



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

INTERVENTI DI MANUTENZIONE ED EVOLUZIONE DEL SOFTWARE: IMPATTO DELLE METRICHE DI RIUSO

RELATORE

Prof. Carmine Gravino

Prof. Fabio Palomba

Dott. Giammaria Giordano

Università degli Studi di Salerno

CANDIDATO

Gerardo Festa

Matricola: 0512105908

"Mira alla luna.

Male che andrà, ti ritroverai tra le stelle"

Sommario

Il riuso del codice è uno degli aspetti più importanti nello sviluppo software, in quanto permette di ridurre i costi di sviluppo e manutenzione, semplificando inoltre il processo di documentazione. Nell'ambito dello sviluppo Object Oriented vengono messi a disposizione dei developers due meccanismi fondamentali per favorire il riuso, ovvero l'ereditarietà, che si articola in ereditarietà di specifica e di implementazione, e la delegazione.

Mentre buona parte della ricerca in questo ambito si è concentrata sull'impatto dei meccanismi di riuso sulla qualità del codice, poco è stato fatto per valutare l'evoluzione dell'utilizzo degli stessi e, in particolare, su come questi possano impattare *a lungo termine* sulla qualità del codice.

La tesi in esame propone dunque uno studio empirico sull'evoluzione dell'ereditarietà di implementazione, dell'ereditarietà di specifica e della delegazione e sul loro impatto sulla qualità del software.

In primo luogo, viene analizzato come i tre meccanismi evolvono su undici sistemi software Open Source, con un approccio *fine-grained* volto a valutare la storia dei sistemi commit per commit. In seguito, vengono utilizzati modelli statistici per capire come l'eredità e la delegazione impattano sulla qualità del codice, espressa in termini di presenza dei difetti. In ultimo, viene valutata la possibilità di predire l'effort necessario per la correzione dei difetti presi in analisi.

È stato evidenziato che i meccanismi di riuso evolvono nel lungo periodo, ma che, seppure il trend sia generalmente crescente - almeno in termini asintotici -, ogni sistema ha una storia diversa, e dunque non è possibile generalizzare i risultati. Inoltre, sul lungo periodo, solo in pochi casi il riuso misurato ha avuto un impatto sulla presenza di difetti nel codice, con effetti variabili. Infine, sul dataset analizzato, è risultato impossibile fare una stima accurata dell'effort per la risoluzione dei difetti.

Indice

Indice	ii
Elenco delle figure	v
Elenco delle tabelle	vii
1 Introduzione	1
1.1 Contesto Applicativo	1
1.2 Motivazioni e Obiettivi	3
1.3 Risultati	4
1.4 Struttura della tesi	4
2 Stato dell'arte	5
2.1 Background	5
2.2 Studi empirici sul riuso	6
2.3 Studi su qualità del codice ed interventi manutentivi	7
2.4 Studi su qualità del codice e presenza di difetti	7
2.5 Repository mining	8
2.6 Limitazioni dello stato dell'arte	9
3 Metodologia	10
3.1 Definizione delle Research Questions	10
3.2 Selezione del contesto	11
3.3 Raccolta dei dati	12

3.3.1	Estrazione delle metriche di riuso	12
3.3.2	Estrazione dei code churns e dei bug	14
3.3.3	Preparazione dei dataset	16
3.4	RQ1: Analisi dell’evoluzione delle metriche di ereditarietà e delegazione	17
3.5	RQ2: Correlazione tra utilizzo del riuso e presenza di difetti nel codice	17
3.6	RQ3: È possibile predire l’effort necessario per la risoluzione dei difetti?	20
3.7	Minacce alla validità	21
4	Risultati	23
4.1	Risultati RQ1	23
4.1.1	Codec	23
4.1.2	Cli	27
4.1.3	CommonsCollections	30
4.1.4	CommonsCsv	32
4.1.5	Compress	34
4.1.6	Gson	37
4.1.7	JacksonCore	39
4.1.8	JacksonDatabind	41
4.1.9	JacksonXml	43
4.1.10	JXPath	45
4.1.11	Time	47
4.1.12	Risposta alla domanda di ricerca RQ1	50
4.2	Risultati RQ2	52
4.2.1	Codec	53
4.2.2	Cli	53
4.2.3	CommonsCollections	56
4.2.4	CommonsCsv	56
4.2.5	Compress	56
4.2.6	Gson	56
4.2.7	JacksonCore	57
4.2.8	JacksonDatabind	57
4.2.9	JacksonXml	57
4.2.10	JXPath	57
4.2.11	Time	58

4.2.12 Risposta alla domanda di ricerca RQ2	58
4.3 Risultati RQ3	59
4.3.1 Codec	59
4.3.2 Cli	59
4.3.3 CommonsCollections	62
4.3.4 CommonsCsv	62
4.3.5 Compress	62
4.3.6 Gson	62
4.3.7 JacksonCore	62
4.3.8 JacksonDatabind	62
4.3.9 JacksonXml	63
4.3.10 JXPath	63
4.3.11 Time	63
4.3.12 Risposta alla domanda di ricerca RQ3	63
5 Conclusioni	64
A Appendice RQ2	66
B Appendice RQ3	76

Elenco delle figure

3.1 Esempio di matrice di correlazione	20
4.1 Codec - Ereditarietà di implementazione	23
4.2 Codec - Ereditarietà di specifica	24
4.3 Codec - Delegazione	26
4.4 Cli - Ereditarietà di implementazione	27
4.5 Cli - Ereditarietà di specifica	28
4.6 Cli - Delegazione	29
4.7 CommonsCollections - Ereditarietà di implementazione	30
4.8 CommonsCollections - Ereditarietà di specifica	31
4.9 CommonsCollections - Delegazione	32
4.10 CommonsCsv - Ereditarietà di implementazione	32
4.11 CommonsCsv - Ereditarietà di specifica	33
4.12 CommonsCsv - Delegazione	34
4.13 Compress - Ereditarietà di implementazione	35
4.14 Compress - Ereditarietà di specifica	36
4.15 Compress - Delegazione	36
4.16 Gson - Ereditarietà di implementazione	37
4.17 Gson - Ereditarietà di specifica	38
4.18 Gson - Delegazione	39
4.19 JacksonCore - Ereditarietà di implementazione	40
4.20 JacksonCore - Ereditarietà di specifica	41

4.21 JacksonCore - Delegazione	41
4.22 JacksonDatabind - Ereditarietà di implementazione	42
4.23 JacksonDatabind - Ereditarietà di specifica	43
4.24 JacksonDatabind - Delegazione	43
4.25 JacksonXml - Ereditarietà di implementazione	44
4.26 JacksonXml - Ereditarietà di specifica	45
4.27 JacksonXml - Delegazione	46
4.28 JXPath - Ereditarietà di implementazione	46
4.29 JXPath - Ereditarietà di specifica	47
4.30 JXPath - Delegazione	48
4.31 Time - Ereditarietà di implementazione	48
4.32 Time - Ereditarietà di specifica	49
4.33 Time - Delegazione	50

Elenco delle tabelle

4.1	Variabili rimosse dai progetti a causa della multicollinearità	52
4.2	Codec - applicazione del modello multinomiale	54
4.3	Cli - applicazione del modello multinomiale	55
4.4	Codec - applicazione del modello lineare generalizzato	60
4.5	Cli - applicazione del modello lineare generalizzato	61
A.1	CommonsCollections - applicazione del modello multinomiale	67
A.2	CommonsCsv - applicazione del modello multinomiale	68
A.3	Compress - applicazione del modello multinomiale	69
A.4	Gson - applicazione del modello multinomiale	70
A.5	JacksonCore - applicazione del modello multinomiale	71
A.6	JacksonDatabind - applicazione del modello multinomiale	72
A.7	JacksonXml - applicazione del modello multinomiale	73
A.8	JxPath - applicazione del modello multinomiale	74
A.9	Time - applicazione del modello multinomiale	75
B.1	CommonsCollections - applicazione del modello lineare generalizzato	77
B.2	CommonsCsv - applicazione del modello lineare generalizzato	78
B.3	Compress - applicazione del modello lineare generalizzato	79
B.4	Gson - applicazione del modello lineare generalizzato	80
B.5	JacksonCore - applicazione del modello lineare generalizzato	81
B.6	JacksonDatabind - applicazione del modello lineare generalizzato	82
B.7	JacksonXml - applicazione del modello lineare generalizzato	83

B.8 JxPath - applicazione del modello lineare generalizzato	84
B.9 Time - applicazione del modello lineare generalizzato	85

CAPITOLO 1

Introduzione

1.1 Contesto Applicativo

Il riuso del Software Il riuso è una pratica di fondamentale importanza nello sviluppo del software, grazie alla quale i programmati posso utilizzare codice già esistente per implementare nuove funzionalità [1]. Ciò permette un considerevole risparmio in termini di tempo di coding - ma anche di testing e manutenzione - rendendo inoltre più semplice il processo di standardizzazione e documentazione. Questa pratica può essere implementata in diversi modi: la creazione e l'utilizzo di librerie, i design pattern, il refactoring, ma anche i più classici *copy-and-paste* e *copy-and-modify* rappresentano esempi di tecniche utilizzabili per ottenere il riuso [2]. Questo assume ulteriore rilevanza nel contesto dei linguaggi OO (Object Oriented) che mettono a disposizione dello sviluppatore meccanismi di astrazione proprio a supporto della riusabilità del codice. Concentrandosi sul linguaggio Object Oriented JAVA, spiccano per importanza l'*ereditarietà*, spesso considerata il fattore principale per la distinzione tra approccio OO e altri paradigmi moderni di programmazione, e la *delegazione*.

Ereditarietà e delegazione L'idea alla base dell'ereditarietà è piuttosto semplice. Questa, infatti, permette la definizione di nuovi oggetti sulla base di altri oggetti già esistenti; quando si crea un nuovo oggetto, dunque, è necessario specificare solo le proprietà che differiscono da quelle dell'oggetto già esistente su cui ci si basa. In questo caso si innesta una gerarchia, nella quale la classe già esistente viene chiamata *super-classe*, *classe-base* o *classe-padre/genitore*,

mentre la nuova classe definita, che eredita lo stato e/o il comportamento dalla classe preesistente, prende il nome di *classe-figlia* o *sotto-classe*. La *delegazione*, invece, è un meccanismo che permette ad una classe di lasciare la responsabilità di un task ad un'altra classe. Questo significa che la classe che beneficia della delegazione potrà utilizzare oggetti istanza di altre classi, ai quali inoltrare messaggi con il fine di fargli eseguire delle azioni. Ciò permette alla classe di esporre determinati comportamenti che vengono in realtà espletati anche (o esclusivamente) grazie ai comportamenti di altre classi.

Gli innegabili vantaggi portati dall'ereditarietà, però, hanno un costo: l'aumento della complessità del sistema e dell'accoppiamento tra le classi [3].

Nonostante l'impatto e l'importanza di questi meccanismi sia stata più volte rimarcata nell'ambito della ricerca, anche con l'introduzione di metriche specifiche come la suite di metriche di Chidamber e Kemerer, poco è stato studiato sull'evoluzione a lungo termine di questi meccanismi in relazione alla qualità del codice [1]. In particolare, non è stato mai chiarito se possa esserci una correlazione tra l'utilizzo del riuso e la presenza di difetti nel codice e, di conseguenza, con l'effort per la risoluzione degli stessi.

Evoluzione del software Per creare un software ben funzionante e di successo è necessario che questo attraversi un continuo processo di modifica dettato dalla necessità di adattarsi a nuovi o diversi requisiti e di stare al passo con l'innovazione tecnologica. Questo processo è rappresentato dall'evoluzione, o manutenzione, del software [4].

