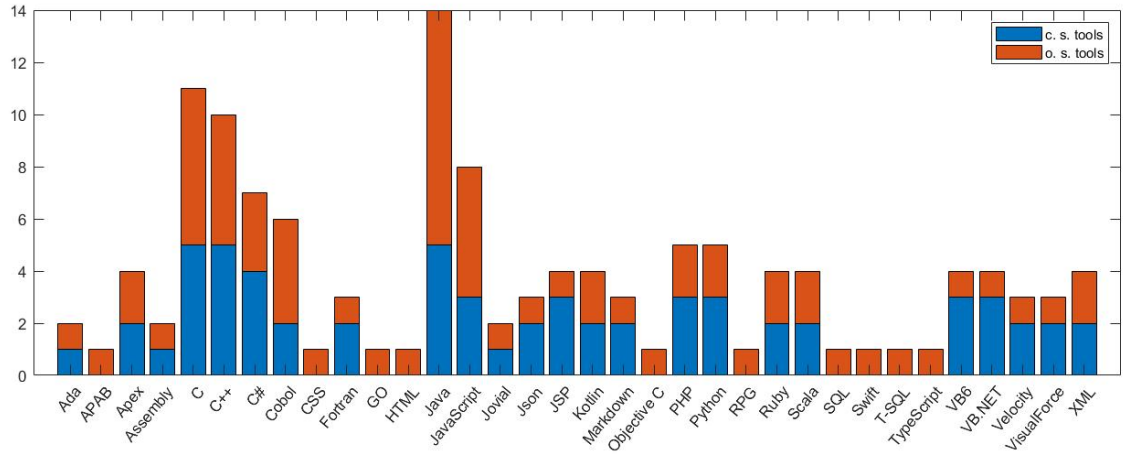

Figura 2.4. Numero di programmi (proprietari ed open-source) per linguaggio

Corrispondenza fra programmi e metriche

Le Tabelle 2.9 e 2.10 mostrano quali metriche sono calcolate dai vari programmi considerati. Per facilitarne la lettura, sono state riportate solo le metriche calcolate da almeno uno dei programmi considerati. Nella prima sezione della tabella, quella più in alto, sono riportate le metriche più popolari, identificate per rispondere alla RQ1. Nella tabella le metriche sono state segnate come supportate da un certo programma solo nel caso in cui si sia trovato un riferimento esplicito a quella metrica nella documentazione del tool. Le suites sono state considerate come supportate dal programma anche se non tutte le metriche facenti parte del set vengono supportate.

Nel caso dei programmi closed source, le metriche sono state ricavate dalla documentazione, che si è spesso rivelata essere piuttosto limitata. Infatti la maggior parte delle volte i tool di questo tipo forniscono un'interfaccia in stile pannello di controllo e i risultati delle varie metriche vengono valutati secondo dei criteri personalizzati, che rendono poco chiaro il rapporto che c'è fra quei risultati ottenuti e i valori che si otterrebbero seguendo puramente la definizione di quelle metriche. Ad esempio, il programma Codacy fornisce un'unica valutazione (un voto da A ad F) complessiva per un progetto, ma questo voto dipende da una serie di parametri specifici al programma in questione e si basa su vari aspetti del codice: error proness, code cmoplexity, lo stile del codice, codice non usato, sicurezza, compatibilità, documentazione e performances.

Tabella 2.9. Programmi Proprietari - Metriche

	Cast's AIP	Codacy	CMT++/CMTJava	Jhawk	Understand	Visual Studio
McCabe's CC	X	X	X	X	X	X
Ce, Efferent Coupling	X			X	X	X
CHANGE						
C&K					X	X
CLOC		X		X	X	
Halstead			X	X		
JLOC		X				
LOC	X	X	X	X	X	X
LCOM2	X				X	
MI			X	X		X
Message Passing Coupling	X			X		
NOM, No. of Methods					X	
NPM, No. Public Methods					X	
STAT, No. Statements				X	X	
McCabe's WMC					X	
AvgCC					X	
Ca, Afferent Coupling						
Coupling Between Obj Classes						
Code Smells						
Essential Complexity					X	
MND, Maximum Nexted Depth			X	X	X	
Mood's Metrics for O.O. Design						
NIV, No. Instance Variables				X	X	
NOC, No. Children						
No. Classes						
No. Commands				X		
No. Directories						
No. Files					X	
No. Queries	X					
RFC, Response For a Class				X		
Test Results and Coverage						
Others	X			X	X	

In alcuni casi, oltre ad alcune metriche che sono state menzionate esplicitamente dai creatori del programma, non è stato possibile trovare una lista completa di tutte le metriche usate internamente dal tool preso in esame.

In molti casi i programmi calcolano anche metriche composte o derivate, metriche che si calcolano a partire da altre metriche presenti in letteratura (ad esempio WMC di McCabe, che si calcola a partire dalle CC di tutte le funzioni di una classe), o metriche che non sono proprio state menzionate negli articoli analizzati per rispondere alla QR1. Per comprendere questi

Tabella 2.10. Programmi O.S. - Metriche

	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco Benchmark	CBR Insight	Halstead Metrics Tool	SonarQube/CodeAnalyzer	jsinspect	escomplex	eslint	CCFinderX	Ref-Finder tool
McCabe's CC		X	X	X				X		X	X	X	
Ce, Efferent Coupling		X		X									
CHANGE												X	X
C&K	X	X											
CLOC						X		X			X		
Halstead							X			X			
JLOC		X		X									
LOC		X		X	X	X		X	X		X	X	
LCOM2													
MI								X		X			
Message Passing Coupling													
NOM, No. of Methods				X				X					
NPM, No. Public Methods	X			X				X					
STAT, No. Statements								X			X		
McCabe's WMC	X					X							
AvgCC													
Ca, Afferent Coupling	X	X		X									
Coupling Between Obj Classes	X					X							
Code Smells								X	X		X	X	X
Essential Complexity		X											
MND, Maximum Nexted Depth											X		
Mood's Metrics for O.O. Design		X											
NIV, No. Instance Variables													
NOC, No. Children	X												
No. Classes		X		X				X			X		
No. Commands													
No. Directories								X					
No. Files								X					
No. Queries													
RFC, Response For a Class	X					X							
Test Results and Coverage		X		X				X					
Others		X		X	X	X		X	X	X	X	X	X

casi particolari abbiamo aggiunto in fondo alla tabella una riga denominata *Others*, proprio per indicare che il programma in questione può analizzare altre metriche, oltre a quelle che abbiamo riportato in tabella.

Come è evidente dalla tabella, nessuno dei programmi supporta tutte le metriche che abbiamo definito come le più popolari. Il numero di metriche, fra quelle più popolari, supportate da un singolo programma va da uno solo, a dieci. Due programmi, ad esempio, presentano solo una metrica, o suite di

metriche, fra la lista di quelle più popolari. Il programma *Halstead Metrics Tool*, come indicato nel nome, è un tool open-source che ha lo scopo di calcolare solo le metriche presenti nella suite di Halstead. Su questo stesso stile è il plug-in *CodeMetrics*, che calcola solamente la Complessità Ciclomatica di McCabe, riportando il suo valore per ogni metodo presente nel progetto e il valore totale per classe.
Quamoco non è sicuramente solo un tool, ma anche un meta modello di qualità, basato su un set di metriche che sono state definite - nell'ambito dell'articolo scientifico che ha presentato questo approccio - come misure di base. Il meta-modello è teoricamente applicabile a ogni tipo di misura che può essere calcolata attraverso un'analisi statica del codice sorgente, tuttavia l'articolo che cita il tool menziona esplicitamente solo la metrica LOC. Altri programmi, come JSInspect, CCFinderX e Ref-Finder Tool, comprendono un elenco limitato delle metriche di manutenibilità identificate in precedenza, visto che la loro attenzione è concentrata su altri aspetti della qualità del codice, come l'individuazione di parti di codice duplicate o code smells.

Programmi come MetricsReloaded, Squale e SonarQube supportano una lunga lista di metriche derivate, che sono state ottenute tramite la specializzazione, la somma o la media di metriche di base, come la Complessità Ciclomatica o la Coupling Between Classes.

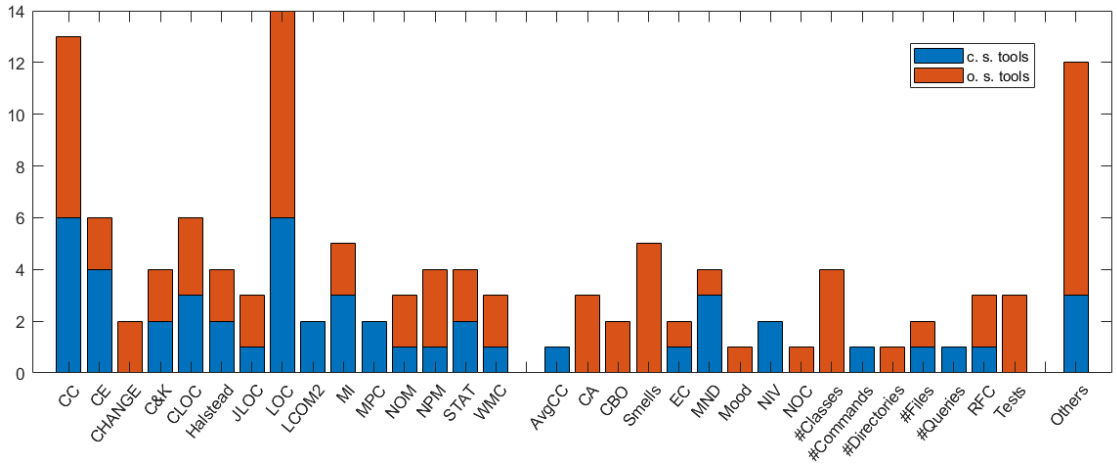

Figura 2.5. Numero di programmi (proprietari ed open-source) per metrica

Il grafico a barre in Figura 2.5 riporta il numero di programmi che supportano ognuna delle metriche considerate. Si può notare come sull'asse delle x non vi siano solo le metriche / suites ritenute più popolari (trovate rispondendo alla RQ1.2), ma anche altre metriche ricavate dalla lista completa.

Tabella 2.11. Programmi disponibili per calcolare le metriche più comuni per i linguaggi più comuni (in grassetto i tools open-source)

Metrica	C	C++	C#	Java	JavaScript
CC - McCabe's Cyclomatic Complexity	CAST's AIP Codacy CMT++ Understand Visual Studio **MetricsReloaded** **Squale** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMT++ Understand Visual Studio **Squale** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMT++ Understand **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMTJava JHawk Understand **MetricsReloaded** **CodeMetrics** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy Understand **CodeAnalyzers** **Escomplex** **Eslint**
Ce - Efferent Coupling	CAST's AIP Understand Visual Studio **MetricsReloaded** **Squale**	CAST's AIP Understand Visual Studio **Squale**	CAST's AIP Understand	CAST's AIP JHawk Understand **MetricsReloaded**	CAST's AIP Understand
CHANGE - No. of lines Changed in Class	**CCFinderX**	**CCFinderX**	**CCFinderX**	**CCFinderX** **Ref-Finder**	
C&K - Chidamber and Kemerer Suite	Understand Visual Studio **MetricsReloaded**	Understand Visual Studio	Understand	Understand **CKJM** **MetricsReloaded**	Understand
CLOC - Comment Lines of Code	Codacy Understand **CBRInsight** **CodeAnalyzers**	Codacy Understand **CBRInsight** **CodeAnalyzers**	Codacy Understand **CBRInsight** **CodeAnalyzers**	Codacy JHawk Understand **CBRInsight** **CodeAnalyzers**	Codacy Understand **CBRInsight** **CodeAnalyzers** **Eslint**
Halstead's Suite	CMT++ **Halstead Metrics T.**	CMT++ **Halstead Metrics T.**	CMT++	CMTJava JHawk **Halstead Metrics T.**	Escomplete
JLOC	—	—	—	Codacy **MetricsReloaded** **Squale**	—
LOC	CAST's AIP Codacy CMT++ Understand Visual Studio **MetricsReloaded** **Squale** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMT++ Understand Visual Studio **Squale** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMT++ Understand **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy CMTJava JHawk Understand **MetricsReloaded** **QuamocoBenchmark** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST's AIP Codacy Understand **CBR Insight** **CodeAnalyzers** **JSInspect** **Eslint**
LCOM2 - Lack of Cohesion in Methods	CAST's AIP Understand	CAST's AIP Understand	CAST's AIP Understand	CAST's AIP Understand	CAST's AIP Understand
MI - Maintainability Index	CMT++ Visual Studio **CodeAnalyzers**	CMT++ Visual Studio **CodeAnalyzers**	CMT++ **CodeAnalyzers**	CMTJava JHawk **CodeAnalyzers**	**CodeAnalyzers** **Eslint**
MPC - Message Passing Coupling	CAST's AIP	CAST's AIP	CAST's AIP	CAST's AIP JHawk	CAST's AIP
NOM - Number of Methods	Understand **Squale** **CodeAnalyzers**	Understand **Squale** **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**
NPM -Number of Public Methods	Understand **Squale** **CodeAnalyzers**	Understand **Squale** **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CKJM** **CodeAnalyzers**	Understand **CodeAnalyzers**
STAT - Number of Statements	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	JHawk Understand **CodeAnalyzers**	Understand **CodeAnalyzers** **Eslint**
WMC - McCabe's Weigthed Methods Count	Understand **CBR Insight**	Understand **CBR Insight**	Understand **CBR Insight**	Understand **CKJM** **CBR Insight**	Understand **CBR Insight**

Due metriche, in particolare, si sono distinte per il numero di programmi che le supportano: LOC e CC. La prima, sebbene molti articoli in letteratura si interrogassero sulla sua efficacia come metrica per stimare la manutenibilità, è supportata da 14 dei 19 tools trovati. La seconda, CC, è supportata da 13 programmi. Il fatto che LOC e CC siano così popolari non è così inaspettato, visto che sono entrambe facili da computare e sono necessarie per calcolare molte altre metriche derivate.
In opposizione a quelle, invece, vi sono tre metriche (LCOM2, CHANGE, MPC) che sono utilizzate solo da due dei programmi analizzati. La metrica CHANGE si riferisce alle linee di codice che sono cambiate fra due differenti versioni dello stesso codice e non è calcolata dalla maggioranza dei tools, poiché fanno analisi statiche solamente su una versione del codice. Questa metrica è calcolata solo da due dei programmi trovati, che puntano a misurare il code refactoring e i code smells. MPC, Message Passing Coupling, è utilizzata da due tools, ma in entrambi i casi viene definita con il nome di fan-out, suo sinonimo.

In generale, i programmi proprietari trovati supportano un numero maggiore di metriche rispetto a quelli open-source, che spesso si sono rivelati essere progetti piccoli, come plug-in, fatti per calcolare una singola suite o metrica.
Considerando solo le metriche trovate negli studi primari, i programmi proprietari sono in grado di calcolare in media poco meno di 8 metriche, mentre quelli open-source circa 5. Prendendo in considerazione solo le metriche facenti parte della lista delle 15 più popolari, vediamo che in media ogni tool proprietario ne calcola 6, mentre per i programmi open-source il valore scende a 3.

La Tabella 2.11 riporta e riassume per i cinque linguaggi più supportati (vedi Figura 2.4) i programmi atti a calcolare le metriche più popolari. Come si può vedere, abbiamo preso in considerazione i linguaggi che erano supportati da almeno 7 dei programmi trovati (ben sopra la media di tutti i linguaggi analizzati).
Per quanto riguarda la metrica JLOC, essendo specifica per Java, non è stata presa in considerazione per gli altri linguaggi di programmazione, per i quali non ha nessun senso.
Nella tabella i programmi open-source sono stati riportati in grassetto per metterli in evidenza e distinguerli facilmente da quelli proprietari.
Come si può vedere, per ogni linguaggio le metriche più comuni possono essere calcolate utilizzando diversi programmi, sia open-source che closed source. Tuttavia, diverse metriche possono essere calcolate da un singolo tool: per

esempio, CCFinderX è l'unico programma che supporta esplicitamente la metrica CHANGE per i linguaggi della famiglia del C; la metrica MPC (Message Passing Coupling) è esplicitamente supportata solo dal programma Application Intelligent Platform di CAST, per i linguaggi della famiglia del C e per il JavaScript.

Tabella 2.12. Set ottimale di tools per i linguaggi di programmazione più supportati

Linguaggio	Set ottimale di tools	Metriche coperte
C	CAST's AIP, Understand, CCFinderX, CMT++	14/14
C++	CAST's AIP, Understand, CCFinderX, CMT++	14/14
C#	CAST's AIP, Understand, CCFinderX, CMT++	14/14
Java	(AIP or Jhawk), (CCFinderX or Ref-Finder), Understand, CTMJava, (MetricsReloaded or Squale or Codacy)	15/15
JavaScript	CAST's AIP, Understand, Escomplete, (CodeAnalyzers or Eslint)	14/14

Tabella 2.13. Set ottimale di tools open-source per i linguaggi di programmazione più supportati

Linguaggio	Set ottimale di tools	Metriche coperte
C	CBR Insight, CCFinderX, CodeAnalyzers, Halstead Metrics Tool, MetricsReloaded	12/14
C++	CBR Insight, CCFinderX, CodeAnalyzers, Halstead Metrics Tool, Squale	11/14
C#	CBR Insight, CCFinderX, CodeAnalyzers	9/14
Java	(CCFinderX or Ref-Finder), CKJM, CodeAnalyzers, Halstead Metrics Tool, MetricsReloaded	13/15
JavaScript	CBR Insight, CCFinderX, CodeAnalyzers	8/14

Le Tabelle 2.12 e 2.13 mostrano il set ottimale di programmi per coprire tutte le metriche più popolari mostrate nella Tabella 2.5. La prima tabella, la 2.12 , prende in considerazione sia i programmi open-source che quelli proprietari, mentre la seconda, la 2.13, considera solo quelli open-source.
Abbiamo definito come set ottimale di programmi un elenco col minor numero possibile di tools che possano coprire il maggior numero possibile di metriche, e suites di metriche, delle 14 più citate (15 nel caso di Java). Nel caso di tools equivalenti, che non cambiano la lunghezza del set o il numero di metriche calcolabili, i programmi sono stati scritti fra parentesi.

Utilizzando sia i programmi open-source che quelli proprietari è possibile calcolare tutte le metriche più citate con un set ottimale di 4 tools per tutti

i linguaggi, eccetto Java che necessita di 5 tools. Per tutti i linguaggi della famiglia del C tutte le metriche sono coperte da Application Intelligence Platform di CAST, Understand, CCFinderX e CMT++; per coprire tutte le metriche per Java, oltre a sostituire CMT++ con CMTJava, serve un tool in più fra Squale, Codacy o MetricsReloaded, per calcolare JLOC. Per JavaScript, CCFinderX e CMT vengono rimpiazzati da Escomplete ed uno fra CodeAnalyzers e Eslint.

Volendo solo usare programmi open-source non è possibile ottenere la piena copertura del set di metriche più citate, anche perché metriche come LCOM2 e MPC non sono supportate da nessuno dei tools open-source trovati. Il massimo numero di metriche che sarebbe possibile supportare con solo questi programmi open-source varia da 8, per il JavaScript, a 13, per il Java.

Capitolo 3

Metrics Aggregator

Nel capitolo precedente della tesi sono state analizzate, facendo una SLR, le metriche più comuni che si possono trovare in letteratura e si è visto quali programmi venissero utilizzati per calcolarle negli articoli scientifici presi in considerazione.
Dopo aver creato un set contenente le metriche statiche più utilizzate dai ricercatori si è visto che non esisteva un singolo programma in grado di calcolarle tutte. L'unico modo per calcolarle è utilizzare un set di programmi già esistenti, ma anche in questo caso, per analizzare un codice sorgente, si andrebbe incontro a diverse problematiche, fra le quali la scelta dei tools da utilizzare, come utilizzarli e come interpretare i risultati ottenuti da ognuno di questi.
Comprendere i risultati dati dai programmi può rivelarsi non immediato nemmeno per coloro che conoscono bene le metriche riportate, poiché alcuni tools computano le metriche non seguendo le formule standard con le quali sono state definite, oppure mascherano i valori ottenuti e danno direttamente delle votazioni al codice, basate sui risultati ottenuti.
Infine non vi è nemmeno un metodo per aggregare i risultati ottenuti in modo omogeneo, indipendente dai singoli programmi scelti per fare l'analisi del codice.

Per assolvere a questi problemi, quindi, si è pensato di creare un programma open-source che potesse andare a risolvere proprio queste criticità. Il tool sviluppato non è un nuovo programma che re-implementa le metriche da zero, bensì è un *aggregatore* che sfrutta programmi di terze parti, preesistenti, per computare le metriche desiderate e presentare i dati in una maniera ben definita; il tutto viene svolto in modo automatizzato per l'utente che, una

volta installato questo aggregatore, potrà utilizzarlo da riga di comando o integrarlo comodamente in un proprio script.

I programmi di terze parti utilizzati per analizzare il codice sono stati raccolti in una cartella ed è stato utilizzato un Makefile per poterli compilare tutti in modo automatico.
L'aggregatore è invece stato scritto in Python, in quanto ritenuto un linguaggio di alto livello ben conosciuto, facile da comprendere e, se necessario, veloce da apprendere. In un primo momento si è pensato di scriverlo in Bash ma, data la complessità del compito, si sarebbe ottenuto un codice meno leggibile, modulare ed estensibile.

Nello scrivere il programma si sono tenute in considerazione delle *best practices* per rendere più leggibile il codice, quali l'utilizzo delle linee guida PEP-8 [1] (linee guida sullo stile del codice), mantenere il codice modulare, contenere la lunghezza delle funzioni e aggiungere commenti dove ritenuto utile.

Il programma creato è open-source, così che possa essere utilizzato facilmente da altri ricercatori e modificato secondo le loro necessità. Il repository contenente il codice si può trovare su GitHub all'indirizzo https://github.com/SoftengPoliTo/SoftwareMetrics.

Per facilitare l'aggiunta di nuovi tools e nuove metriche si è cercato di progettare il programma in maniera modulare, in modo che l'aggiunta di un nuovo analizzatore non richiedesse modifiche al codice già esistente. Alcune (minime) modifiche potrebbero, come spiegato in seguito, essere necessarie per garantire un certo tipo di qualità sui risultati da parte dell'aggregatore.

La Figura 3.1 fa vedere come è strutturato l'aggregatore, mostrando le sue parti logiche principali, che sono descritte qui di seguito.