In generale, si possono distinguere gli interventi fatti sul software in due classi: interventi *evolutivi/perfettivi* ed interventi *correttivi*. La prima classe fa riferimento all'aggiunta di codice volto alla creazione di una nuova funzionalità per venire incontro a nuovi o modificati requisiti, mentre la seconda fa riferimento a tutti gli interventi che si rendono necessari per eliminare malfunzionamenti - o difetti - o per migliorare le performance.

Nell'ambito dei software open source, GitHub rappresenta uno strumento fondamentale sia per gli sviluppatori che per i ricercatori, che, per mezzo di *Repository*, *Commit*, *Versioni* e *Release*, possono seguire dettagliatamente l'evoluzione dei progetti esaminati.

La manutenzione è un processo costoso e talvolta complesso che potrebbe impattare sulla qualità del codice, arrivando anche a snaturare il design originale o a ridurre la manutenibilità stessa [5]. Per poter stimare l'effort che un certo intervento di manutenzione richiede, è possibile considerare la metrica del *code churn*, ovvero la somma delle linee di codice aggiunte, modificate ed eliminate da un commit che introduce una modifica alla repository.

Qualità del codice e difetti La qualità del software è un argomento molto vasto e ampiamente discusso. Ci sono più definizioni di qualità del software, a seconda della prospettiva dalla quale la si osserva. L'IEEE la definisce come il grado con cui un sistema soddisfa i requisiti e le aspettative degli utenti. Negli anni la definizione è stata estesa, ad esempio da Pressman, per comprendere anche le caratteristiche implicite di un prodotto software, ovvero quelle caratteristiche che ci si aspetta gli sviluppatori rispettino anche laddove non siano presenti vincoli contrattuali [6].

La qualità del software è certamente uno degli argomenti cardine del Software Engineering, con i ricercatori che si sono occupati di trovare il modo di ottenere una misura della stessa. Il risultato è rappresentato, a livello di codice, da molte metriche standard, a partire dalla più banale LOC (Lines of Code), passando a metriche più complesse come la RFC (Response for a Class) o la Complessità Ciclomatica [7]. In ambito Object Oriented vengono spesso utilizzate le metriche della suite di Chidamber e Kemerer , che permettono di avere una visione generale sul design delle classi e dunque di osservare come l'Object Orientation viene utilizzata all'interno di un progetto [8]. Una bassa qualità del codice porta con ogni probabilità all'introduzione di difetti - o *faults, bugs* - nel software. I difetti sono stati a loro volta oggetto di ampia sperimentazione, soprattutto in ambito di predizione; si è cercato, infatti, di collegare cambiamenti nelle metriche della qualità del codice alla presenza o introduzione di fault, per giungere dunque a modelli in grado di specificare la cosiddetta *fault-proneness* di una classe [9]. La presenza di difetti, di riflesso, rappresenta un buon indicatore per la qualità del codice.

1.2 Motivazioni e Obiettivi

Studi empirici precedenti hanno avuto come target l'evoluzione dei meccanismi di ereditarietà all'interno di software Open Source Java, come quello di Nasseri *et al.* [10] che in particolare si concentra sulle gerarchie di eredità, mentre altri studi si sono soffermati sull'impatto dell'ereditarietà sulle metriche software e sulla manutenibilità [11]. Pochi studi, invece, hanno valutato l'ereditarietà nel contesto dell'evoluzione del software mettendola in relazione con la qualità del software. Tra questi, lo studio di Giordano *et al.* [1], che esprime la qualità del software in termini di severità dei code smells, è quello che più si avvicina all'obiettivo di questa tesi. Laddove sia stata analizzata l'evoluzione di un sistema, tuttavia, è stato effettuato un campionamento delle release o delle versioni su cui focalizzarsi, mentre in letteratura non esistono studi che hanno percorso l'intera storia di un progetto software. Per

questo motivo, è stato deciso di dirigersi verso un percorso *fine-grained*, che prevede l'analisi **commit per commit** di più sistemi software.

In particolare, in questo lavoro di tesi viene presentata una analisi empirica che ha lo scopo di investigare come i meccanismi di ereditarietà e delegazione evolvono nel tempo, considerando anche l'impatto che questi hanno sull'evoluzione della qualità del software. Vengono presi in esame ereditarietà e delegazione in quanto rappresentano delle astrazioni atte, per lo più, a favorire il riuso del codice.

La qualità del software viene misurata in termini di presenza di difetti e lo studio intende anche stabilire se è possibile predire i costi per la risoluzione dei difetti stessi.

1.3 Risultati

Dallo studio emerge che i meccanismi di riuso evolvono nel lungo periodo, ma non è possibile generalizzare i risultati su tutti i progetti. Ciascun software, infatti, in particolar modo quando analizzato a livello dei commit, ha una storia propria, influenzata anche dalle modalità con cui è gestita la repository. Solo in pochi casi il riuso ha impattato sulla presenza di difetti nel codice, con effetti variabili da progetto a progetto. Infine, l'effort per la risoluzione dei difetti non è risultato stimabile con accuratezza.

1.4 Struttura della tesi

Il resto della tesi è strutturato come descritto in seguito. Nel Capitolo 2 vengono descritti i lavori presenti in letteratura sugli aspetti di ricerca trattati nello studio e viene fornito, inoltre, un background sul riuso. Il Capitolo 3 formalizza gli obiettivi dello studio in research questions e descrive la metodologia utilizzata per perseguirli, spiegando la procedura di raccolta dei dati, motivando le decisioni prese per l'analisi e individuando le minacce alla validità dello studio. Il Capitolo 4 mostra i risultati ottenuti su tutti i progetti analizzati, per poi rispondere alle research questions. Infine, il Capitolo 5 riporta le conclusioni e accenna spunti per possibili studi futuri.

CAPITOLO 2

Stato dell'arte

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nello studio.

2.1 Background

Nei linguaggi Object Oriented, e in particolare in Java, è presente la possibilità di fare uso dell'ereditarietà per stabilire una dipendenza gerarchica tra due classi per mezzo di due costrutti:

- **Extends:** con l'utilizzo di questa keyword, una classe A eredita lo stato e il comportamento di una classe B, innestando una gerarchia in cui B rappresenta la *super-classe* e A la *sotto-classe*. Quando viene utilizzato un oggetto della classe A, si avranno a disposizione tutti gli attributi e i metodi presenti - e implementati - in B.
- **Implements:** con l'utilizzo di questa keyword, una classe A eredita i metodi definiti da un'*interfaccia* B. Un'interfaccia contiene solo metodi astratti, ovvero metodi senza corpo. Questo comporta che tutte le classi che ereditano - per mezzo di *implements* - da un'interfaccia devono provvedere all'implementazione di questi metodi.

Da qui in poi, la prima tipologia di ereditarietà verrà chiamata **ereditarietà di implementazione**, mentre la seconda **ereditarietà di specifica**. Quest'ultima, anche nota come eredità di interfaccia, sposa il principio di sostituzione di Liskov, secondo il quale laddove fosse sempre

possibile sostituire un oggetto di tipo B con un oggetto di tipo A, allora A rappresenta un sottotipo di B [12]. Inoltre, l'ereditarietà di specifica si confà al principio di eredità stretta, per il quale i discendenti di una classe non modificano o eliminano nessuno dei comportamenti ereditati. Dunque se A eredita da B tramite ereditarietà di specifica, A esporrà lo stesso comportamento definito in B, aggiungendo, eventualmente, più features.

Per l'ereditarietà di implementazione, invece, questi due principi non valgono, in quanto la classe A che eredita da B riceve tutti le operazionimesse da B. In più, l'eredità di implementazione viola il principio di encapsulamento, grazie al quale una classe può mascherare la sua struttura interna e impedire l'accesso al suo stato e al suo comportamento (o a parte di esso). Infatti, se una classe A estende una classe B, la classe A ha la possibilità di richiamare qualsiasi metodo della classe B, anche quelli che la classe B non espone all'esterno. Ciò comporta che la sotto-classe potrebbe utilizzare i metodi della super-classe in maniera non appropriata. La delegazione pone una soluzione al problema dell'incapsulamento. Grazie a questo meccanismo una classe può sfruttare le operazioni che un'altra classe mette a disposizione - e dunque *delegare* a questa parte delle sue responsabilità - senza necessità di creare una relazione di ereditarietà.

2.2 Studi empirici sul riuso

Molte analisi si sono concentrate sui meccanismi di ereditarietà e riuso, evidenziandone aspetti positivi e negativi. Più esperimenti hanno riportato un generale impatto negativo del riuso sulla manutenibilità del software, con un focus sulla metrica DIT (Depth of Inheritance Tree) presente nella suite di metriche OO di Chidamber e Kemerer. Lo studio di *Prechelt et al.* [13] ha dimostrato che mantenendo bassa la profondità degli alberi di ereditarietà si riduce l'effort manutentivo, mentre lo studio di *Daly et al.* [14] è giunto alla conclusione che più una classe si trova in basso nell'albero di ereditarietà, più è difficile da manutenere, andando dunque a rafforzare l'analisi di *Prechelt et al.*

Yu et al. hanno riportato l'impatto negativo che il riuso ha sulla qualità del codice, trovando una forte correlazione tra Number of Children e la presenza di difetti [15], mentre *Brito e Melo* hanno utilizzato le metriche MOOD per valutare il riuso e hanno trovato una correlazione negativa con la densità dei difetti, seppure i benefici decrescano con un'utilizzo massiccio dell'ereditarietà [11]. *Chhikara et al.* [16], invece, hanno evidenziato come l'ereditarietà migliori la qualità del software, riducendo la complessità del codice, mentre gli studi di *Vinobha et al.*

[17] hanno riportato una correlazione positiva tra ereditarietà, riusabilità e manutenibilità. Dunque, da un lato ci sono studi che hanno dimostrato come il riuso possa impattare in negativo la qualità del software, in particolar modo sulla fault-proneness delle classi, mentre dall'altro è stato provato come possa impattare positivamente sulla manutenibilità del codice.

2.3 Studi su qualità del codice ed interventi manutentivi

Diversi studi si sono soffermati sulla relazione che c'è tra la presenza di un difetto nel codice e l'intervento che ne consegue per correggerlo. La maggior parte ha cercato di creare un modello predittivo per stimare l'effort necessario per la correzione. È il caso, ad esempio, di *Song et al.* [18], che hanno proposto un predittore basato su rule mining.

Molti esperimenti hanno cercato di trovare quale fosse la miglior metrica per la predizione dell'effort, e molteplici studi hanno sostenuto la validità dei **code churns** in questo ruolo. Lo studio di *Shibab et al.* [19] spiega che le linee di codice da sole non bastano come predittore, ma vanno messe insieme proprio ai churns per essere più efficaci. I code churns non rappresentano altro che **la somma delle righe aggiunte, rimosse o modificate** da un commit che apporta un cambiamento alla repository.

Catolino et al. [5] hanno creato un modello efficace di predizione dell'effort di un intervento manutentivo in termini di code churns, mentre *Ferdian Thung* [20] si è concentrato proprio sulla predizione di interventi correttivi, stimando l'effort, di nuovo, in termini di code churns. È chiaro dunque come i code churns rappresentino una buona metrica per la predizione dell'effort.