3.1 Interfaccia e Output dell'aggregatore

3.1.1 Interfaccia e Input

Il programma ha una semplice interfaccia da linea di comando, che accetta in input sia un path ad un file o una cartella da analizzare, che il path ad un file di tipo *Json Compilation Database*. Questi ultimi sono files in formato Json che vengono creati durante la compilazione di un codice sorgente e

[1] https://www.python.org/dev/peps/pep-0008

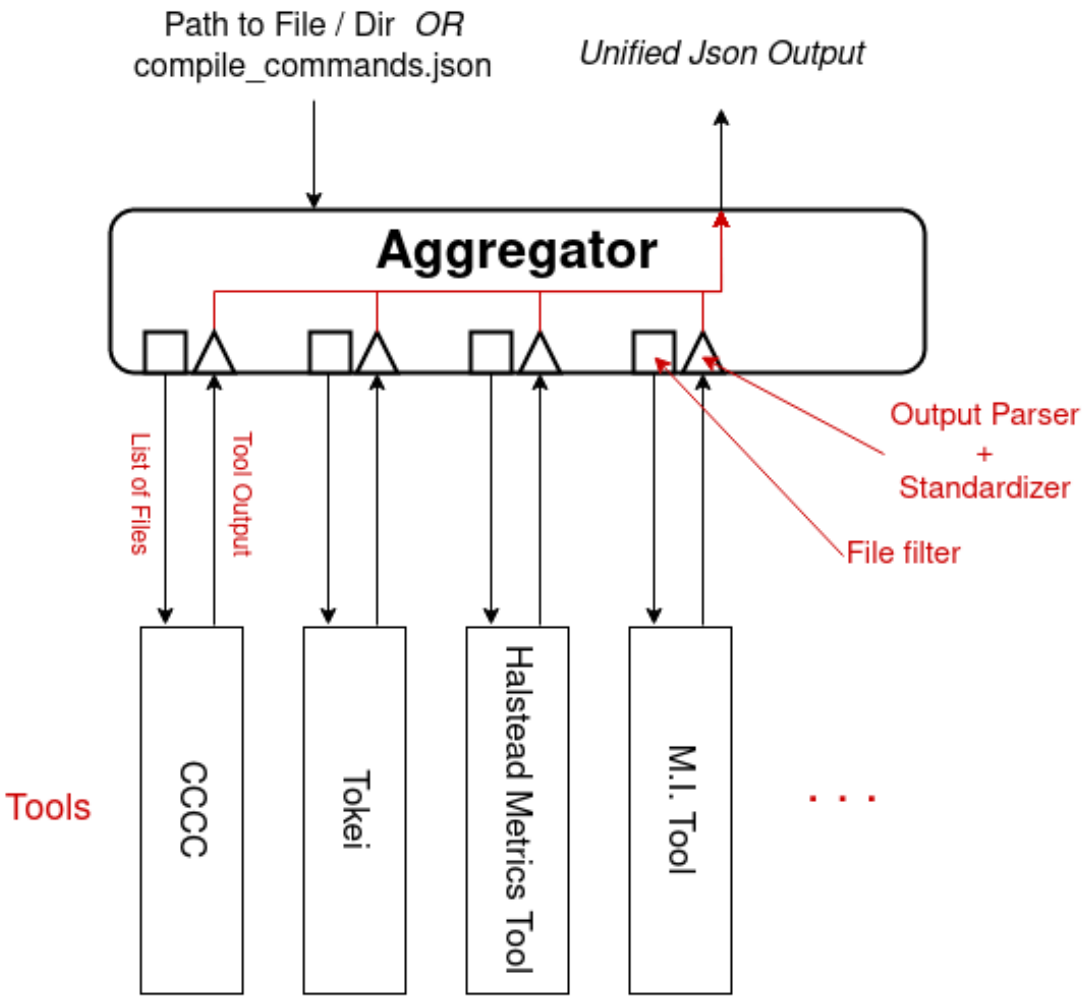

Figura 3.1. Struttura dell'Aggregatore

specificano come riprodurre una certa compilazione. Sono convenzionalmente denominati "compile_commands.json" e consistono in un array di oggetti denominati "command objects". Ognuno di questi oggetti contiene la directory di lavoro, il comando di compilazione eseguito e il file sorgente processato in quel preciso passo della compilazione.
Supportare questo tipo di database permette di utilizzare l'analizzatore per fare anche delle analisi più specifiche, ad esempio può essere utile a studiare dei sorgenti pensati per funzionare su diverse architetture, dando la possibilità di analizzare il progetto nella sua interezza o solo i files che sono necessari per una certa architettura.

3.1.2 Output

Quando viene eseguito l'aggregatore si può specificare una directory di destinazione dove salvare i risultati ottenuti dall'analisi. Nel caso non si indichi nessuna cartella di output, il programma provvederà a crearne una nella directory corrente di lavoro.

Nella directory di output vi potranno essere delle sottocartelle contenenti i risultati dei singoli analizzatori chiamati dall'aggregatore e il file Json contenente i risultati aggregati.

Formato dell' Output Json

Il file Json creato dall' aggregatore contiene i risultati ottenuti dai vari analizzatori, ordinati secondo una struttura dati ben definita.

```
{
  // Global stats
  "LOC": 0,
  "CLOC": 0,

  "Halstead": {
     "Operands": {
        "..." : 0,
        // ...
     },
     "Operators": {
        "..." : 0,
        // ...
     },
     "n1": 0.0,
     "n2": 0.0,
     "Vocabulary": 0.0,
     // ...
  },
  // END of Global stats

  "classes": [{          // Per-class stats
     "C&K":{
        // ...
```

```
25          }
26          "WMC": 0,
27          // ...
28      }],
29
30      "files": [{             // Per-files stats
31          "filename": "/path/to/file",
32          "type": "C",
33
34          "LOC": 0,
35          "CLOC": 0,
36          // ...
37
38          "Halstead": {
39              // ...
40          },
41
42          "functions": [{         // Per-function metrics
43              "function name": "name(...)",
44              "line number": 1,
45
46              "LOC": 0,
47              "CLOC": 0,
48              "Lines": 0,
49              // ...
50          }]
51      }]
52  }
```

Come si può vedere dallo spezzone di codice Json riportato qui sopra, i dati sono organizzati in una struttura ad albero. Sono stati inseriti dei "commenti" in stile C++ al solo fine di migliorarne la leggibilità.
La Figura 3.2 riporta in un diagramma a blocchi la medesima struttura, per una più facile ed immediata comprensione.
Vi è un primo livello contenente i risultati globali delle metriche, ovvero i risultati su tutti i files analizzati, come, ad esempio, le linee di codice totali o le linee di commento totali.
All'interno dell'oggetto radice, oltre ai valori globali, vi sono due oggetti contenitore: una lista di oggetti denominata "files" e una lista (opzionale) di

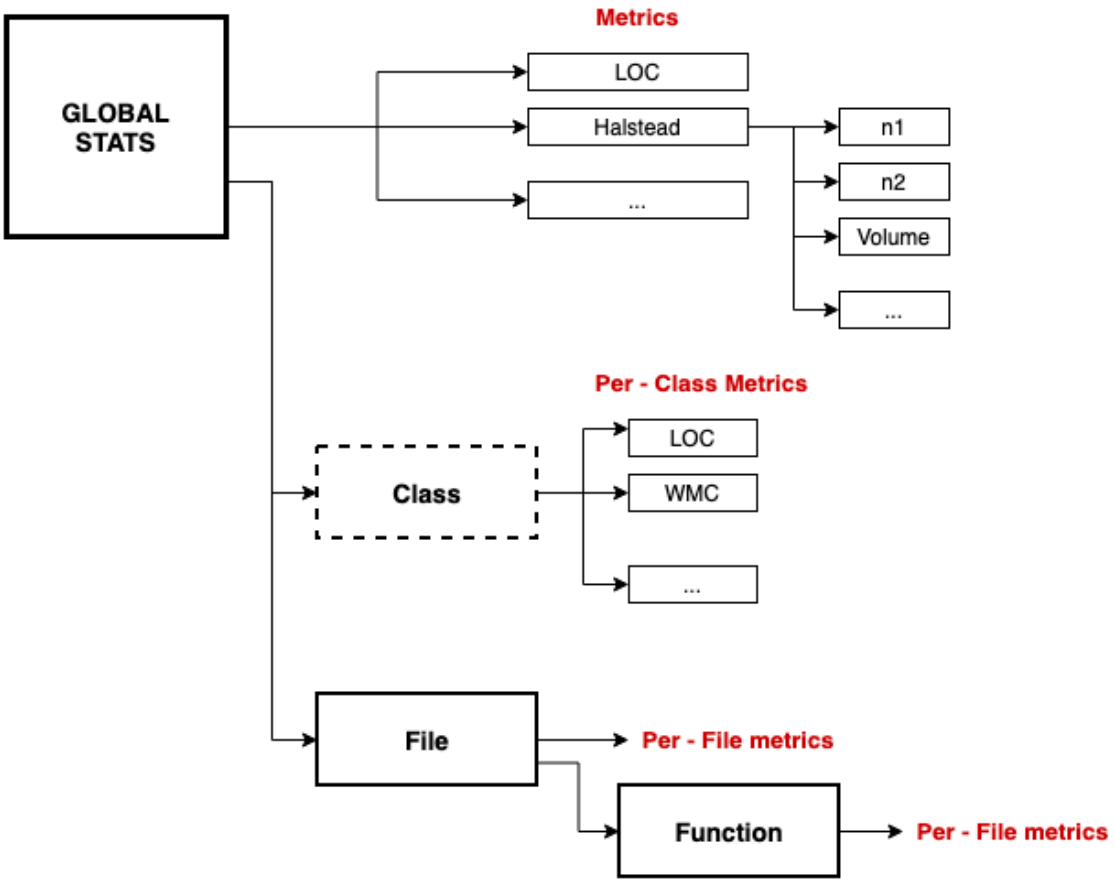

Figura 3.2. Json output structure

classi.
La lista delle classi è opzionale, perché è popolabile solo nel caso in cui si analizzi un linguaggio di programmazione orientato ad oggetti, ed ogni oggetto che la compone contiene al suo interno una lista di metriche che hanno come scope la classe ed, eventualmente, informazioni sulla classe stessa.
Gli oggetti nella lista di files contengono le informazioni ed i valori calcolati per il file in esame. All'interno di questi troviamo l'elenco delle funzioni dichiarate in quel file, con le relative metriche.
Le metriche semplici sono riportate come Json Strings o Json Values, aventi come Key il nome stesso della metrica, per intero o come sigla. Le metriche appartenenti ad una suite vengono raggruppate all'interno di un Json Object chiamato con il nome della suite.

3.2 Analizzatori

Come prima cosa sono stati cercati dei programmi che calcolassero le metriche volute. Sono stati scelti dei programmi open-source poiché, come detto in precedenza, potendo leggere il codice è possibile controllare come vengono computate le metriche ed, eventualmente, modificare il comportamento di tali strumenti, cosa che è poi avvenuta. Sapere esattamente come vengono calcolate le varie metriche è di grande importanza nell'ottica del progetto per Mozilla, in quanto nella fase finale, quella di confronto dei due linguaggi, si potrà sapere esattamente come vengono calcolate tutte le metriche ed avere, così, dei risultati di maggiore significato.
Inoltre, scegliendo dei programmi open-source in questo punto, la fase futura di porting delle metriche su Rust dovrebbe risultare semplificata, in quanto si disporrà non solo degli articoli scientifici descriventi le varie metriche, ma si potrà vedere effettivamente come sono stati implementati gli algoritmi per analizzare i linguaggi C e C++.

I tools presi in considerazione sono stati scelti guardando fra quelli trovati negli articoli analizzati scrivendo la SLR, fra quelli già conosciuti dagli autori della medesima e utilizzando programmi trovati tramite ricerche online.

3.2.1 Programmi scelti

Di seguito sono elencati i programmi che, al momento della scrittura della tesi, sono inclusi nell'aggregatore. Sono stati anche riportati gli url ai repository originali, anche se alcuni di questi analizzatori sono stati modificati e corretti, prima di essere utilizzati. I links alle versioni modificate si possono facilmente trovare nel repository dell'aggregatore indicato all'inizio di questo capitolo.