2.4 Studi su qualità del codice e presenza di difetti

La correlazione tra la qualità del codice e la presenza dei difetti rappresenta un topic centrale per l'ingegneria del software. Molti studi si sono concentrati sulla predizione dei difetti sulla base di metriche di qualità del software. *Catolino et al.* [5] utilizzano un set da ventiquattro metriche precedentemente validate da *Rahman et al.* [21] e *Kamei et al.* [22] allo scopo di creare un predittore Just-In-Time (o JIT) per i faults e inoltre asseriscono che, in linea con studi recenti, le migliori performance per i modelli predittivi si ottengono utilizzando combinazioni di metriche. Uno studio condotto da *Zhou e Leung* [23] ha validato le metriche della suite di Chidamber e Kemerer come buoni predittori per la predizione di faults. In particolare, i faults venivano classificati in due categorie di peso in modo da discernere la

severità. Le metriche hanno formato bene come predittori per i faults di basso livello, peggio per quelli di alto livello. Lo studio è stato successivamente ampliato e approfondito da *Singh et al.* [24], che hanno introdotto un livello intermedio di severità e hanno ribadito come le metriche CK siano buoni predittori per la fault-proneness di basso e medio livello, ma meno buoni per la predizione dei difetti più gravi.

Anche *Gyimothy et al.* [9] hanno validato le metriche di Chidamber e Kemerer nell'ambito della fault-proneness prediction, in particolare individuando la CBO (*Coupling Between Objects*) come la metrica di maggior impatto. È evidente, alla luce di questi studi, che la qualità del codice sia strettamente legata alla presenza di difetti.

2.5 Repository mining

Il repository mining consiste nell'esplorazione di una repository software con lo scopo di carpire delle informazioni relative allo sviluppo. Una repository infatti, come quelle ospitate da GitHub, permette di osservare tutta l'evoluzione del progetto tramite l'analisi dei commit, ovvero degli snapshot della repository. I commit, dopo un parsing performato da Github, danno la possibilità di osservare quali file sono stati modificati e in che modo, oltre a contenere metadati sull'autore, sulla data dell'operazione e altro ancora. Queste informazioni permettono dunque di fare analisi non solo strettamente legate al codice, ma anche alla community di sviluppatori dietro i progetti Open Source.

In letteratura vengono spesso utilizzati framework come PyDriller e Repodriller [25] [26] che semplificano il processo di mining [27] [28] [29]. Nonostante questi due tool abbiano delle limitazioni nelle performance, come sostenuto da *Steinhauer e Palomba* [30], restano ampiamente utilizzati negli studi che necessitano di una overview sull'evoluzione del software nel tempo.

2.6 Limitazioni dello stato dell’arte

Sulla base di quanto descritto, si può osservare come, nonostante il riuso sia stato oggetto di massiccia sperimentazione, non sia ancora perfettamente chiaro il ruolo dell’ereditarietà, con studi che imputano a questa la decadenza della qualità del codice, mentre altri la mettono in relazione con un’aumento della stessa. Inoltre, difficilmente studi precedenti hanno suddiviso l’ereditarietà in ereditarietà di specifica e di implementazione, rendendo complesso discernere tra l’impatto che queste due possono avere individualmente.

Il riuso è stato inoltre utilizzato come predittore per la fault-proneness delle classi, e in questo studio viene utilizzata proprio la presenza di faults, o difetti, per valutare la qualità del software.

Il lavoro che più si avvicina a quello proposto è quello di *Giordano et al.* [1] che considera sì la differenza tra i tipi di ereditarietà, ma misura la qualità del codice in termini di severità dei code smells. Inoltre, l’analisi sull’evoluzione è basata su pochi punti della storia del software, in quanto viene preso in esame un sottoinsieme delle release dei sistemi software. Nello studio presentato in questo lavoro di tesi, invece, la granularità viene raffinata, facendo mining su tutti i commit dei progetti software e osservando, dunque, i cambiamenti che avvengono nel cosiddetto *short-term*. Nonostante l’analisi commit per commit sia stata effettuata in ambito di defect prediction, come nello studio di *Pascarella et al.* [31], questa tecnica non è mai stata utilizzata per l’analisi dell’evoluzione dell’utilizzo del riuso nei progetti software.

CAPITOLO 3

Metodologia

In questo capitolo vengono formalizzati gli obiettivi dello studio, vengono definiti i sistemi e il metodo che ha permesso di effettuare la raccolta dei dati per poi descrivere la strategia di analisi per ogni research question formulata. Il capitolo si chiude con la discussione delle minacce alla validità dello studio condotto.

3.1 Definizione delle Research Questions

L'obiettivo dello studio è capire come i meccanismi di ereditarietà e delegazione evolvono nel tempo e come questi impattano sulla presenza di difetti all'interno del codice, cercando inoltre di stimare l'effort necessario per gli interventi correttivi per i suddetti difetti. La presenza dei difetti viene intesa come indicatore per la qualità del software.

La ricerca si è concentrata su tre obiettivi principali. In primo luogo, viene analizzata l'evoluzione dei meccanismi di riuso. Questo porta alla definizione della prima Research Question:

RQ1. Come varia l'utilizzo dei meccanismi di riuso nell'evoluzione del software?

Come detto precedentemente, il focus è stato posto sui meccanismi di ereditarietà di implementazione, ereditarietà di specifica e delegazione, rendendo necessaria la definizione di tre sotto-domande di ricerca, ovvero:

RQ1₁. Come varia il riuso in termini di ereditarietà di implementazione nell’evoluzione del software?

RQ1₂. Come varia il riuso in termini di ereditarietà di specifica nell’evoluzione del software?

RQ1₃. Come varia il riuso in termini di delegazione nell’evoluzione del software?

Una volta conclusa l’analisi sull’evoluzione del riuso, lo studio si concentra sull’impatto che questa evoluzione ha sulla qualità del codice, intesa come presenza di difetti nel software. Dunque, viene formulata la seconda Research Question:

RQ2. Esiste una correlazione tra l’utilizzo dei meccanismi di riuso e la presenza di difetti nel codice?

Infine, lo studio indaga sulla possibilità di predire l’effort necessario per la correzione dei difetti, espresso in termini di code churns. Ciò viene formulato dalla terza Research Question:

RQ3. È possibile predire l’effort necessario per la correzione dei difetti?

3.2 Selezione del contesto

Il contesto dello studio consiste di undici progetti Open Source Java, quali CODEC, CLI, COMMONSCOLLECTIONS, COMMONSCSV, COMPRESS, GSON, JACKSONCORE, JACKSONDATABIND, JACKSONXML, JXPATH e TIME. Questi progetti non sono frutto di una scelta casuale, bensì rappresentano un sottoinsieme dei diciassette software esaminati da Defects4J. Defects4j propone una collezione di difetti di sistemi Java e fornisce anche un’architettura utile per la riproduzione dei bug a supporto della ricerca nell’ambito dell’ingegneria del software [32].

Proprio la replicabilità dei bug ha indotto i ricercatori ad utilizzare questo tool per ricerche che prevedono l’analisi dei difetti, spesso nell’ambito dell’Automatic Bug Repair, come nel caso di Martinez *et al.* [33], di Durieux *et al.* [34] e di Jiang *et al.* [35]. Altri invece si sono soffermati su uno studio dell’anatomia dei bug del dataset proposto [36] [37]. Defects4j rappresenta dunque uno strumento valido e molto utilizzato nel campo del Software Engineering.

Alcuni dei progetti proposti dal tool, tuttavia, sono stati scartati per molteplici ragioni. Prima

di passare alle motivazioni della selezione, è bene far presente che lo studio è fortemente legato a GitHub. Siccome non ci si sofferma solo sui bug, ma si vuole valutare (per RQ1) l’evoluzione dei meccanismi di riuso, per i progetti viene considerato tutto lo storico di Commits presenti su Git, e Defects4j viene utilizzato per ottenere la lista dei commit che introducono difetti e di quelli che rimuovono i difetti. I sistemi, quindi, vengono clonati da GitHub seppure Defects4J dia la possibilità di fare il checkout dei progetti nelle versioni in cui presentano bug. Questo ha portato all’eliminazione di tre dei diciassette progetti del tool, ovvero JFREECHART, i cui bug venivano riportati sulle versioni hostate da SourceForge, MATH e LANG, i cui commit che introducono/rimuovono difetti sono stati rimossi da qualsiasi branch, restando dunque inaccessibili.

CLOSURECOMPILER e MOCKITO sono stati scartati a causa della loro dimensione che non permetteva la terminazione in tempi accettabili dell’esosa estrazione dei dati per ogni commit. Questa scelta è motivata anche dal fatto che, essendo il numero di repository analizzate significativo, il rapporto costi/benefici anche in termini di time consumption non era sufficientemente vantaggioso per considerarli. Infine JSOUP non è stato preso in considerazione in quanto ha presentato un problema di compatibilità con la libreria JGit, utilizzata in seguito per il mining delle repository.

3.3 Raccolta dei dati

3.3.1 Estrazione delle metriche di riuso

Per eseguire questa parte di raccolta dei dati viene utilizzato un tool sviluppato e utilizzato nel contesto universitario, opportunamente modificato per meglio adempiere agli obiettivi dello studio. Questo tool, scritto in Java, sfrutta il framework RepoDriller per analizzare la repository ed estrae informazioni riguardanti l’ereditarietà e le metriche OO di Chidamber e Kemerer. In primis viene impostato il mining in modo che vengano analizzati tutti i commit della repository data in input. A questo punto, per ogni commit viene avviata una visita che inizia con il checkout della repository a quel commit. L’operazione di checkout permette di riportare la versione della repository ad una precedente. Ciò è possibile perché, come scritto in precedenza, un commit rappresenta uno snapshot del sistema. Si utilizza una cartella temporanea per lo storage della versione di cui è stata effettuata il checkout.

La visita per il commit continua con il parsing dei contenuti, sfruttando JDT, di ogni singolo file .java presente nella cartella temporanea. Sui file che hanno subito il parsing vengono dunque calcolate le metriche CK, qui riportate:

1. *Lines Of Code (LOC)*: il numero delle linee di codice esaminate nella classe,
2. *Weighted Methods per Class (WMC)*: somma pesata dei metodi della classe,
3. *Response For a Class (RFC)*: il numero dei metodi che possono essere richiamati in risposta ad un messaggio ricevuto da un oggetto della classe,
4. *Lack of Cohesion of Methods (LCOM)*: il numero di metodi nella classe che non si somigliano meno il numero dei metodi che si somigliano,
5. *Coupling Between Objects (CBO)*: il numero delle classi con le quali la classe in esame è accoppiata,
6. *Depth of Inheritance Tree (DIT)*: la lunghezza del path più lungo dalla classe fino alla radice della gerarchia di ereditarietà,
7. *Number Of Children (NOC)*: il numero delle classi figlie che ereditano dalla classe in esame

Per quanto riguarda invece l'ereditarietà di specifica e l'ereditarietà di implementazione, si osserva se la classe utilizza le rispettive keyword, ovvero 'implements' e 'extends' rispettivamente. L'esito di questa verifica viene riportata in formato booleano. Dunque, almeno in questa fase *class-level*, le due metriche - riportate come **intInh** e **impInh**, per interfaceInheritance e implementationInheritance - non rappresentano altro che dei flag che indicano se la classe fa uso o meno della specifica ereditarietà.

Per la delegazione, invece, il discorso è diverso; vengono analizzate, infatti, le chiamate fatte a metodi esterni e per ognuna di queste viene incrementato un contatore.

Per ogni file analizzato, viene salvata una nuova linea all'interno di un file .csv che dunque ha, per ogni riga: COMMIT, CLASSE, INTINH (InterfaceInheritance, ereditarietà di specifica), IMPINH (ImplementationInheritance, ereditarietà di implementazione), DELEGATIONS (delegazione), LOC, WMC, RFC, LCOM, CBO, DIT, NOC. Ciò permette l'utilizzo della coppia [commit, classe] come chiave principale per scorrere il dataset.

3.3.2 Estrazione dei code churns e dei bug

La seconda parte dell'estrazione, invece, riguarda i bug e i code churns. Vengono utilizzati script Python scritti ad hoc¹ che sfruttano l'interazione via Bash con Git e Defects4J, per poi fare uso del tool PyDriller per il mining delle repository. In primo luogo viene effettuata la clonazione dei progetti in locale per mezzo della '*clone*' di Git, generando inoltre un file '*commitsInfo.csv*' che contiene la lista degli identificativi dei commit - ovvero gli hash - presenti nel main - o default - branch di quella repository, con tanto di date ed eventuali tag. Con la parte relativa ai tag è stato anche implementato un sistema che permette di associare ai commit una versione in maniera time-based; tuttavia ciò non rientra negli obiettivi dello studio. In seguito, una query a Defects4j permette di ottenere un file CSV, nominalmente *<NomeProgetto>Revisions.csv*, contenente le informazioni dei bug riportate dal tool per quel progetto. In particolare, questo file fa corrispondere a un '*bug.id*' l'hash del commit che ha introdotto il bug, sotto la colonna '*revision.id.buggy*' e l'hash del commit che ha risolto il bug, sotto la colonna '*revision.id.fixed*'. Il '*bug.id*' rappresenta l'identificativo del bug in forma numerica, in quanto questo è il metodo con cui Defects4J li cataloga. La query fa in modo di prendere in considerazione solo i difetti che nella documentazione del tool vengono presentati come attivi, scartando dunque i bug deprecati, ovvero quelli che Defects4J non riesce a replicare.

Successivamente, di nuovo grazie a Git, viene generato l'insieme di tutti i file .java creati nella storia della repository. Vengono tuttavia scartati i file di test. Viene poi creato un Miner sulla base di Pydriller, che prende in input il path della repository e il commit da analizzare. L'esecuzione avviene sul commit solo laddove questo sia presente nel main branch e, inoltre, non sia un merge.

I merge sono particolarmente problematici, in quanto Pydriller restituisce un insieme vuoto di file modificati per essi. La motivazione che viene data dal creatore di Pydriller [38] è che, nei fatti, un merge non introduce nuovi file ma esegue solo la risoluzione dei conflitti tra i file provenienti dai due branch. Riportare la modifica dei file nel merge comporterebbe quindi un'inesattezza, in quanto risulterebbe che i file sono stati aggiunti/modificati due volte: una volta con l'aggiunta/modifica effettiva nel branch non principale e una volta nel merge. Tuttavia, è bene notare che laddove venga effettuato un merge di un branch secondario e del main branch, i commit del branch secondario rientrano nel main branch e verranno dunque presi in analisi dallo studio. Il problema del branching viene analizzato nelle minacce

¹Gli script sono disponibili su GitHub, al link <https://github.com/GerardoFesta/TesiRiusoDifettiEffort>

alla validità, nella Sezione 3.7.

Tornando all’analisi dei singoli commit, il Miner analizza tutti i file modificati per carpire le informazioni sulle righe aggiunte e rimosse - *i.e.* code churns -, il numero delle linee di codice, la complexity e il token count. Queste informazioni vengono organizzate e restituite all’interno di un dizionario le cui chiavi non sono altro che i nomi dei file, mentre i valori sono a loro volta dei dizionari che contengono le informazioni sopra descritte, più alcuni flag che riportano se il file è stato creato o eliminato in quel commit. Nel dizionario, inoltre, vengono inseriti anche i file non coinvolti dal commit, con un flag che indica, appunto, che il file non viene modificato. In più, si prendono in considerazione anche i file non .java modificati dal commit; questi verranno raggruppati (ovvero le loro informazioni, quali linee aggiunte e rimosse, vengono sommate come se si trattasse di un unico file) e inseriti nel dizionario con key ‘*altrifile*’.

Questo miner singolo verrà richiamato da un ‘*commitFeeder*’, ovvero una classe che scorre la lista di commit precedentemente generata in ‘*commitsInfo.csv*’, sfrutta il miner per ricevere i dati relativi ai file e gestisce i dati di ritorno. La manipolazione dei dati avviene per mezzo della libreria Pandas, che permette di gestire i dataset sotto forma di DataFrame in maniera completa. I dati vengono salvati in un file CSV che contiene tutti i commit e ad ogni commit associa tutte le classi (modificate e non dal commit) con i dati estratti dal miner, tenendo inoltre traccia della storia dei file, riportando dunque per ogni file se questo fosse o meno esistente quando la repository si trovava aggiornata a quel commit. Questo dataset può essere indicizzato per mezzo della coppia [commit, classe]. Infine, il *commitFeeder* accede alle informazioni sui bug precedentemente generate nel file <*NomeProgetto*>*Revisions.csv* e va a segnalare nel dataset i file modificati da un commit che introduce un difetto. Questo avviene per mezzo della colonna ‘Bug’ che avrà valore 1 solo laddove vengono rispettate le condizioni appena descritte. La stessa cosa avviene per i commit che risolvono un bug, mediante l’utilizzo della colonna ‘Fix’.

3.3.3 Preparazione dei dataset

I due dataset generati precedentemente sono ad una granularità ancora più fine di quella desiderata dallo studio. Questi, infatti, si trovano al *class-level*, mentre il livello desiderato è il *commit-level*. È necessario, logicamente, aggregare i dati. Tuttavia, prima di fare ciò è importante unire le informazioni dei due dataset.

Sapendo che il dataset più grande tra i due è il secondo, in quanto associa ad ogni commit tutti i file (anche quelli non esistenti al momento del commit), viene eseguito un merge del primo su questo, sfruttando la chiave formata da [commit, classe]. Questo passaggio è particolarmente importante perché permette di evitare informazioni importanti quali, ad esempio, il numero di righe rimosse in un commit che elimina dei file. La motivazione deriva dalla generazione dei due dataset: se l'*i*-esimo commit ha eliminato un file dalla repository, nel primo dataset questo file non sarà presente, in quanto vengono presi in considerazione solo i file effettivamente esistenti nel checkout della repository all'*i*-esimo commit. Facendo il merge del secondo dataset, in cui questo file è presente e riporta sia l'eliminazione sia il numero di righe rimosse, sul primo dataset questa informazione sarebbe stata persa.

Dunque, viene effettuato il salvataggio di queste informazioni inserendo nuove colonne apposite per contenerle, si rende il dataset omogeneo, eliminando tutte le righe che non sono presenti nel primo dataset. Vengono trasformate le colonne relative alle metriche di ereditarietà: invece del booleano 'True' o 'False', adesso assumono valore 1 o 0.

A questo punto si è pronti a risalire di granularità. Il dataset viene raggruppato sui commit - per mezzo di `groupBy` - e i dati aggregati per far in modo di avere, per ogni commit: *la somma, la media e la mediana di ogni singola metrica CK, la somma di delegations, di intInh e di impInh, un flag che indica se il commit ha introdotto un bug e un flag che indica se il commit ha corretto un bug, il numero di file presenti, il numero di righe aggiunte e rimosse*.

È bene notare come, dopo questa operazione, **le metriche di ereditarietà rappresentano per ogni commit il numero di classi che utilizzano l'ereditarietà, rispettivamente di specifica e di implementazione**.

3.4 RQ1: Analisi dell’evoluzione delle metriche di ereditarietà e delegazione

Per rispondere alla prima domanda di ricerca, viene analizzato l’andamento delle tre metriche oggetto dello studio (*i.e.*, intInh, implInh, e delegation). Siccome lo studio osserva le repository sin dall’*early stage*, i valori delle metriche vengono normalizzati per LOC (Lines of Code) e per numero di file. Questo permette di evitare uno studio banale nel caso in cui l’evoluzione del sistema fosse lineare: se col tempo un progetto software cresce nelle dimensioni, ci si può aspettare che incrementino anche il numero di classi che utilizzano l’ereditarietà (seppur questo possa non essere sempre vero). È bene invece valutare il rapporto tra le classi che utilizzano l’ereditarietà e il numero di file, in modo da ottenere l’andamento della frequenza relativa. La normalizzazione per LOC viene eseguita anche per continuità con il lavoro di *Giordano et al.* [1], anche se è da notare come le metriche di ereditarietà siano calcolate in maniera diversa. Nel suddetto, l’ereditarietà non veniva misurata infatti come numero di classi che utilizzano il meccanismo, ma in maniera più articolata. Data la natura del dataset in esame, tuttavia, ha senso utilizzare anche il numero di file per questa operazione.

3.5 RQ2: Correlazione tra utilizzo del riuso e presenza di difetti nel codice

Per dare una risposta a RQ2, viene definito un modello statistico che mette in relazione le metriche di riuso, insieme alle metriche della suite CK, alla presenza di bug. Per rendere il problema uno di classificazione, la presenza dei bug viene riportata in termini di incrementi e decrementi.

Response Variable/Variabile dipendente Questa viene rappresentata dalla presenza di bug. Per computarla viene effettuato un po’ di pre-processing. Inizialmente nel dataset è presente la colonna ‘Bug’ che ha valore 1 laddove il commit introduca un bug, 0 altrimenti, mentre la colonna ‘Fix’ ha valore 1 laddove il commit corregga un difetto. Per tipologia di bug che Defects4J propone, un difetto viene introdotto da un singolo commit senza vincoli sul tipo di file che quel commit può aver modificato. Questo significa che un bug potrebbe anche essere introdotto dalla modifica di un file di configurazione. Ciò tuttavia non è vero per il fix dei bug: infatti, se è vero che, un po’ come per i bug, il fix avviene in un singolo

commit, la rimozione del difetto viene effettuata modificando esclusivamente file .java, e quindi classi.

Per rendere possibile la classificazione, vengono fissati tre livelli: "Stable", "Increase" e "Decrease". Consideriamo l'i-esimo commit e indichiamo con $bug[i]$ l'i-esimo valore della colonna 'Bug' e con $fix[i]$ l'i-esimo valore della colonna 'Fix', mentre chiamiamo $tot[i - 1]$ il risultato ottenuto dalla computazione precedente. Calcoliamo $tot[i]$ come $tot[i] = tot[i - 1] + bug[i] - fix[i]$. Se $tot[i] = tot[i - 1]$ allora indichiamo l'i-esimo commit come "Stable", in quanto non c'è stata variazione nel numero di difetti; se invece $tot[i] > tot[i - 1]$ allora l'i-esimo commit verrà taggato con "Increase", ad indicare che il numero di bug è incrementato rispetto al commit precedente; infine, indichiamo con "Decrease" l'i-esimo commit laddove $tot[i] < tot[i - 1]$ in quanto un fix ha fatto calare il numero dei difetti presenti nel progetto. Per il primo commit $tot[0] = 0$.

Questo metodo, che computa i commit a coppie, si è reso necessario anche perché è possibile che un commit che corregge un difetto finisca per introdurne un altro. In quei casi il numero di bug resta, nei fatti, stabile.

La variabile così costruita rappresenta la response variable dell'esperimento.

Variabili indipendenti I fattori che si vogliono valutare sono le metriche del riuso, ovvero *impInh*, *intInh* e *delegations*, come definite nelle sezioni precedenti.

Variabili di controllo Naturalmente la variazione del numero dei bug potrebbe dipendere da altri fattori esterni alle metriche considerate come variabili indipendenti. In particolare, sono state prese in considerazione le seguenti metriche: DIT (Depth of Inheritance Tree), NOC (Number Of Children), LOC (Lines of Code), LCOM (Lack of Cohesion of Methods), WMC (Weighted Methods per Class), RFC (Response for a Class) e CBO (Coupling Between Objects).

Queste metriche, che inizialmente venivano calcolate per ogni file dei singoli commit, sono state aggregate per riportare l'analisi al *commit-level* e sono presenti nel dataset sotto forma di media, mediana e somma per ogni commit. In continuità con quanto fatto per la variabile dipendente, per la quale i commit vengono considerati a coppie per poter evidenziare la variazione del numero di difetti presenti, anche per queste metriche viene calcolata la variazione. In questo caso il calcolo è più semplice.