- **CCCC** [2]**:** può analizzare codice scritto in Java, C e C++ e può computare la metrica LOC, la Complessità Ciclomatica (CC) di McCabe e la suite di Chidamber e Kemerer (C&K). È citato in letteratura, ma il suo sviluppo è interrotto dal 2013.
- **Tokei** [3]**:** è un programma scritto in Rust che si occupa di calcolare esclusivamente le metriche LOC e CLOC, di riportare il numero di linee

[2] https://sourceforge.net/projects/cccc

[3] https://github.com/XAMPPRocky/tokei

vuote ci sono in ogni file e il numero totale di files analizzati. Il progetto è attivamente mantenuto dalla community, è multipiattaforma, è estremamente veloce nel fare l'analisi del codice parsificato, supporta centinaia di estensioni diverse e i risultati, oltre che stampati su terminale, possono essere riportati in formato Json, Yaml, Toml o Cbor.

- **Maintainability Index [4]:** è un programma scritto in python per analizzare codici C, C++ e Java, basato su lizard code complexity analyzer. Fra le metriche che può calcolare troviamo la CC, il numero di parametri delle funzioni e il Maintainability Index di ognuna di esse. Il programma non è più sviluppato dalla fine del 2014.

- **Halstead Metrics Tool [5]:** è un analizzatore per codici Java, C e C++, è scritto in Java. Si incentra solo sulle metriche della suite di Halstead. È l'unico dei tools qui presentati che presenti un'interfaccia grafica, ma non presenta un'interfaccia da terminale. Esporta i dati solo in formato Pdf o Html. L'ultima versione di questo tool risale alla fine del 2016.

3.2.2 Modifiche effettuate agli analizzatori

Mentre venivano scelti quali tools utilizzare, durante dei test preliminari, è risultato evidente che il programma "Halstead Metrics Tool" necessitasse di modifiche. Quest'ultimo, infatti, invertiva i valori n_1 con N_1 e n_2 con N_2, calcolava il Volume di Halstead con una formula errata e non aveva un'interfaccia che permettesse di farlo funzionare in maniera automatica da linea di comando.
Dopo aver corretto gli errori alle formule, si è creata una semplice interfaccia da terminale per questo analizzatore.
Poiché i due formati (pdf e html) forniti in output da questo tool non sarebbero risultati facili da parsificare con l' aggregatore, si è aggiunta la possibilità di ottenere i risultati in formato Json.

In questa fase si è rivelato importante aver scelto di utilizzare programmi open source, oppure si sarebbe dovuto cercare un tool diverso per calcolare la suite di Halstead, oppure crearne uno nuovo.
Gli altri programmi non hanno avuto bisogno di modifiche sostanziali, è stata solo applicata una patch al tool CCCC.

[4] https://sourceforge.net/projects/maintainabilityindex

[5] https://sourceforge.net/projects/halsteadmetricstool

3.3 Input per gli analizzatori

Ogni analizzatore viene chiamato tramite il modulo python *subprocess*, che lo esegue come nuovo processo.
Ad ogni tool viene passata una *lista di files* da analizzare, partendo dal path o dal file compile_commands.json specificati. Nel caso ciò non sia supportato dal programma specifico, vi è la possibilità di eseguire il tool una volta per ogni file da analizzare oppure di passargli direttamente il path alla directory da analizzare e scartare successivamente i risultati non voluti. In fase di programmazione, per l'aggiunta dello strumento, va studiato e applicato l'approccio più corretto e applicabile in generale.

Quando si passa la lista di files da analizzare al tool, o quando lo si esegue su ogni singolo file, bisogna avere l'accortezza di scartare i files che non può analizzare, per evitare eventuali risultati errati. Questo controllo lo si è implementato basandosi sulle estensioni dei singoli files.

3.4 Processare output degli analizzatori

Ogni singolo analizzatore ha un proprio modo di presentare i risultati ottenuti, sia sotto il punto di vista della struttura dei risultati, che per i formati scelti. CCCC, ad esempio, ritorna i risultati scrivendoli in diversi files Xml dentro una nuova directory. Mentre M.I. Tool ritorna i risultati in formato Xml tramite stdout.
Poiché vi è una gran differenza fra i vari output, e la complessità nel processarli varia molto, si è deciso di dividere in *due parti* questo il processo: una parte di parsing e una di formattazione dei dati. Le funzioni che si occupano di questo si trovano nel file denominato *output_unifier.py*.

Una volta ottenuto un output da un analizzatore, qualunque sia la sua forma, lo si parsifica in dati più facilmente gestibili dal programmatore e comprensibili dall'interprete python. Questo compito è demandato a funzioni che sono state chiamate, per convenzione, *output_reader functions*.

Nella seconda parte di questo sotto-processo i dati ottenuti vengono ora elaborati e modificati per dar loro una forma standard. Per questo motivo queste funzioni sono state denominate *standardizer functions*. La forma standard a cui ci si riferisce è una rappresentazione interna, tramite liste e dizionari, del formato output in Json descritto nella Sezione 3.1.2.

3.5 Unificazione dei singoli outputs e calcolo metriche secondarie

Per unire assieme i risultati ottenuti con i vari analizzatori si è creata una funzione chiamata *unifier_merger*, che permette di fondere assieme in maniera automatica i risultati di due tools qualsiasi, fintantoché entrambi i dati passati rispettino la struttura definita per l'output Json. Per questa ragione è di particolare importanza, quando si aggiunge il supporto per un nuovo analizzatore, controllare che l'output della standardizer function rispetti la struttura definita.

Nel caso in cui la funzione di merging trovi nell'output da unire una metrica già presente nella struttura dati principale, questa verrà semplicemente scartata. In questo modo si semplifica la scrittura delle funzioni di parsing descritte nella sezione precedente.
Se più tools computano una stessa metrica e si ritiene che uno di essi non la calcoli con sufficiente accuratezza, basterà cancellare la metrica desiderata dall'output della standardizer function di quell'analizzatore, evitando così che i risultati di minore qualità vadano ad essere inseriti nei risultati finali.

3.5.1 Metriche secondarie

Dopo aver unito assieme gli outputs di tutti i tools, l'aggregatore chiama le funzioni presenti nel file *metrics.py*. In questo file vi sono solo funzioni il cui scopo è, partendo dall'output standardizzato, quello di completare e integrare i risultati ottenuti dagli altri tools, nonché di computare ulteriori metriche secondarie.

Si parla di integrare le metriche già esistenti quando, ad esempio, un analizzatore calcola una determinata metrica per ogni file passatogli, ma non dà il risultato globale, che però può essere a sua volta calcolato con i dati a disposizione.
Un esempio di ciò è avvenuto per la metrica di Halstead: il tool usato calcola le metriche della suite analizzando i singoli files, ma non dà informazioni su tutto il progetto. In questo caso è stata creata una funzione *helper* che prende, per tutti i files analizzati, gli operands e gli operators trovati dal tool e computa tutti i valori delle metriche di quella suite.

Si parla invece di calcolare nuove metriche quando queste si possono computare dai valori già presenti. Queste metriche sono definite come *secondarie* in quanto derivate da altre, calcolate direttamente analizzando il codice.

Ne è un esempio la metrica WMC (Weighted Method Count) di McCabe, che viene computata utilizzando i valori della Complessità Ciclomatica delle singole funzioni analizzate.

Capitolo 4

Porting in Rust e Confronto

4.1 Porting delle metriche

Dopo aver creato l'aggregatore e averlo predisposto per l'analisi di codici C e C++, si sono cercati degli analizzatori per i codici sorgenti scritti in Rust. Sono stati trovati solo tre programmi in grado di computare delle metriche di manutenibilità per questo linguaggio, uno di questi è Tokei, già presentato nel capitolo precedente. Il secondo programma è Clippy[1], un tool di *linting* scritto in Rust, modulare e ben mantenuto dalla community. È stato preso in considerazione perché, fra le altre cose, calcola la Cyclomatic Complexity. Infine è stato trovato Rust-Code-Analysis, un *crate* (libreria) Rust di recente creazione, capace di analizzare più linguaggi di programmazione, fra cui C++, Go, Java, JavaScript, Python e Rust. Originariamente scritto da sviluppatori di Mozilla, è stata fatta una fork del progetto originale per continuare agevolmente allo sviluppo del programma secondo il nostro studio. Sono già stati contattati gli sviluppatori originali e, man mano che verranno implementate le modifiche, si faranno delle Pull Requests sul loro repository per chiedere loro di aggiungere le nuove funzioni al tool originale.

[1] https://github.com/rust-lang/rust-clippy

4.1.1 I problemi riscontrati

Delle metriche più utilizzate in letteratura, trovate e definite nel Capitolo 2, ben poche erano disponibili per Rust. Le altre metriche, quindi, si dovevano implementare ex novo. Prima di fare il porting di una metrica su un nuovo linguaggio, però, bisogna prima considerare se è fattibile o meno farlo e porsi domande sul come. Ad esempio, cercare di implementare la suite di C&K per linguaggi non orientati ad oggetti non ha senso; così come una metrica equivalente alla JLOC non può essere implementata per linguaggi che non supportino un generatore di documentazione.

Per quanto riguarda metriche come LOC e CLOC, metriche che calcolano il volume di un codice o di parti di esso, queste sono possibili da portare in Rust ed ha senso farlo.
È possibile, ed ha senso, implementare una metrica equivalente a JavaDoc LOC (JLOC), poiché Rust ha un generatore di documentazione ufficiale, *rustdoc*. In questo linguaggio le righe di documentazione si distinguono dai commenti perché iniziano con 3 slashes.
La Cyclomatic Complexity di McCabe può essere implementata poiché si tratta di calcolare i path indipendenti di esecuzione di un programma; La Weighted Method Count somma le complessità di tutte le funzioni implementate nel codice in analisi, quindi è implementabile.
Sebbene la suite di Halstead sia già presente nel tool Rust Code Analysis, volendo comunque svolgere una analisi a livello teorico, si può constatare come abbia senso implementare questa metrica, che ha come unico requisito quello di poter individuare tutti gli *operators* e gli *operands* in un codice.
Si può anche fare il porting di Number of Methods (NOM) e Number of Public Methods (NPM), in quanto vi è il concetto di funzione privata e pubblica. Il Maintainability Index è composta da metriche che abbiamo visto poco sopra essere computabili anche per questo linguaggio, quindi è possibile implementarla per questo linguaggio.

Le metriche Ce, MPC, LCOM2 e la suite di C&K sono metriche che misurano proprietà legate alle classi, quindi richiedono linguaggi orientati ad oggetti. Rust, però, è un caso particolare, in quanto è sia considerabile come linguaggio orientato ad oggetti, che non.[2] È considerabile come tale poiché è possibile avere un approccio ad oggetti utilizzando *struct* e blocchi *impl* per

[2] https://doc.rust-lang.org/book/ch17-00-oop.html

implementare gli i dati contenuti negli oggetti e i metodi ad essi associati. Allo stesso tempo, non ha dei veri e propri oggetti, come possono essere le classi in C++, che contengono sia i dati che i metodi e non ha il concetto di *ereditarietà*, ma usa i *traits*.
Per implementare queste ultime metriche, quindi, bisognerà capire come adattarle ad un linguaggio avente queste particolari caratteristiche. Metriche come Ce e MPC potrebbero essere implementabili senza particolari accorgimenti, ma per implementare la suite C&K bisognerà prendere in considerazione l'inesistenza dell'ereditarietà, sostituita dai traits.

4.1.2 Rust Code Analysis

Rust Code Analysis[3] si basa sulla libreria Tree-sitter[4], che parsifica il codice passatogli e costruisce un albero di sintassi. Con questo albero si possono andare a svolgere tutte quelle analisi sul codice che servono a calcolare le varie metriche.
Si possono, ad esempio, cercare nei vari nodi espressioni come l' If, l'Else, il For, il While, il Case ed il Catch per poter contare quanti possibili path ha una particolare funzione e definire, in questo modo, la sua Complessità Ciclomatica.
Si possono individuare le varie classi e calcolare il numero di funzioni al suo interno; o calcolare il numero totale di funzioni presenti nel progetto. Anche calcolare suites come Halstead non risulta particolarmente complesso, poiché si possono facilmente trovare tutti gli *operators* e gli *operands* del codice analizzato. Una volta che si hanno questi numeri risulta immediato computare i valori delle singole metriche appartenenti al set.

Al momento il programma può analizzare soltanto un file per volta e il suo output viene salvato in un file in formato Json, che può essere facilmente parsificato e trasformato seguendo l'output standard definito nella Sezione 3.1.2.

4.2 Confrontare i codici

Una volta trovati e sviluppati i softwares per calcolare le metriche di interesse, sia quelli per analizzare i codici scritti in C, C++ che quelli scritti in Rust,

[3] https://github.com/SoftengPoliTo/rust-code-analysis

[4] https://tree-sitter.github.io/tree-sitter

non resta che analizzare dei softwares e confrontare i risultati ottenuti.

Affinché il confronto abbia un senso, i codici da analizzare devono essere equivalenti. Bisogna, quindi, scegliere un algoritmo, una funzione o un programma e scriverlo sia in C/C++, se non si ha già il codice in questi linguaggi, che in Rust.

Utilizzando l'aggregatore descritto nel Capitolo 3 si avranno i risultati in files Json che seguono il formato standard precedentemente definito, quindi il confronto risulterà essere più facile da effettuare. Un confronto manuale è possibile, sebbene fattibile solo nel caso in cui si stiano confrontando codici molto brevi e composti da pochi files. Il modo più semplice per effettuarlo è quello di creare uno script.

4.2.1 Paragonare codici equivalenti

Le singole metriche ritornano un numero, che può essere intero o decimale. Le suites invece possono essere viste come un insieme di metriche, ovvero un contenitore che comprende dei valori da analizzare separatamente, ma che concorrono a dare un'idea generale sul codice analizzato.

Confrontare una metrica presente nel Json è facile, ma richiede di sapere come interpretare i risultati della metrica in questione. Per la metrica LOC, ad esempio, maggiore è il numero ritornato, maggiore è la lunghezza del codice in analisi e, secondo la teoria, minore è la sua manutenibilità. Sapendo questo è naturale concludere che, per quella metrica, il codice migliore è quello avente LOC minore.
Per metriche come CLOC e JLOC (anche se questa metrica è specifica per Java, è un buon esempio) vale il contrario, poiché maggiori sono le linee di commenti o di documentazione, maggiore è la leggibilità del codice. Quindi sarà migliore il software avente CLOC maggiore. Per quanto riguarda il M.I., minore è il numero ritornato, minore è la manutenibilità del codice. Confrontare automaticamente le metriche nella *sezione globale* è sempre possibile ed offre una visuale generale sui due codici.

Il confronto fra i singoli files o fra le singole funzioni dei due progetti, invece, è più complicato. Per poterli paragonare in modo automatico, infatti, bisogna trovare un modo per riconoscere le funzioni equivalenti. Una soluzione potrebbe essere, per quanto possibile, quella di mantenere nei due progetti gli stessi nomi sia per i files che per le funzioni.
Non tutte le metriche potranno essere calcolate e utilizzate per tutti i linguaggi. Ad esempio, per linguaggi come il C, che non sono orientati ad

oggetti, non si potranno calcolare metriche pensate per quel paradigma; in più, paragonando un codice C ad uno scritto in C++, la struttura dei due programmi potrebbe essere diversa, dato il diverso approccio tenuto durante le fasi di programmazione.

4.2.2 Confronto tramite script

Come detto prima, il confronto manuale è facile da effettuare solo per codici di modeste dimensioni, poiché, con l'aumentare del numero di files e di funzioni, gli outputs crescerebbero rapidamente in lunghezza, diventando difficili da leggere. Per ovviare a questo problema si è creato uno script python, *metrics-diff*, che sfrutta la struttura ben definita dell'output Json per trovare le differenze fra due diversi codici analizzati.

Il comparatore accetta in input i path ai due json da paragonare ed una serie di argomenti opzionali che permettono di eseguire il confronto in vari modi e con vari livelli di granularità. È possibile infatti scegliere se comparare o meno la parte contenente le statistiche globali, i blocchi contenenti le metriche sui singoli files, oppure se abilitare il confronto fra singole funzioni. È anche possibile indicare solo un sottoinsieme di metriche da mettere a confronto, così da escludere eventuali risultati non pertinenti all'analisi che si vuole effettuare. I risultati del confronto vengono presentati in formato testuale sul terminale.

Capitolo 5

Conclusioni e Lavori futuri

5.1 Conclusioni

La manutenibilità del codice è un aspetto estremamente importante per sviluppare al meglio un progetto e in letteratura sono presentate innumerevoli metriche e programmi per stimare tale caratteristica. Data la complessità e l'alto numero di studi effettuati sull'argomento, vi sono opinioni diverse sulle varie metriche presentate, alcune delle quali in totale disaccordo con altre. Non vi sono, quindi, delle metriche che siano considerate univocamente utili ed adatte a tale scopo. Ciò è stato nuovamente confermato durante l'analisi della letteratura effettuata.

5.1.1 SLR

Nella SLR condotta all'inizio del lavoro, i cui punti salienti sono riportati nel Capitolo 2, si è cercato di ottenere una panoramica sulle metriche più comuni utilizzate in letteratura nell'ultimo ventennio e sui programmi utilizzati dagli autori degli articoli.

Metriche

Durante l'analisi della letteratura, effettuata seguendo le linee guida di Kitchenham et al.[54], sono state identificate un totale di 174 metriche, riportate

per intero nella Tabella 5.2. Le metriche citate solo in uno degli studi selezionati, la maggioranza di queste, non sono state prese in considerazione. Dopo aver dato un punteggio ad ogni metrica, basato sul numero di citazioni e sull'opinione positiva o meno degli autori degli studi che le citavano, alla fine sono state scelte quelle aventi punteggio e numero di citazioni sopra i valori soglia individuati. Sono state trovate un totale di 13 singole metriche e 2 suites di metriche:

- CC - McCabe's Cyclomatic Complexity
- CHANGE - Number of Lines Changed in Class
- *C&K - Chidamber & Kemerer's Suite*
- CLOC - Comments Lines of Code
- *Halstead's Suite*
- JLOC - JavaDoc Lines of Code
- LOC - Lines of Code
- LCOM2 - Lack of Cohesion in Methods
- MI - Maintainability Index
- MPC - Message Passing Coupling
- NOM - Number of Methods
- NPM - Number of Public Methods
- STAT - Number of Statements
- WMC - McCabe's Weighted Methods Count

Di ognuna di queste metriche sono state riportate delle brevi descrizioni, le loro formule e, ove possibile, il riferimento al paper degli autori che le hanno ideate.

Programmi utilizzati in letteratura

Dopo aver trovato le metriche più comuni, abbiamo cercato i programmi utilizzati negli studi primari selezionati per calcolare le metriche di manutenibilità. Sono stati trovati un totale di 38 softwares, presentati in Tabella 2.6. Di questi, è stato possibile recuperarne solamente la metà. Di questi, 6 sono proprietari e 13 open-source. Gli altri 19 programmi citati non sono stati trovati poiché gli articoli che ne parlavano non offrivano informazioni

sufficienti per poterli individuare. Dei programmi trovati, alcuni non sono più attivamente mantenuti da più di un anno.
I programmi usati negli studi selezionati, comunque, rappresentano soltanto una piccola parte dei tools disponibili online per calcolare le metriche di manutenibilità. Quindi, sebbene sia utile sapere quali programmi siano stati utilizzati in letteratura, dato l'esiguo numero di softwares trovati, risulta utile fare delle ricerche online per capire quali altri programmi siano disponibili.
Vi sono alcuni softwares che supportano un gran numero di metriche e di linguaggi, ma la maggioranza può analizzare un numero esiguo di linguaggi (da uno a tre) e di metriche. Analizzandoli si può vedere come i linguaggi di programmazione più supportati siano il Java (supportato da 14 dei programmi in analisi), il C (da 11 programmi), il C++ (da 10 programmi), il JavaScript (da 8 programmi) e il C# (da 7 programmi).
Per quanto riguarda le metriche, LOC e CC sono le due più supportate, che possono essere calcolate da quasi tutti i tools trovati. In media, i programmi proprietari possono calcolare più metriche di quelli open-source, che spesso sono solo dei progetti di modeste dimensioni.

5.1.2 L'aggregatore

Nella seconda parte della tesi, descritta nel Capitolo 3, si è creato un programma atto a calcolare le metriche più usate in letteratura, trovate eseguendo la SLR. Il software sviluppato, che è stato scritto in python, esegue diversi analizzatori, parsifica i loro risultati, li riunisce in un unico output e computa delle ulteriori metriche secondarie.

Al momento della scrittura di questa tesi, gli analizzatori utilizzati dall'aggregatore sono: Tokei, Halstead Metrics Tool, CCCC e Maintainability Index Tool. L'aggregatore è stato progettato per essere modulare ed estensibile; non è difficile, quindi, aggiungere ulteriori programmi a quelli già presenti.
Inoltre, il software supporta potenzialmente qualsivoglia linguaggio di programmazione: basta che sia supportato da almeno uno degli analizzatori utilizzati.

Gli outputs dei vari analizzatori vengono standardizzati seguendo una struttura ben definita, che permette di ritornare i valori delle varie metriche in modo indipendente dal particolare programma utilizzato. I singoli risultati vengono riuniti in un output unico, in formato Json, standardizzato e facilmente interpretabile sia da persone che da eventuali programmi.

Dopo aver fuso tutti gli outputs in uno unico, standard, l'aggregatore computa e aggiunge delle metriche secondarie, ricavabili dagli altri dati. Un

esempio è Weighted Methods Count di McCabe (WMC), che somma assieme i valori della CC di tutti i metodi presenti in una Classe.

L'utilità di questo nuovo programma sta non tanto nell'automatizzare l'esecuzione dei vari analizzatori supportati, che comunque semplifica il lavoro agli utilizzatori, ma soprattutto nel ritornare i valori delle metriche computate in un formato standard, indipendente dagli specifici analizzatori utilizzati e dal linguaggio del codice analizzato. Le metriche vengono computate seguendo le definizioni dei loro ideatori, senza offrire un'interpretazione dei valori ottenuti. In questo modo sarà l'utilizzatore finale del programma a poter decidere come meglio interpretare ed utilizzare i risultati acquisiti, secondo le sue necessità.

L'output standardizzato

L'output prodotto dall'aggregatore è in Json, poiché è un formato human-readable, semplice da imparare e molto popolare. È supportato dalla maggioranza dei linguaggi di programmazione, che hanno librerie stabili ed efficienti per la parsificazione dei dati trasmessi in questo formato.

L'output dell'aggregatore contiene i risultati dei vari analizzatori utilizzati, rappresentati seguendo una struttura ben definita, riportata nel diagramma a blocchi in Figura 3.2. I dati sono organizzati in una struttura ad albero, contenente nel suo primo livello le metriche calcolate su tutti i files analizzati. Vi sono poi due liste di oggetti: *classes* e *files*. La prima contiene le informazioni e le metriche calcolate sulle classi presenti nel codice; può essere vuota, se il codice analizzato non ha classi. La seconda contiene tutte le metriche computate sui singoli files. Ogni file ha al suo interno, oltre alle metriche ad esso relative, una lista contenente tutte le metriche relative alle singole funzioni presenti in quel file.
Ogni metrica semplice è rappresentata da un oggetto Json String o Json Value, mentre le metriche appartenenti ad una suite sono raggruppate dentro un Json Object avente il nome della suite stessa.