Prendiamo per esempio la metrica WMC: indichiamo il valore (medio) di WMC per l'i-esimo commit con $WMC[i]$. Calcoliamo - e successivamente utilizziamo nel modello - la variabile

$diffWMC[i] = WMC[i] - WMC[i - 1]$. Anche in questo caso, per la prima riga il valore di $WMC[0]$ è 0.

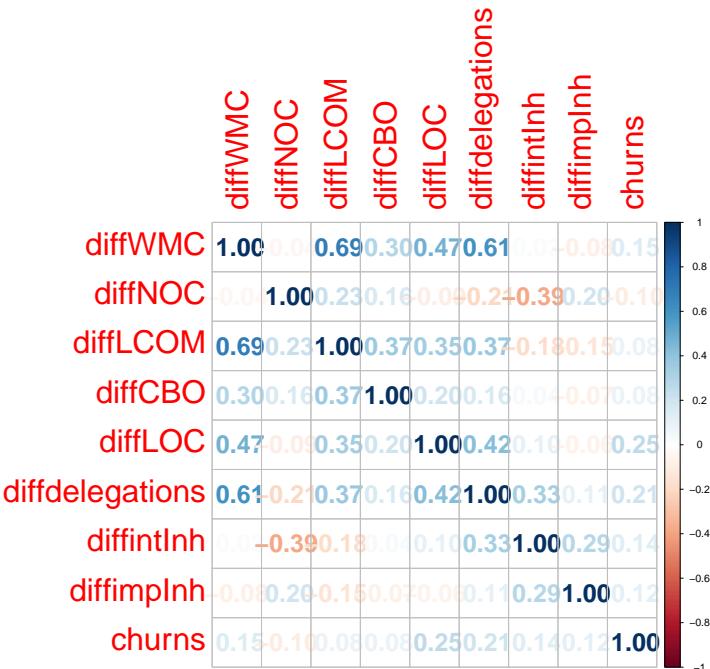
Oltre alle suddette metriche vengono utilizzate le linee aggiunte e le linee rimosse dai commit. Tuttavia, a differenza di quanto accade per le features viste in precedenza, in questo caso non c'è bisogno di calcolare la variazione. Questo perché il numero di linee aggiunte e rimosse da un commit rappresenta intrinsecamente una variazione dello stato della repository rispetto al commit precedente.

È bene sottolineare l'utilizzo di metriche quali NOC e DIT, concettualmente legate al riuso e all'ereditarietà e quindi alle variabili definite come indipendenti. Questo può essere visto sia come un modo per integrare la visione specifica di riuso con una più globale e, in generale, molto spesso adottata in letteratura, ma anche come una possibile baseline, sulla quale valutare la potenza predittiva delle variabili introdotte.

Esecuzione del modello statistico Avendo impostato la variabile dipendente sui tre livelli "Stable", "Increase" e "Decrease", viene utilizzato un modello multinomiale log-lineare. Questo, implementato in R grazie al pacchetto *nnet* - che contiene funzioni che sfruttano le reti neurali - e più in dettaglio grazie alla funzione *multinom*, è un metodo di classificazione applicabile laddove la response variable sia categorica e composta da almeno due livelli.

Il modello multinomiale ha bisogno di avere un livello di riferimento, in questo caso "Stable", sul quale poi restituisce dei coefficienti per ogni variabile dipendente che indicano come le variabili influiscono sul passaggio dal livello di riferimento agli altri livelli, in questo caso "Increase" e "Decrease". Il risultato sarà dunque una tabella con alle righe le metriche e sulle colonne "Increase" e "Decrease". Ad esempio, un coefficiente positivo per la variabile *diffWMC* nella colonna "increase", significa che un'incremento unitario della variabile in questione porterebbe ad un aumento della probabilità che venga introdotto un bug, con l'aumento che è rappresentato proprio dal coefficiente.

È stato preso in esame il possibile problema della multicollinearità, e si è scelto di verificare le relazioni tra le variabili indipendenti per mezzo della matrice di correlazione, di cui la Figura 3.1 è un esempio. Laddove venga trovata una correlazione tra due variabili superiore allo 0.75, viene rimossa una delle due variabili dal modello. La scelta ricade su quella la cui rimozione permette di risolvere più relazioni superiori allo 0.75, ma si tiene in considerazione anche la correlazione delle due variabili indipendenti con quella dipendente, eliminando, nel caso, quella con correlazione minore.

**Figura 3.1:** Esempio di matrice di correlazione

3.6 RQ3: È possibile predire l'effort necessario per la risoluzione dei difetti?

Per dare una risposta a RQ3, viene definito un modello statistico che mette in relazione le metriche di riuso, insieme alle metriche della suite CK e alla presenza di bug, ai code churns, ovvero la somma delle linee aggiunte e rimosse da ciascun commit. In questo caso, dunque, si vuole predire una variabile continua e per questo motivo viene scelto il Generalized Linear Model che permette di rilassare le assunzioni su cui si basa la regressione lineare. Anche in questo caso viene verificata la collinearità dei dati sfruttando la correlation matrix, proprio come avviene per RQ2.

3.7 Minacce alla validità

Validità di impostazione Questa tipologia di minaccia si riferisce alla relazione tra teoria e pratica, o osservazione. Principalmente questa è legata al dataset utilizzato. La scelta dei progetti da analizzare può essere un punto di debolezza per l'esperimento, tuttavia questi progetti sono ben noti in letteratura in quanto parte di un dataset molto utilizzato, quale Defects4j, il cui utilizzo viene documentato nella Sezione 3.2. Il processo che individua i bug potrebbe essere oggetto di criticità, ma questi provengono proprio da Defects4j, per cui vale quanto riportato sopra. Anche il calcolo delle metriche potrebbe rappresentare una minaccia, ma questo viene fatto sulla base della loro definizione classica, laddove disponibile. Le metriche non presenti in letteratura, ovvero quelle di ereditarietà e delegazione, sono nuove introduzioni; tra le tre, quella più soggetta a opinabilità è la delegazione. Questa dovrebbe essere calcolata come la somma delle chiamate a metodi delle variabili d'istanza, mentre nello studio viene rilassato questo vincolo, in quanto si considerano tutte le chiamate a metodi non della classe in esame. Si tratta dunque di un'approssimazione della delegation vera e propria.

Oggetto di analisi è stato il problema del branching che si presenta quando si effettua mining delle repository GitHub. Il branching è un fattore che può cozzare con un'analisi in termini di evoluzione della repository, in quanto quest'ultima va fatta su base cronologica, mentre Github gestisce la struttura dei commit mediante albero. Quando si effettua un merge di un branch secondario all'interno del main branch, i commit del primo rientrano nel main, andando potenzialmente a frapporsi tra commit già presenti nel main branch, rompendo dunque - per un certo numero di commit - il concetto di sequenzialità padre-figlio dei commit nel dataset. Si tratta di un problema già noto in letteratura. Diversi studi hanno comparato l'utilizzo del solo main branch, scartando dunque i commit che provengono da altri branch a seguito di un merge, e di tutti i branch. I risultati dimostrano che, a livello di defect prediction, considerare tutti i branch comporta un non significativo aumento delle performance [39]. D'altro canto, eliminare i commit provenienti da altri branch a seguito di un merge, o l'utilizzo di tecniche di ordinamento per cercare di ripristinare la sequenzialità padre-figlio, avrebbe introdotto ulteriori minacce alla validità. Con l'eliminazione, infatti, sarebbero andati persi diversi bug del dataset e inoltre questo avrebbe inciso sulla predizione dell'effort, non considerando dei commit che, nei fatti, sono stati effettuati. L'ordinamento avrebbe potuto sì ri-approssimare l'ordinamento padre-figlio ma avrebbe spezzato, dall'altro lato, l'ordine temporale che viene utilizzato.

Validità interna Le minacce alla validità interna riguardano i fattori che possono influenzare i risultati. Per RQ2 e RQ3 vengono utilizzate diverse variabili di controllo per poter stimare in maniera più sicura l'effetto delle metriche di riuso sulla presenza dei bug per RQ2 e l'effort per la risoluzione dei bug per RQ3. Le metriche sono ampiamente note in letteratura: trattasi delle metriche CK, già utilizzate in più studi per la predizione della presenza di difetti o la fault-proneness.

Validità di conclusione Le minacce alla validità di conclusione riguardano principalmente i modelli statistici utilizzati. Per RQ2 è stato scelto il Multinomial Logistic Linear Model, che si presta bene a gestire una variabile dipendente categorica a tre livelli e un insieme di variabili indipendenti continue. Prima di eseguire il modello, per ogni progetto viene verificata la collinearità dei dati, ovvero si verifica se ci sono delle variabili indipendenti che sono correlate e che potrebbero portare il modello ad inflazionare alcune variabili. Il test viene eseguito analizzando la matrice di correlazione di Kendall e osservando le variabili a coppie, scartando poi una delle due variabili laddove il coefficiente di correlazione fosse superiore allo 0.75 (la scelta della variabile da scartare è meglio spiegata nella sezione 3.5). Per RQ3, invece, trattandosi di variabile dipendente continua e variabili indipendenti sia continue che categoriche, viene scelto il Generalized Linear Model, che permette di rilassare le assunzioni della regressione lineare. Anche in questo caso, però, viene utilizzato il test per gestire la multicollinearità. Dunque, le assunzioni dei modelli sono validate.

Validità esterna Le minacce alla validità esterna si focalizzano sulla possibilità di generalizzare i risultati ottenuti. Purché venga analizzato un numero consistente di sistemi, undici, non è possibile dichiarare i risultati generalizzabili per via della mole e della diversità dei progetti Open Source Java presenti.

CAPITOLO 4

Risultati

In questo capitolo vengono mostrati e analizzati i risultati ottenuti per RQ1, RQ2, e RQ3 su tutti i software oggetto dello studio.

4.1 Risultati RQ1

4.1.1 Codec

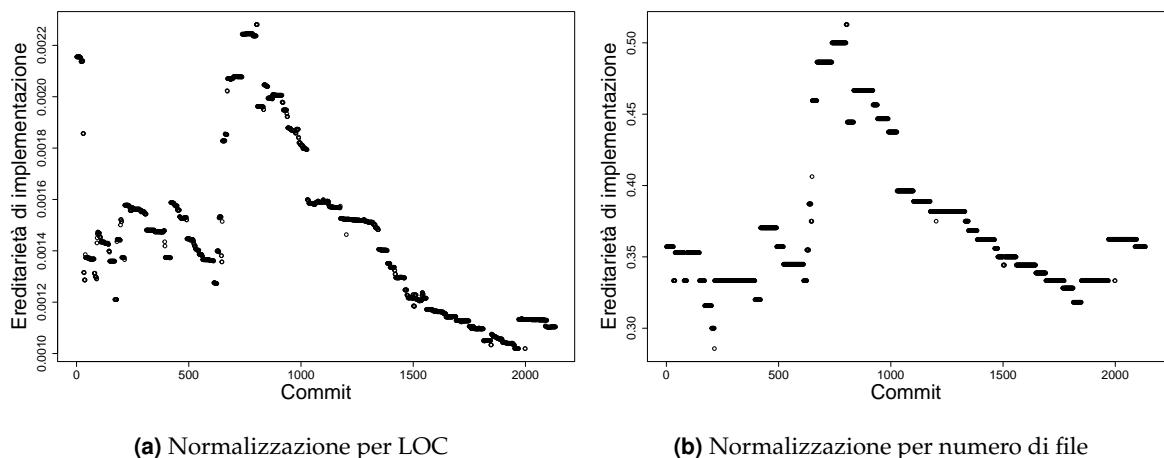


Figura 4.1: Codec - Ereditarietà di implementazione

RQ1₁. I grafici in Figura 4.1 mostrano un andamento piuttosto simile. Inizialmente, il grafico normalizzato per LOC ha un picco, mentre quello normalizzato per numero di file resta

piuttosto stabile. Questo è dovuto al fatto che il primo commit della repository inserisce 14 file, di cui 5 utilizzano l'ereditarietà di implementazione, a fronte di meno di 2500 righe di codice. Nei commit successivi, però, vengono aggiunte più di 1000 righe di codice, mentre il numero dei file (e di metriche) resta invariato. Questo giustifica la discesa dal primo picco nei grafici in Figura 4.1. Il successivo tratto di stallo, o comunque di non significativi incrementi e decrementi, è presente perché la metrica dell'ereditarietà di implementazione cresce in maniera piuttosto stabile rispetto al numero di file/linee di codice, con aggiunte di file e linee di codice graduali. Il secondo picco, invece, è comune a entrambe le normalizzazioni, ed è significativo. Questo deriva da un periodo di poco rework o aggiunta di codice. Le uniche occasioni in cui viene aggiunto un numero sostanzioso di linee di codice è con la creazione di nuovi file. Questi file che vengono creati però, sono spesso parte di una gerarchia che parte dalla classe AbstractCavephone. Il declino sembra iniziare al commit numero 806, che inizia ad aggiungere una buona quantità di nuovo codice in 6 nuovi file (su un totale, in quel momento, di 45) senza che nessuna di queste classi utilizzi meccanismi di ereditarietà. Da lì la discesa prosegue in maniera simile, anche se nel grafico in Figura 4.1(a) la discesa è più ripida. Questo è giustificato dal fatto che la repository, superati i 1100 commit, vede una modifica molto lenta (e rara) del numero dei file, mentre il codice già esistente subisce parecchie modifiche. La risalita finale è dovuta all'aggiunta di 3 file non molto grandi nel commit 1970 che sfruttano l'ereditarietà di implementazione.

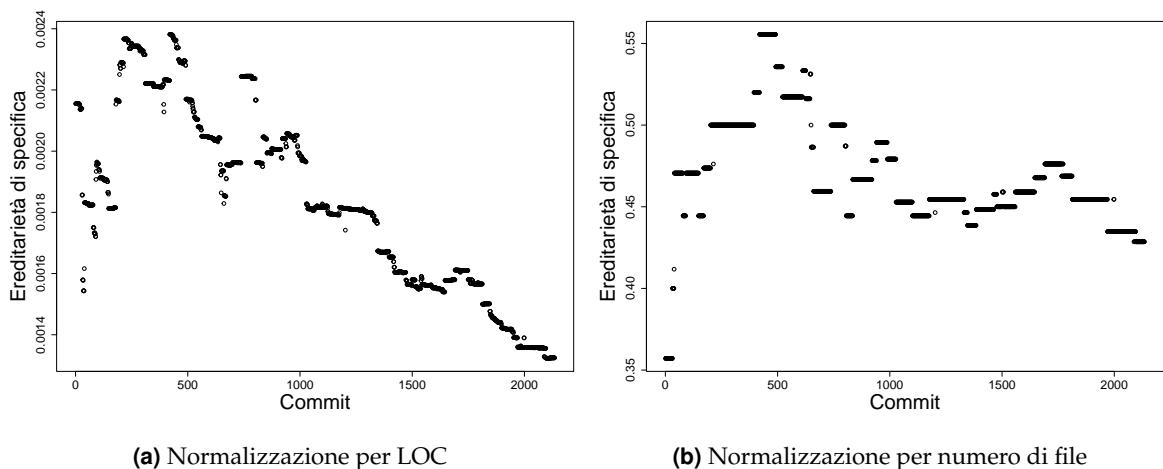


Figura 4.2: Codec - Ereditarietà di specifica

RQ1₂. Anche i grafici in Figura 4.2 sono di forma simili e, inoltre, somigliano anche ai grafici dell'ereditarietà di implementazione, ovvero quelli in Figura 4.1. Il picco in questo

caso viene raggiunto leggermente prima e in maniera meno ripida. Questo succede perché l'ereditarietà di specifica cresce in maniera molto più graduale rispetto alla controparte, e dunque questa metrica varia più spesso, soprattutto in fase iniziale. Inizialmente, infatti, entrambe le ereditarietà sono utilizzate in 5 classi diverse, ma intorno ai 250 commit, vediamo come l'ereditarietà di specifica venga utilizzata in 12 classi, mentre quella di implementazione in 8 (a fronte di un aumento di 10 file in totale). In questa fase vengono create molte classi di Codec/Decodec specifiche, che implementano da interfacce generali.

In seguito, il trend è in discesa per gli stessi motivi descritti per l'ereditarietà di implementazione. L'andamento vede poi un momento di stallo, più visibile in Figura 4.2(b), causata da un periodo in cui pochi file vengono introdotti e non c'è sostanziale modifica della metrica. La breve risalita che avviene tra i 1700 e i 1800 commit è dovuta all'introduzione di due file (su due in totale introdotti) che implementano due interfacce, ma gli eventi non sono collegati.

Possiamo dunque concludere che **per le due ereditarietà il trend è simile**, con un iniziale impiego più sostanziale (in relazione, naturalmente, all'evoluzione del sistema), seguito da un declino quando il progetto raggiunge una certa maturità. In Codec **l'ereditarietà di specifica è più utilizzata rispetto a quella di implementazione**, complice la definizione di molti tipi di Decoder/Encoder che ereditano da interfacce.

Possiamo anche osservare come, seppur con una modulazione diversa, gli andamenti dei due tipi di normalizzazione abbiano un trend piuttosto simile, con la normalizzazione per LOC che presenta, però, una decrescita più rapida e netta rispetto alla normalizzazione per numero di file.

RQ1₃. La delegazione, illustrata nei grafici in Figura 4.3, segue un trend diverso dalle due metriche di ereditarietà. Infatti, seppure abbia una crescita iniziale che culmina in un picco in maniera simile a quanto accade per le altre due metriche, e poi inizi un processo di discesa, questo si interrompe molto velocemente, con un aumento piuttosto repentino. La crescita avviene, di nuovo, in relazione al grande effort che porta la repository in una prima situazione di stabilità. Intorno agli 850 commit avviene la modifica che porta di nuovo i grafici a crescere. Si tratta di un'aggiunta non sostanziale, si parla di circa 250 linee di codice aggiunte, ma fa crescere abbastanza la metrica. In seguito, possiamo vedere la differenza sostanziale con le metriche di ereditarietà. Il rework a seguito del quale i grafici delle altre due metriche crollano, non incide particolarmente sulla delegazione, che nel caso del grafico in Figura 4.3(a) resta piuttosto stabile, indice del fatto che aggiunte e rimozioni di linee di

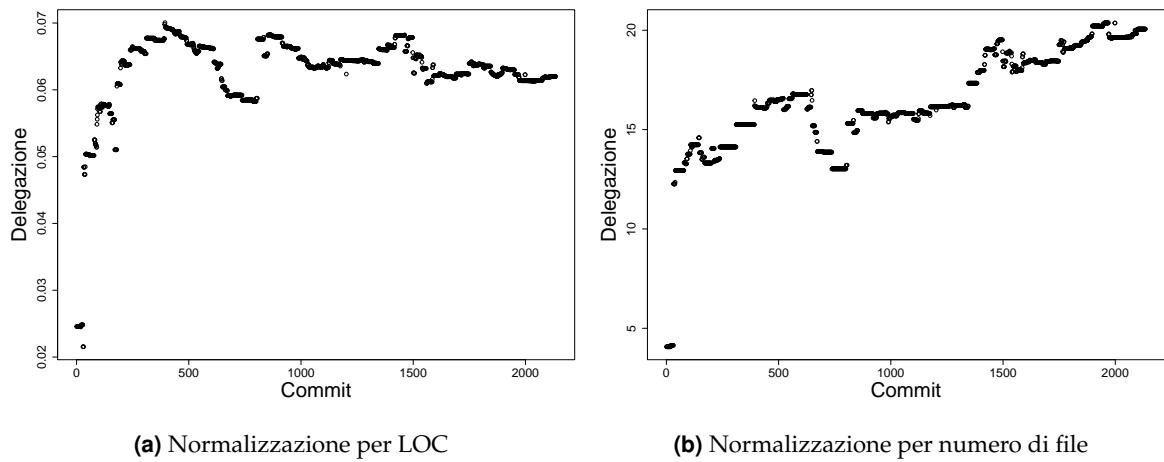


Figura 4.3: Codec - Delegazione

codice comportano una modifica coerente della metrica, mentre nel grafico in Figura 4.3(b) c'è addirittura una crescita. Il motivo di questa è già stato illustrato per le metriche di ereditarietà: c'è poca modifica nel numero dei file ma il rework, che principalmente aggiunge linee di codice a file già esistenti, fa comunque aumentare la delegazione.

Possiamo concludere che, per Codec, **il trend della delegazione non segue quello delle altre due metriche**, tendendo, una volta raggiunto il primo apice, a restare piuttosto stabile.

4.1.2 Cli

I grafici di Cli hanno richiesto la rimozione di due outlier, presenti ai commit 387 e 388. In questi due commit venivano rimosse tutte le linee di codice e veniva compattato tutto su una sola riga. In seguito a questi due commit, i file vengono ripristinati allo stato precedente. Questo portava le metriche (rapportate al numero delle linee di codice) ad avere valori molto più alti rispetto ai restanti punti, rendendo invisibili le altre variazioni.

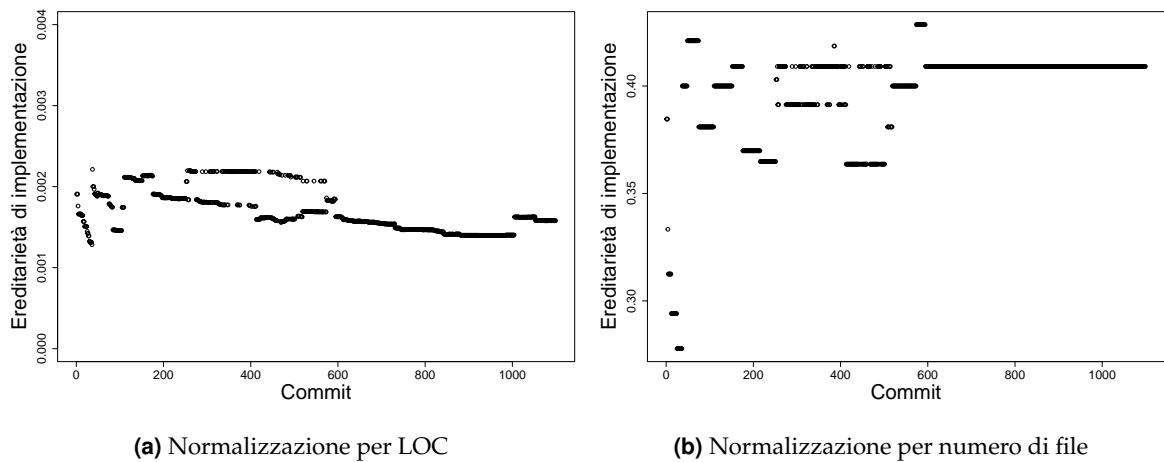


Figura 4.4: Cli - Ereditarietà di implementazione

RQ1. Inizialmente 5 classi (su 13 classi totali) utilizzano l'ereditarietà di implementazione. L'iniziale calo è dovuto ad una prima fase di sviluppo in cui, sebbene non vengano introdotti moltissimi file o linee di codice, vede l'ereditarietà ferma, per cui il rapporto di presenza scende. Dopo i primi 37 commit, tuttavia, si raggiunge il picco di presenza dell'ereditarietà, con l'introduzione di 2 sole nuove classi a fronte di un aumento di 3 della metrica (la terza unità di incremento è dovuta alla modifica di una classe già presente). Per la normalizzazione su numero di file possiamo vedere un ulteriore incremento, dovuto al drop di un file non coinvolto dall'ereditarietà, mentre per la normalizzazione su linee di codice si osserva un trend di leggera decrescita, dovuto a un generale incremento del volume di codice a fronte di una mancata introduzione di nuovi file.