Si possono trovare degli esempi di codici analizzati, e dei relativi risultati, al link https://doi.org/10.6084/m9.figshare.12011343.

5.1.3 Porting delle metriche in Rust

Online sono stati trovati solo tre programmi in grado di computare delle metriche per codici scritti in Rust: Tokei, che calcola delle metriche di volume (LOC, CLOC); Clippy, un tool di linting che computa anche la CC; Rust-Code-Analysis, un nuovo analizzatore sviluppato da Mozilla.

Si è deciso di utilizzare Rust-Code-Analysis poiché è quello più promettente, dato che si basa sulla libreria Tree-sitter per parsificare vari linguaggi di programmazione ed ottenere un albero di sintassi che può essere facilmente analizzabile in cerca dei dati necessari.

Prima di fare il porting di una metrica in un nuovo linguaggio ci si deve chiedere se ha senso farlo, in quanto alcune metriche sono legate a particolari aspetti di un linguaggio, come la JLOC, o ad un particolare paradigma, come la suite di C&K, che è stata pensata per i linguaggi orientati ad oggetti.

5.1.4 Come confrontare le metriche

Una volta raggruppati o sviluppati gli analizzatori per i linguaggi di interesse, si può pensare di confrontarli per capire quali di quelli aiutano a tenere alta la manutenibilità dei softwares scritti. Affinché il confronto fra questi linguaggi abbia un senso, i codici da analizzare devono essere equivalenti e i risultati degli analizzatori utilizzati dovranno essere comparabili. Il confronto risulta più semplice da effettuare utilizzando l'Aggregatore descritto nel Capitolo 3, in quanto i risultati saranno in un formato compatibile. Sebbene sia possibile effettuare il paragone manualmente, è consigliabile utilizzare uno script che automatizzi l'operazione, soprattutto se i codici presi in considerazione sono abbastanza grandi.

Bisogna tenere a mente che non tutte le metriche potrebbero essere calcolabili per entrambi i linguaggi che si vogliono paragonare e che, mentre confrontare le metriche nella sezione globale del Json è sempre possibile, confrontare i singoli files o le singole funzioni può non essere sempre fattibile.

5.2 Lavori futuri

Uno degli obiettivi futuri è quello di migliorare ulteriormente l'Aggregatore creato, correggendo eventuali difetti, eseguendo le analisi in parallelo ed integrando al suo interno altri analizzatori. Così facendo sarà possibile analizzare più linguaggi di programmazione ed aggiungere ulteriori metriche, fornendo

risultati più completi. Inoltre, avendo più dati su cui lavorare, sarà possibile ricavare ulteriori metriche derivate.
Grazie alla flessibilità della struttura del nostro output, unita alle caratteristiche del formato Json, gli script preesistenti creati dagli utilizzatori del nostro software non necessiteranno di modifiche dopo eventuali aggiornamenti o aggiunte di analizzatori nell' Aggregatore.

Per rendere più fruibile il programma, sarebbe utile fornire un'interfaccia grafica che faciliti la visualizzazione dei risultati ottenuti.

Un secondo obiettivo futuro è quello di completare l'analizzatore per Rust e fare il porting di quante più metriche possibile, ponendo particolare attenzione a come integrare le metriche studiate per i linguaggi orientati ad oggetti. Oltre a poterlo utilizzare come programma a sé stante, questo analizzatore sarà utilizzabile tramite l'Aggregatore.

Una volta fatto il porting delle metriche, si potranno eseguire ulteriori test, per ottenere risultati più precisi riguardo alla manutenibilità dei progetti scritti in Rust e, in special modo fare dei paragoni con quelli scritti in C.

Bibliografia

[1] W. Claes, *Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering* , Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 2014, 10.1145/2601248.2601268, https://doi.org/10.1145/2601248.2601268

[2] P. Jacso, *Calculating the h-index and other bibliometric and scientometric indicators from Google Scholar with the Publish or Perish software*, Online Information Review, 2009, 10.1108/14684520911011070

[3] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidi, A. Goeb, J. Streit, *The Quamoco Product Quality Modelling and Assessment Approach*, Proceedings - International Conference on Software Engineering, 2012, 10.1109/ICSE.2012.6227106

[4] D. Roberts, W. Opdyke, K. Beck, M. Fowler, J. Bran, *Refactoring: improving the Design of Existing Code*, Addison-Wesley Professional, 1999

[5] N. Barbosa and K. Hirama, *Assessment of Software Maintainability Evolution Using C K Metrics* , 2013, 10.1109/TLA.2013.6684398

[6] A. Kaur and K. Kaur and K. Pathak, *Software maintainability prediction by data mining of software code metrics* , 2014 International Conference on Data Mining and Intelligent Computing (ICDMIC), 2014, 10.1109/ICDMIC.2014.6954262

[7] A. Tahir and R. Ahmad, *An AOP-Based Approach for Collecting Software Maintainability Dynamic Metrics* , 2010 Second International Conference on Computer Research and Development, 2010, 10.1109/ICCRD.2010.26

[8] A. Kaur and K. Kaur and K. Pathak, *A proposed new model for maintainability index of open source software* , Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization, 2014, 10.1109/ICRITO.2014.7014758

[9] Y. Tian and C. Chen and C. Zhang, *AODE for Source Code Metrics for Improved Software Maintainability* , 2008 Fourth International Conference on Semantics, Knowledge and Grid, 2008, 10.1109/SKG.2008.43

[10] I. Kádár and P. Hegedus and R. Ferenc and T. Gyimóthy, *A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability* , 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, 10.1109/SANER.2016.42

[11] J. Ludwig and S. Xu and F. Webber, *Compiling static software metrics for reliability and maintainability from GitHub repositories* , 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017, 10.1109/SMC.2017.8122569

[12] L. Wang and X. Hu and Z. Ning and W. Ke, *Predicting Object-Oriented Software Maintainability Using Projection Pursuit Regression* , 2009 First International Conference on Information Science and Engineering, 2009, 10.1109/ICISE.2009.845

[13] N. Narayanan Prasanth and S. Ganesh and G. Arul Dalton, *Prediction of maintainability using software complexity analysis: An extended FRT* , 2008 International Conference on Computing, Communication and Networking, 2008, 10.1109/ICCCNET.2008.4787774

[14] J. Ostberg and S. Wagner, *On Automatically Collectable Metrics for Software Maintainability Evaluation* , 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, 2014, 10.1109/IWSM.Mensura.2014.19

[15] M. Saboe, *The use of software quality metrics in the materiel release process experience report* , Proceedings Second Asia-Pacific Conference on Quality Software, 2001, 10.1109/APAQS.2001.990008

[16] G. Kaur and B. Singh, *Improving the quality of software by refactoring* , 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), 2017, 10.1109/ICCONS.2017.8250707

[17] Sjøberg, Dag I.K. and Anda, Bente and Mockus, Audris, *Questioning Software Maintenance Metrics: A Comparative Case Study* , Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM2012, 10.1145/2372251.2372269, http://doi.acm.org/10.1145/2372251.2372269

[18] A. Hindle and M. W. Godfrey and R. C. Holt, *Reading Beside the Lines: Indentation as a Proxy for Complexity Metric* , 2008

16th IEEE International Conference on Program Comprehension, 2008, 10.1109/ICPC.2008.13

[19] A. Jain and S. Tarwani and A. Chug, *An empirical investigation of evolutionary algorithm for software maintainability prediction* , 2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS), 2016, 10.1109/SCEECS.2016.7509314

[20] S. Misra and A. Adewumi and L. Fernandez-Sanz and R. Damasevicius, *A Suite of Object Oriented Cognitive Complexity Metrics* , 2018, 10.1109/ACCESS.2018.2791344

[21] P. Vytovtov and E. Markov, *Source code quality classification based on software metrics* , 2017 20th Conference of Open Innovations Association (FRUCT), 2017, 10.23919/FRUCT.2017.8071355

[22] M. Wahler and U. Drofenik and W. Snipes, *Improving Code Maintainability: A Case Study on the Impact of Refactoring* , 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, 10.1109/ICSME.2016.52

[23] M. Pizka, *Code Normal Forms* , 29th Annual IEEE/NASA Software Engineering Workshop, 2005, 10.1109/SEW.2005.22

[24] D. Threm and L. Yu and S. Ramaswamy and S. D. Sudarsan, *Using normalized compression distance to measure the evolutionary stability of software systems* , 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015, 10.1109/ISSRE.2015.7381805

[25] T. L. Alves and C. Ypma and J. Visser, *Deriving metric thresholds from benchmark data* , 2010 IEEE International Conference on Software Maintenance, 2010, 10.1109/ICSM.2010.5609747

[26] M. I. Sarwar and W. Tanveer and I. Sarwar and W. Mahmood, *A comparative study of MI tools: Defining the Roadmap to MI tools standardization* , 2008 IEEE International Multitopic Conference, 2008, 10.1109/INMIC.2008.4777767

[27] N. E. Gold and A. M. Mohan and P. J. Layzell, *Spatial complexity metrics: an investigation of utility* , 2005, 10.1109/TSE.2005.39

[28] A. F. Yamashita and H. C. Benestad and B. Anda and P. E. Arnstad and D. I. K. Sjoberg and L. Moonen, *Using concept mapping for maintainability assessments* , 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, 10.1109/ESEM.2009.5314234

[29] H. Liu and X. Gong and L. Liao and B. Li, *Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution* , 2018

IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, 10.1109/COMPSAC.2018.10332

[30] L. M. d. Silva and F. Dantas and G. Honorato and A. Garcia and C. Lucena, *Detecting Modularity Flaws of Evolving Code: What the History Can Reveal?* , 2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse, 2010, 10.1109/SBCARS.2010.11

[31] R. Gonçalves and I. Lima and H. Costa, *Using TDD for developing object-oriented software — A case study* , 2015 Latin American Computing Conference (CLEI), 2015, 10.1109/CLEI.2015.7360030

[32] K. Chatzidimitriou and M. Papamichail and T. Diamantopoulos and M. Tsapanos and A. Symeonidis, *npm-Miner: An Infrastructure for Measuring the Quality of the npm Registry* , 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), 2018

[33] J. Gil and M. Goldstein and D. Moshkovich, *An empirical investigation of changes in some software properties over time* , 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, 10.1109/MSR.2012.6224285

[34] S. Rongviriyapanish and T. Wisuttikul and B. Charoendouysil and P. Pitakket and P. Anancharoenpakorn and P. Meananeatra, *Changeability prediction model for java class based on multiple layer perceptron neural network* , 2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2016, 10.1109/ECTICon.2016.7561392

[35] M. Yan and X. Zhang and C. Liu and J. Zou and L. Xu and X. Xia, *Learning to Aggregate: An Automated Aggregation Method for Software Quality Model* , 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, 10.1109/ICSE-C.2017.139

[36] Y. Ma and K. He and B. Li and Xiaoyan Zhou, *How multiple-dependency structure of classes affects their functions a statistical perspective* , 2010 2nd International Conference on Software Technology and Engineering, 2010, 10.1109/ICSTE.2010.5608763

[37] Jermakovics, Andrejs and Moser, Raimund and Sillitti, Alberto and Succi, Giancarlo, *Visualizing Software Evolution with Lagrein* , Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, ACM2008, 10.1145/1449814.1449843, http://doi.acm.org/10.1145/1449814.1449843

[38] Ludwig, Jeremy and Xu, Steven and Webber, Frederick, *Static Software Metrics for Reliability and Maintainability* , Proceedings of the 2018 International Conference on Technical Debt, ACM2018, 10.1145/3194164.3194184, http://doi.acm.org/10.1145/3194164.3194184