La normalizzazione mitiga l'effetto del branching, discusso in Sezione 3.7, soprattutto quando si osserva il grafico relativo alla normalizzazione su LOC, ovvero quello in Figura 4.4(a), mentre la normalizzazione su numero di file mostrata in Figura 4.4(b) risulta meno leggibile. L'effetto è comunque visibile tra i 200 e i 600 commit, laddove si può vedere come l'evoluzione segua due tracce distinte. Indagando sull'accaduto, si scopre come gli sviluppatori abbiano,

ad un certo punto, deciso di dividere due versioni del software precedentemente presenti nello stesso branch. La versione CLI 2.0 è quella che si trova più in alto, la versione CLI 1.0 è quella più in basso.

Infine, si tende a continuare sul filone di CLI 1.0, con un trend leggermente verso il basso per quanto concerne il grafico di più semplice analisi, ovvero quello in Figura 4.4(a), con un incremento leggero sul finale, dovuto alla rimozione di 1000 linee di codice. Sul fronte del grafico in Figura 4.4(b), invece, c'è una stabilizzazione: nessun file viene rimosso o aggiunto nell'ultimo lasso di tempo.

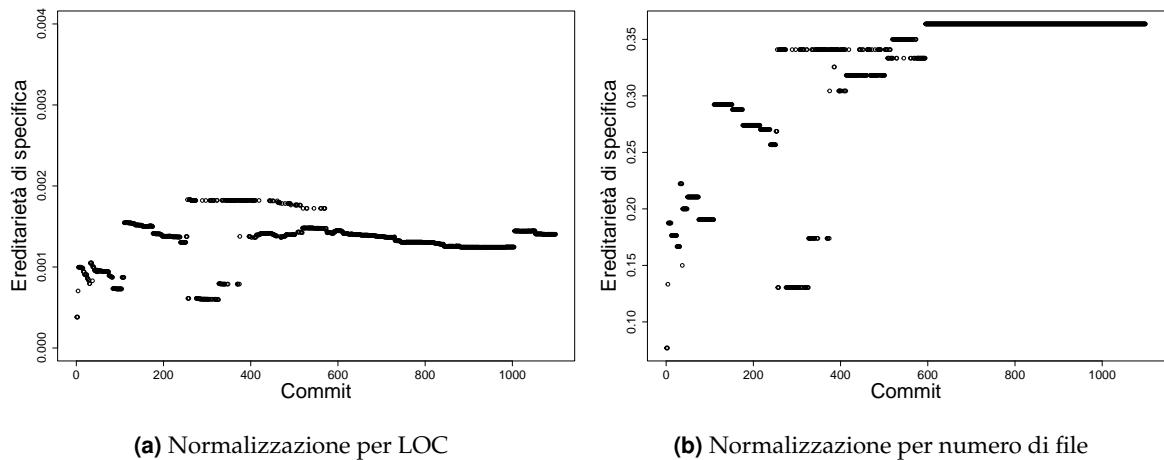


Figura 4.5: Cli - Ereditarietà di specifica

RQ1₂. Come visibile dai grafici in Figura 4.5, l'andamento dell'ereditarietà di specifica è del tutto simile a quello dell'ereditarietà di implementazione. A questo punto, è bene soffermarsi sulle differenze.

La differenza più marcata si trova tra i 200 e i 400 commit, laddove ci sono dei punti più in basso di quanto ci si aspettasse. La motivazione è di nuovo il branching. L'impatto in questo caso è più netto. Nel branch più voluminoso, infatti, ha poco meno del doppio del numero di file rispetto al branch meno numeroso. Tuttavia, mentre l'ereditarietà di implementazione diminuisce della metà nel passaggio di branch, l'ereditarietà di specifica perde ben i 4/5 del suo valore. Successivamente, però, la metrica cresce e si riporta su un path assimilabile a quello dell'altra metrica di ereditarietà.

Possiamo concludere che, in linea di massima, **l'ereditarietà di implementazione viene usata leggermente in più dell'ereditarietà di specifica**, seppure il progetto non usi nessuna delle

due in maniera massiccia. Dopo un'iniziale crescita, inoltre, **si raggiunge la stabilità**, per cui l'uso dell'ereditarietà non aumenta né diminuisce.

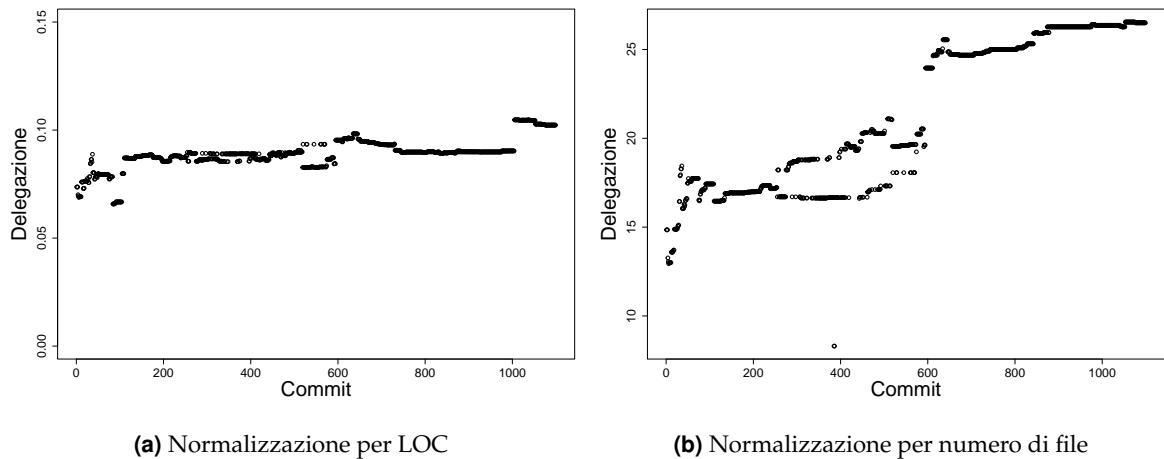


Figura 4.6: Cli - Delegazione

RQ1₃. La delegazione presenta delle differenze notevoli tra i grafici in Figura 4.6(a) e Figura 4.6(b) delle normalizzazioni. Mentre, infatti, per la normalizzazione su LOC l'evoluzione resta piuttosto stabile, l'evoluzione normalizzata sul numero di file mostra un trend crescente (al netto del fattore branching). Questo succede perché a fronte di poche modifiche al numero di file, il lavoro di rework e refactoring, soprattutto dal punto di vista dell'aggiunta di codice in classi già esistenti, è predominante. In continuità con quanto fatto per le due metriche precedenti, la valutazione avviene sul grafico con la normalizzazione per NOC.

Per Cli **l'evoluzione della delegazione è piuttosto lineare**, con un rapporto d'utilizzo che resta stabile nel corso della storia della repository.

4.1.3 CommonsCollections

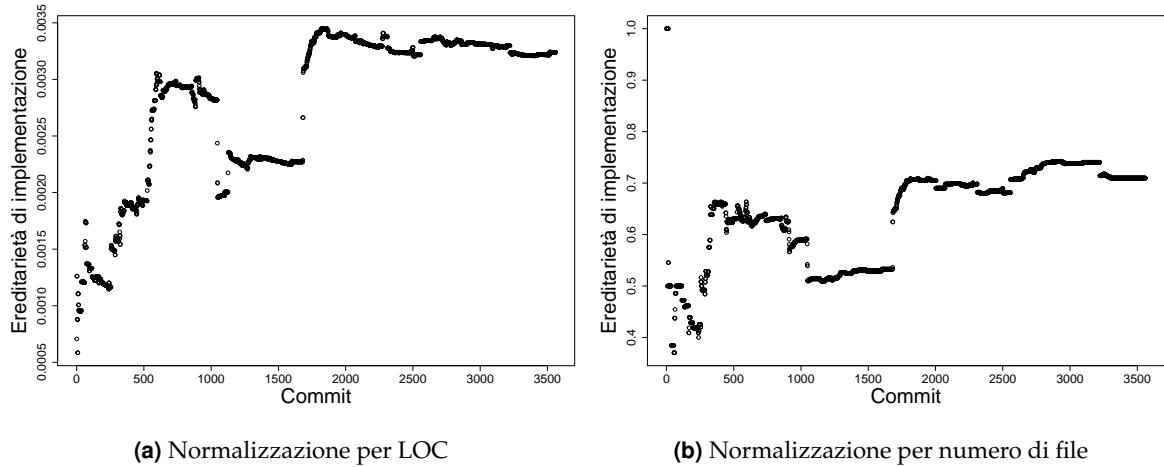


Figura 4.7: CommonsCollections - Ereditarietà di implementazione

RQ1₁. L’andamento dei due grafici per l’ereditarietà di implementazione, in Figura 4.7, risultano simili, seppur con una modulazione verso il basso per il grafico (b). Questo probabilmente è causato dalla situazione in avvio: a differenza delle repository viste finora, CommonsCollections parte con un singolo file, per poi incrementare in maniera molto graduale (almeno inizialmente). Tutti i primi file inseriti sfruttano l’ereditarietà di implementazione e questo spiega la partenza altissima del grafico. L’incremento prosegue (al netto di un breve calo dovuto all’aggiunta di un buon numero di file e linee di codice) fino a raggiungere un primo picco e un breve periodo di stabilità. Intorno al commit 1050 c’è un calo netto, che coinvolge una decina di commit prima di trovare una fase di stabilità. Il crollo è dovuto alla rimozione di un pacchetto non più utilizzato e proveniente da un altro progetto Apache, ovvero commons-primitives, i cui file erano stati inseriti gradualmente. È evidente che questi file utilizzassero l’ereditarietà di implementazione in maniera piuttosto marcata, altrimenti invece di un drop ci sarebbe stato un incremento o uno stallo. Intorno ai 1700 commit, viene effettuato un merge manuale di un branch, che comporta molto rework, e stavolta coinvolge anche l’ereditarietà in positivo. A fronte di un aumento di un solo file e di circa 300 linee di codice, la metrica subisce un’incremento di 25 unità. Tuttavia questo commit non è quello che fa raggiungere il picco massimo, bensì è quello che si trova al centro del gap. La metrica migliora la sua presenza in percentuale pochi commit dopo, quando vengono eliminati molti file dalla repository. Al netto di brevi e poco significative crescite e decrescite, la presenza dell’ereditarietà di implementazione tende a rimanere costante nel finale