[39] Arshad, Shumail and Tjortjis, Christos, *Clustering Software Metric Values Extracted from C# Code for Maintainability Assessment* , Proceedings of the 9th Hellenic Conference on Artificial Intelligence, ACM2016, 10.1145/2903220.2903252, http://doi.acm.org/10.1145/2903220.2903252

[40] Curtis, Bill and Sappidi, Jay and Subramanyam, Jitendra, *An Evaluation of the Internal Quality of Business Applications: Does Size Matter?* , Proceedings of the 33rd International Conference on Software Engineering, ACM2011, 10.1145/1985793.1985893, http://doi.acm.org/10.1145/1985793.1985893

[41] Lee, Young and Chang, Kai H., *Reusability and Maintainability Metrics for Object-oriented Software* , Proceedings of the 38th Annual on Southeast Regional Conference, ACM2000, 10.1145/1127716.1127737, http://doi.acm.org/10.1145/1127716.1127737

[42] Chávez, Alexander and Ferreira, Isabella and Fernandes, Eduardo and Cedrim, Diego and Garcia, Alessandro , *How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study* , Proceedings of the 31st Brazilian Symposium on Software Engineering, ACM2017, 10.1145/3131151.3131171, http://doi.acm.org/10.1145/3131151.3131171

[43] Mamun, Md Abdullah Al and Berger, Christian and Hansson, Jörgen, *Correlations of Software Code Metrics: An Empirical Study* , Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, ACM2017, 10.1145/3143434.3143445, http://doi.acm.org/10.1145/3143434.3143445

[44] Chhillar, Rajender Singh and Gahlot, Sonal, *An Evolution of Software Metrics: A Review* , Proceedings of the International Conference on Advances in Image Processing, ACM2017, 10.1145/3133264.3133297, http://doi.acm.org/10.1145/3133264.3133297

[45] Matsushita, Tsubasa and Sasano, Isao, *Detecting Code Clones with Gaps by Function Applications* , Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation,

ACM2017, 10.1145/3018882.3018892, http://doi.acm.org/10.1145/3018882.3018892

[46] Bohnet, Johannes and Döllner, Jürgen, *Monitoring Code Quality and Development Activity by Software Maps* , Proceedings of the 2Nd Workshop on Managing Technical Debt, ACM2011, 10.1145/1985362.1985365, http://doi.acm.org/10.1145/1985362.1985365

[47] Sinha, Bhaskar Raj and Dey, Pradip Peter and Amin, Mohammad and Badkoobehi, Hassan, *Software Complexity Measurement Using Multiple Criteria* , Consortium for Computing Sciences in Colleges2013, http://dl.acm.org/citation.cfm?id=2458539.2458566

[48] , *IEEE Standard Glossary of Software Engineering Terminology* , 1990, 10.1109/IEEESTD.1990.101064

[49] T. J. McCabe, *A Complexity Measure* , 1976, 10.1109/TSE.1976.233837

[50] S. R. Chidamber and C. F. Kemerer, *A metrics suite for object oriented design* , 1994, 10.1109/32.295895

[51] Li, Wei and Henry, Sallie, *Object-oriented Metrics That Predict Maintainability* , Elsevier Science Inc.1993, 10.1016/0164-1212(93)90077-B, http://dx.doi.org/10.1016/0164-1212(93)90077-B

[52] Halstead, Maurice H., *Elements of Software Science (Operating and Programming Systems Series)* , Elsevier Science Inc.1977

[53] P. Oman and J. Hagemeister, *Metrics for assessing a software system's maintainability* , Proceedings Conference on Software Maintenance 1992, 1992, 10.1109/ICSM.1992.242525

[54] Barbara Kitchenham and O. Pearl Brereton and David Budgen and Mark Turner and John Bailey and Stephen Linkman, *Systematic literature reviews in software engineering – A systematic literature review* , 2009, https://doi.org/10.1016/j.infsof.2008.09.009, http://www.sciencedirect.com/science/article/pii/S0950584908001390

[55] Barbara A, Kitchenham and Charters, Stuart, *Guidelines for performing Systematic Literature Reviews in Software Engineering* , 2007

[56] Herraiz, Israel and Gonzalez-Barahona, Jesus and Robles, Gregorio, *Towards a Theoretical Model for Software Growth* , 2007, 10.1109/MSR.2007.31

[57] D. Spinellis, *Tool writing: a forgotten art? (software tools)* , 2005, 10.1109/MS.2005.111

[58] G. Jay, J. Hale, R. Smith, D. Hale, N. Kraft, C. Ward , *Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship* , 2009, 10.4236/jsea.2009.23020

Appendix

Tabella 5.1: Studi Selezionati

ID	Autori	Titolo	Anno	*Score*
[5]	N. Barbosa, K. Hirama	Assessment of Software Maintainability Evolution Using C K Metrics	2013	5
[6]	A. Kaur, K. Kaur, K. Pathak	Software maintainability prediction by data mining of software code metrics	2014	5
[7]	A. Tahir, R. Ahmad	An AOP-Based Approach for Collecting Software Maintainability Dynamic Metrics	2010	4
[8]	A. Kaur, K. Kaur, K. Pathak	A proposed new model for maintainability index of open source software	2014	4
[9]	Y. Tian, C. Chen, C. Zhang	AODE for Source Code Metrics for Improved Software Maintainability	2008	5
[10]	I. Kádár, P. Hegedus, R. Ferenc, T. Gyimòthy	A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability	2016	4
[11]	J. Ludwig, S. Xu, F. Webber	Compiling static software metrics for reliability and maintainability from GitHub repositories	2017	5
[12]	L. Wang, X. Hu, Z. Ning, W. Ke	Predicting Object-Oriented Software Maintainability Using Projection Pursuit Regression	2009	4
[13]	N. Narayanan Prasanth, S. Ganesh, G. Arul Dalton	Prediction of maintainability using software complexity analysis: An extended FRT	2008	4

Proseguimento della Tabella: Studi Selezionati

ID	Autori	Titolo	Anno	*Score*
[14]	J. Ostberg, S. Wagner	On Automatically Collectable Metrics for Software Maintainability Evaluation	2014	4
[15]	M. Saboe	The use of software quality metrics in the materiel release process experience report	2001	4
[16]	G. Kaur, B. Singh	Improving the quality of software by refactoring	2017	4
[17]	D. I. K. Sjøberg, B. Anda, A. Mockus	Questioning Software Maintenance Metrics: A Comparative Case Study	2012	5
[18]	A. Hindle, M. W. Godfrey, R. C. Holt	Reading Beside the Lines: Indentation as a Proxy for Complexity Metric	2008	4
[19]	A. Jain, S. Tarwani, A. Chug	An empirical investigation of evolutionary algorithm for software maintainability prediction	2016	4
[20]	S. Misra, A. Adewumi, L. Fernandez-Sanz, R. Damasevicius	A Suite of Object Oriented Cognitive Complexity Metrics	2018	5
[21]	P. Vytovtov, E. Markov	Source code quality classification based on software metrics	2017	4
[22]	M. Wahler, U. Drofenik, W. Snipes	Improving Code Maintainability: A Case Study on the Impact of Refactoring	2016	4
[23]	M. Pizka	Code Normal Forms	2005	4
[24]	D. Threm, L. Yu, S. Ramaswamy, S. D. Sudarsan	Using normalized compression distance to measure the evolutionary stability of software systems	2015	4
[25]	T. L. Alves, C. Ypma, J. Visser	Deriving metric thresholds from benchmark data	2010	4
[26]	M. I. Sarwar, W. Tanveer, I. Sarwar, W. Mahmood	A comparative study of MI tools: Defining the Roadmap to MI tools standardization	2008	4
[27]	N. E. Gold, A. M. Mohan, P. J. Layzell	Spatial complexity metrics: an investigation of utility	2005	4
[28]	A. F. Yamashita, H. C. Benestad, B. Anda, P. E. Arnstad, D. I. K. Sjøberg, L. Moonen	Using concept mapping for maintainability assessments	2009	5

Proseguimento della Tabella: Studi Selezionati

ID	Autori	Titolo	Anno	*Score*
[29]	H. Liu, X. Gong, L. Liao, B. Li	Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution	2018	5
[30]	L. M. D. Silva, F. Dantas, G. Honorato, A. Garcia, C. Lucena	Detecting Modularity Flaws of Evolving Code: What the History Can Reveal?	2010	4
[31]	R. Gonçalves, I. Lima, H. Costa	Using TDD for developing object-oriented software — A case study	2015	4
[32]	K. Chatzidimitriou, M. Papamichail, T. Diamantopoulos, M. Tsapanos, A. Symeonidis	npm-Miner: An Infrastructure for Measuring the Quality of the npm Registry	2018	4
[33]	J. Gil, M. Goldstein, D. Moshkovich	An empirical investigation of changes in some software properties over time	2012	4
[34]	S. Rongviriyapanish, T. Wisuttikul, B. Charoendouysil, P. Pitakket, P. Anancharoenpakorn, P. Meananeatra	Changeability prediction model for java class based on multiple layer perceptron neural network	2016	4
[35]	M. Yan, X. Zhang, C. Liu, J. Zou, L. Xu, X. Xia	Learning to Aggregate: An Automated Aggregation Method for Software Quality Model	2017	4
[36]	Y. Ma, K. He, B. Li, Xiaoyan Zhou	How multiple-dependency structure of classes affects their functions a statistical perspective	2010	4
[37]	A. Jermakovics, R. Moser, A. Sillitti, G. Succi	Visualizing Software Evolution with Lagrein	2008	4
[38]	J. Ludwig, S. Xu, F. Webber	Static Software Metrics for Reliability and Maintainability	2018	5
[39]	S. Arshad, C. Tjortjis	Clustering Software Metric Values Extracted from C# Code for Maintainability Assessment	2016	4
[40]	B. Curtis. J. Sappidi, J. Subramanyam	An Evaluation of the Internal Quality of Business Applications: Does Size Matter?	2011	4

Proseguimento della Tabella: Studi Selezionati				
ID	**Autori**	**Titolo**	**Anno**	***Score***
[41]	Y. Lee, K. H. Chang	Reusability and Maintainability Metrics for Object-oriented Software	2000	4
[43]	M. A. A. Mamun, C. Berger, J.Hansson	Correlations of Software Code Metrics: An Empirical Study	2017	5
[44]	R. S. Chhillar, S. Gahlot	An Evolution of Software Metrics: A Review	2017	4
[45]	T. Matsushita, I. Sasano	Detecting Code Clones with Gaps by Function Applications	2017	4
[46]	J. Bohnet, J. Döllner	Monitoring Code Quality and Development Activity by Software Maps	2011	4
[47]	B. R. Sinha, P. P. Dey, M. Amin, H. Badkoobehi	Software Complexity Measurement Using Multiple Criteria	2013	4
[42]	A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, A. Garcia	How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study	2017	4
Fine della Tabella: Studi Selezionati				

Tabella 5.2: Metriche - Studi (Completa)

Inizio della Tabella:Metriche - Studi (Completa)				
Suite	**Metrica**	**Studi che la usano**	**Cit.**	**Score**
-	LOC, Lines of Code	[5], [8], [9], [11], [14], [15], [17], [18], [19], [25], [26], [43], [44], [42]	14	10
-	NOM, No. of Methods	[5], [12], [34], [43]	4	4
-	CHANGE, No. of Lines Changed in the Class	[5], [6], [8], [12]	4	4
-	Jensen's Nf	[9]	1	1
-	STAT, No. of Statements	[6], [43], [8], [42]	4	4
-	No. of Query	[6], [8]	2	2
Chidamber & Kemerer	WMC, Weighted Methods per Class	[5], [6], [12], [17], [19], [20], [25], [29], [33], [35], [39], [44], [47]	13	11
Chidamber & Kemerer	DIT, Depth of Inheritance Tree	[5], [6], [12], [14], [17], [19], [20], [25], [33], [34], [35], [39], [44], [47], [42]	15	13
Chidamber & Kemerer	NOC, Number of Children	[5], [6], [12], [14], [17], [20], [25], [33], [35], [39], [44], [47], [42]	13	11
Chidamber & Kemerer	CBO,Coupling Between Objects	[5], [6], [8], [11], [14], [17], [19], [20], [25], [33], [34], [35], [36], [39], [44], [47], [42]	17	15

Continuazione della Tabella: Metriche - Studi (Completa)

Suite	Metrica	Studi che la usano	Cit.	Score
Chidamber & Kemerer	RFC, Response for Class	[5], [6], [10], [11], [12], [14], [17], [19], [20], [25], [33], [35], [39], [41], [44], [47]	16	14
Chidamber & Kemerer	LCOM, Lack of Cohesion in Methods	[5], [6], [12], [14], [17], [20], [25], [33], [34], [35], [36], [41], [44], [47]	14	12
Chidamber & Kemerer	LCOM2, Lack of Cohesion in Methods	[19], [39], [42]	3	3
Chidamber & Kemerer	NPM, Number of Public Methods	[6], [19], [34], [39]	4	4
Li & Henry (L&H)	NOC, Number of Children	[8]	1	1
Li & Henry (L&H)	MPC, Message Passing Coupling	[8], [12], [35], [41]	4	4
Li & Henry (L&H)	RFC, Response for a Class	[8]	1	1
Li & Henry (L&H)	LCOM, Lack of Cohesion in Methods	[8]	1	1
Li & Henry (L&H)	WMC, Weighted method per class	[8]	1	1
Li & Henry (L&H)	SLOC, Source Lines of Code	[8]	1	1
Li & Henry (L&H)	DAC	[12], [35]	2	2
Li & Henry (L&H)	SIZE2	[35]	1	1
Li & Henry (L&H)	NOM	[35]	1	1
Li & Henry (L&H)	DIT, Depth of Inheritance Tree	[8]	1	1
Halstead	Halstead Vocabulary (n)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
Halstead	Halstead Volume (V)	[6], [8], [14], [15], [17], [18], [21], [26], [44], [47]	10	8
Halstead	Halstead Bugs (B)	[6], [8], [14], [21], [44], [47]	6	4
Halstead	Halstead Length (N)	[6], [8], [9], [14], [15], [18], [21], [44], [47]	9	7
Halstead	Halstead Difficulty (D)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
Halstead	Halstead Effort (E)	[6], [8], [14], [15], [18], [21], [44], [47]	8	6
-	Dcy, Number of Dependencies	[6]	1	1
-	Ca, Afferent Coupling	[6], [19]	2	2
-	Ce, Efferent Coupling	[6], [8], [19]	3	3
-	Number of attributes Added	[6]	1	1
-	Number of Commands	[6], [8]	2	2
-	CLOC, Comments Lines of Code	[6], [8], [9], [17], [34], [42]	6	6
-	NOOC, Number of operations overridden	[6]	1	1
-	JLOC, JavaDoc lines of Code	[6], [8], [34]	3	3
-	NOAC, Number of Operations added	[6], [8]	2	2
-	PDcy, Number of package dependencies	[6], [8]	2	2
-	CONS, Number of Constructors	[6], [8]	3	2
-	Level, Level order	[6]	1	1
-	Dcy*, Number of transitive dependencies	[6]	1	1
-	Level*, Level order	[6]	1	1
-	Inner*, Number of inner classes	[6]	1	1
-	CSOA, Class Size (Operations +Attributes)	[6], [8]	2	2
-	CSA, Class Size(Attributes)	[6], [8]	2	2
-	Dpt, Number of dependents	[6]	1	1
-	CSO, Class Size (operations)	[6], [8]	2	2

Continuazione della Tabella: Metriche - Studi (Completa)

Suite	Metrica	Studi che la usano	Cit.	Score
-	Avg cc, Average cyclomatic complexity	[6]	2	2
-	AODE, Aggregating One-Dependence Estimators	[9]	1	1
-	CC, McCabe's Cyclomatic Complexity	[9], [14], [15], [17], [18], [21], [25], [26], [29], [39], [43], [44] [47], [42]	14	12
-	WMC, McCabe's Weighted Method Count	[8], [11], [14], [16], [34], [36]	6	6
-	McCabe's Weighted Method Count - Unweighted	[11]	1	1
-	Bandwidth	[9]	1	1
-	Number of Comment Characters	[9]	1	1
-	Numer of Code Characters	[9]	1	1
-	Total Number of Characters	[9]	1	1
-	Code to Comment Ratio	[11], [14]	2	2
-	MI, Maintainability Index	[8], [14], [15], [17], [26]	5	3
Misra's metrics	MC, Method Complexity	[20]	1	1
Misra's metrics	CWC, Coupling Weight for a Class	[20]	1	1
Misra's metrics	AC, Attribute Complexity	[20]	1	1
Misra's metrics	CLC, Class Complexity	[20]	1	1
Misra's metrics	CC, Code Complexity	[20]	1	1
Misra's metrics	AMC, Average Method Complexity	[20]	1	1
Misra's metrics	AMCC, Average Method Complexity per Class	[20]	1	1
Misra's metrics	ACC, Average Class Complexity	[20]	1	1
Misra's metrics	ACF, Average Coupling Factor	[20]	1	1
Misra's metrics	AAC, Average Attributes per Class	[20]	1	1
Structural Measures	OMMIC, Coupling	[17]	1	1
Structural Measures	TCC, Tight class Cohesion	[17], [42]	2	2
Structural Measures	WMC1, Size of Classes	[17]	1	1
(Code Smells)	Feature Envy (# per KLOC of code)	[17]	1	1
(Code Smells)	God Class (# per KLOC of code)	[17]	1	1
-	Non commenting lines of code	[43]	1	1
-	Non commenting lines of new code	[43]	1	1
-	Number of classes (including nested classes, interfaces, enums and annotations)	[43], [42]	2	2
-	Number of files	[43], [45]	2	2
-	Number of directories	[43]	1	1
-	Complexity average by file	[43]	1	1
-	Cyclomatic complexity in classes	[43]	1	1
-	Complexity average by class	[43]	1	1
-	Cyclomatic complexity in functions	[43]	1	1
-	Complexity average by function	[43]	1	1
(Dynamic metric)	DCBO, Dynamic Coupling Between Objects	[7]	1	1
(Dynamic metric)	MTTF, The Mean-Time-To-Failure	[7]	1	1
(Dynamic metric)	MTBF, Mean-Time-Between-Failures	[7]	1	1
-	Welker and Oman	[8]	1	1
-	OSmax, Maximum Operation Size	[8]	1	1
-	Ocmax, Maximum Operation Complexity	[8]	1	1
-	NAA, Number of Attributes Added	[8]	1	1

Continuazione della Tabella: Metriche - Studi (Completa)

Suite	Metrica	Studi che la usano	Cit.	Score
-	Cyclic, Number of Cyclic Dependencies	[8]	1	1
-	Proposed by [8]	[8]	1	1
-	NOI, Number of Outgoing Invocations	[10]	1	1
-	TLLOC, Total Logical Lines of Code	[10]	1	1
-	TNOS, Total Number of Statements	[10]	1	1
-	CI, Clone Instances	[10]	1	1
-	TCLOC, Total Comment Lines of Code	[10]	1	1
-	Logic	[14]	1	1
-	Dataflow	[14]	1	1
-	Nesting Depth	[14]	1	1
-	Clone Coverage	[14]	1	1
-	Bug Patterns	[14]	1	1
-	Coupling and Cohesion	[14]	1	1
-	Test Results and Coverage	[14]	1	1
-	Indentation as Proxy for Complexity Metric	[18]	1	1
-	Cocol's metric	[21]	1	1
-	RAM, RAM + CPU mem usage	[21]	1	1
-	Proposed by [21]	[21]	1	1
-	CBM, Coupling Between Methods	[19]	1	1
-	IC, Inheritance Coupling	[19]	1	1
-	CAM, Cohesion Among Methods of a Class	[19]	1	1
-	MFA, Measure of Functional Abstraction	[19]	1	1
-	MOA, Measure Of Aggregation	[19], [34]	2	2
-	DAM, Data Access Metric (Card Metric)	[19], [47]	2	2
-	LCOM3, Lack of Cohesion of Methods	[19], [42]	2	2
-	NPATH	[25]	1	1
-	Spatial Complexity Metrics	[27]	1	1
History Sensitive m.	pLOC	[32]	1	1
History Sensitive m.	rpiLOC	[32]	1	1
History Sensitive m.	rpdLOC	[32]	1	1
History Sensitive m.	rdocLOC	[32]	1	1
History Sensitive m.	rniLOC	[32]	1	1
History Sensitive m.	TL	[32]	1	1
-	Dominators Tree Metrics	[33]	1	1
-	I, Instability	[34]	1	1
-	NOP, Number of Polymorphic Methods	[34]	1	1
-	RCI, Ratio of Cohesion Interactions	[41]	1	1
-	MC, Method Coupling	[41]	1	1
-	ICC, Internal Class Complexity (CAC + CMC)	[41]	1	1
-	ECC, External Class Complexity (CIC + ICP)	[41]	1	1
-	CPC, Class Coupling Complexity	[41]	1	1
-	CHC, Class Coupling Complexity	[41]	1	1
-	Token Count	[44]	1	1
Mood's Metrics	MHF, Method Hiding Factor	[44]	1	1
Mood's Metrics	AHF, Attribute Hiding Factor	[44]	1	1

Continuazione della Tabella: Metriche - Studi (Completa)

Suite	Metrica	Studi che la usano	Cit.	Score
Mood's Metrics	MIF, Method Inheritance Factor	[44]	1	1
Mood's Metrics	AIF, Attribute Inheritance Factor	[44]	1	1
Mood's Metrics	PF, Polymorphism Factor	[44]	1	1
Mood's Metrics	CF, Coupling Factor	[44]	1	1
Narsimhan's metrics	AID, Average Interaction Density	[44]	1	1
Narsimhan's metrics	IID, Incoming Interaction Density	[44]	1	1
Narsimhan's metrics	OID, Outgoing Interaction Density	[44]	1	1
-	Branching Complexity (Sneed Metric)	[42]	1	1
-	Data complexity (Chapin Metric)	[42]	1	1
-	Decisional complexity (McClure)	[42]	1	1
Aspect-based metrics	Number of Aspects	[44]	1	1
Aspect-based metrics	NPA, Number of Pointcuts per Aspect	[44]	1	1
Aspect-based metrics	NAA, Number of Advices per Aspect	[44]	1	1
Aspect-based metrics	DCP, Degree of Crosscutting per Pointcut	[44]	1	1
Aspect-based metrics	RAD, Response for Advice	[44]	1	1
-	FANIN	[42]	1	1
-	FANOUT	[42]	1	1
-	Coupling Intensity	[42]	1	1
-	Coupling Dispersion	[42]	1	1
-	Essential Complexity	[42]	1	1
-	Paths	[42]	1	1
-	Nesting (Max Nest)	[42]	1	1
-	Base Classes (IFANIN)	[42]	1	1
-	Override Ratio	[42]	1	1
-	NIV, Instance Variables	[42]	1	1
-	NIM, Instance Methods	[42]	1	1
-	WOC, Weight of Classes	[42]	1	1
-	NOPA, Number of Public Attributes	[42]	1	1
-	Aggregate Stability	[24]	1	1
-	Branch Stability	[24]	1	1
-	Vision Distance	[24]	1	1
-	Version Stability	[24]	1	1
-	Structure Stability	[24]	1	1
-	Shotgun Surgery	[30]	1	1

Fine della Tabella: Metriche - Studi (Completa)