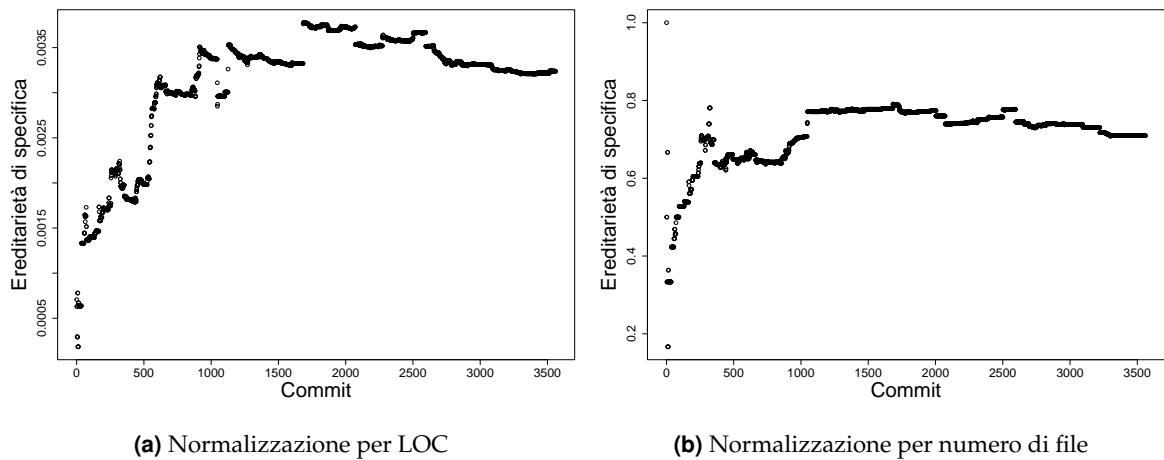
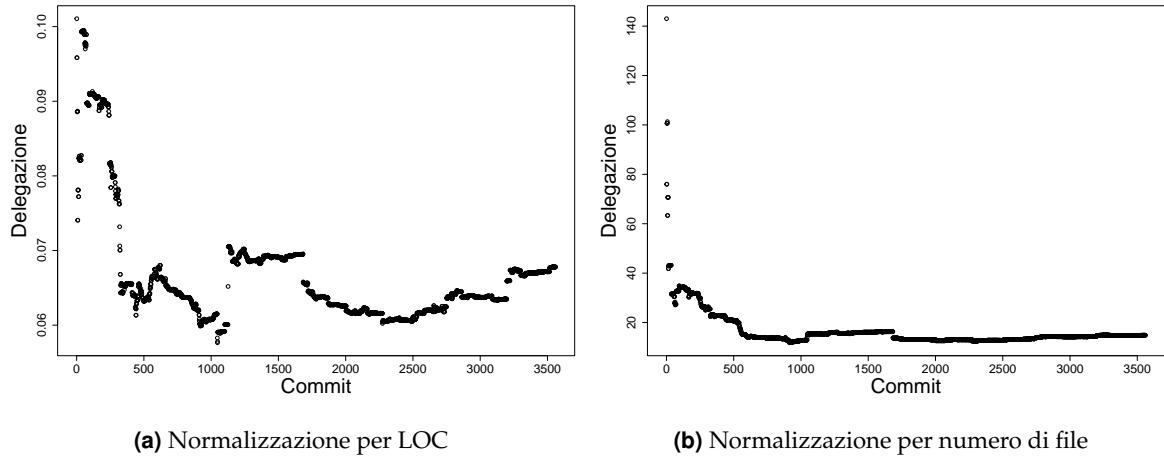


Figura 4.8: CommonsCollections - Ereditarietà di specifica

RQ1₂. L'ereditarietà di specifica, come mostrato nei grafici in Figura 4.8 progredisce in maniera molto simile all'ereditarietà di implementazione, pur non subendo il drop che avviene in Figura 4.7 tra i 1000 e i 1500 commit. Questo implica che la presenza dell'ereditarietà di specifica all'interno del pacchetto eliminato era di gran lunga inferiore a quella dell'ereditarietà di implementazione. Non ci sono particolarità diverse da analizzare, ma si può notare come, sul finale, **le due metriche terminano con lo stesso valore**. Infatti, il progetto arriverà agli utimi commit con 237 classi che fanno uso di 'implements' e 237 classi che fanno uso di 'extends'.

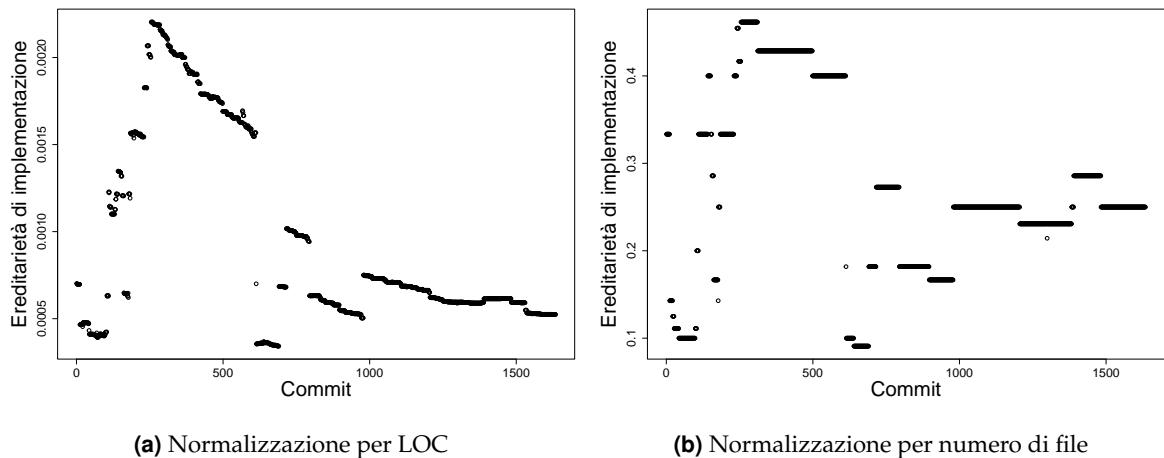
Per CommonsCollections dunque, **il riuso espresso in termini di ereditarietà cresce e raggiunge un picco, per poi assestarsi su valori leggermente più bassi**.

RQ1₃. Anche i due grafici in Figura 4.9 che mostrano l'andamento della delegazione hanno andamento simile, con la normalizzazione per numero di file che però è rimodulata, a causa di un range di valori considerato più alto. La motivazione è la stessa spiegata per l'eredità di implementazione. È sicuramente importante evidenziare come il trend sia diverso rispetto alle altre due metriche, indice che non è presente correlazione tra queste e la delegazione. Al contrario dell'ereditarietà di implementazione, giova della rimozione del pacchetto esterno, con un aumento destinato poi ad essere quasi annullato. Il trend non è chiarissimo. Se è vero che c'è una importante discesa iniziale, è anche vero che in seguito ci sono diverse oscillazioni. Considerando anche il grafico relativo alla normalizzazione per numero di file, però, possiamo vedere come le variazioni non siano poi così evidenti.

**Figura 4.9:** CommonsCollections - Delegazione

4.1.4 CommonsCsv

CommonsCsv è un progetto di piccole dimensioni, con pochi file presenti che, soprattutto nelle fasi iniziali vengono aggiunti e rimossi con molta frequenza. Questo comporta una difficoltà aggiuntiva nella lettura dei grafici. Considerando come le metriche di riuso sono calcolate e sapendo che il progetto lavora molto sullo stesso insieme di file, ci si può comunque aspettare che i grafici delle due ereditarietà risultino calanti da un certo punto in poi, mentre sulla delegazione non si può ipotizzare molto fare molte ipotesi.

**Figura 4.10:** CommonsCsv - Ereditarietà di implementazione

RQ1₁. I grafici in Figura 4.10 sono simili nell’andamento, eccetto per il finale, che vede la normalizzazione per numero di file mantenere un livello più alto rispetto alla normalizzazione

per LOC. Il picco viene raggiunto quando, intorno al commit 250, quasi la metà delle classi (5 su 11) utilizzano l'ereditarietà di implementazione. Per la prima parte del periodo che segue, l'ereditarietà in sé non subisce grandi modifiche, vengono aggiunte linee di codice e un file. Il crollo netto è dovuto dall'eliminazione di quasi un terzo dei file (4 su 15). Tutti e quattro i file utilizzavano l'ereditarietà di implementazione, per questo motivo questa, nel commit in esame e nel commit seguente, finisce per essere quasi azzerata. La risalita avviene a scalini, a causa della scarsità di modifiche e aggiunte che vengono fatte al sistema a fronte di un buon numero di commit. Dopo aver raggiunto il tetto massimo, comunque, il trend sembra di generale decrescita.

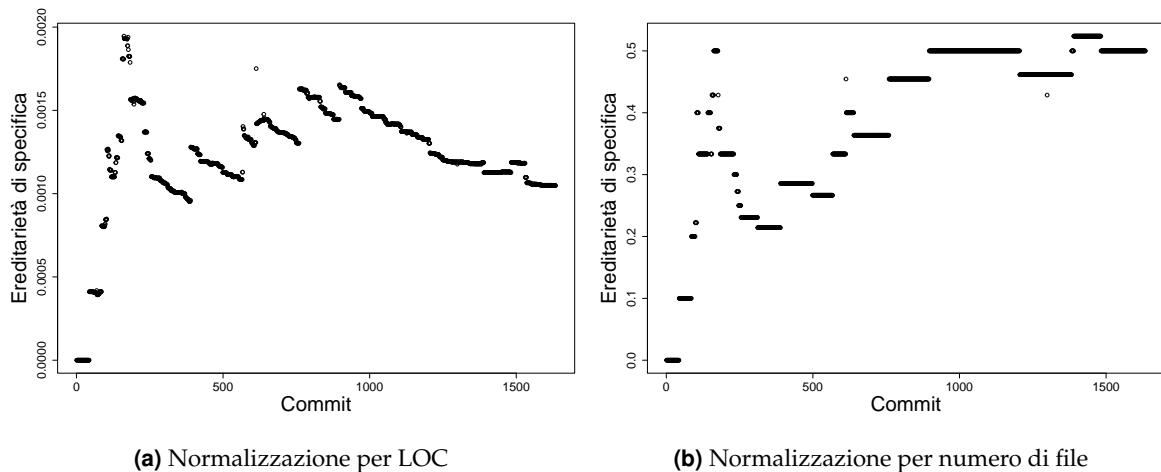
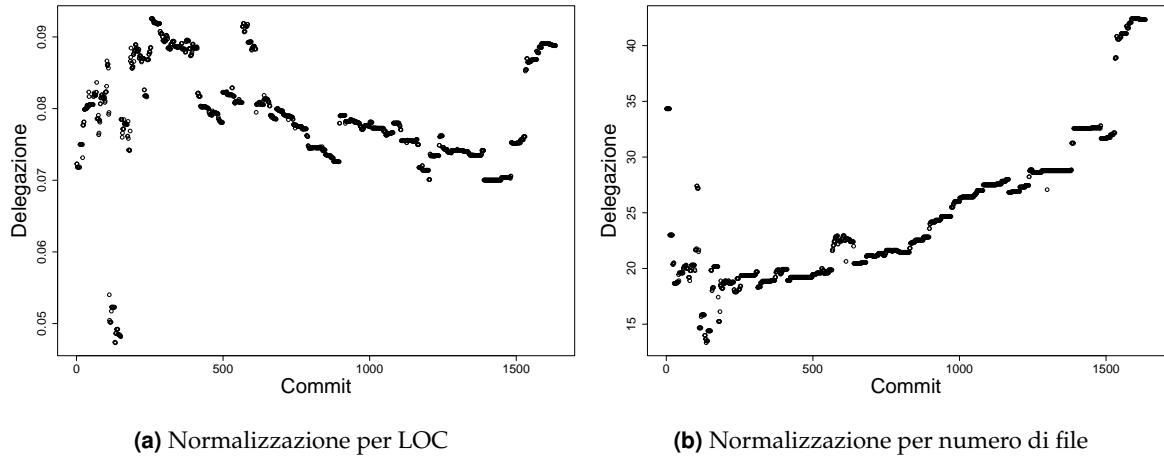


Figura 4.11: CommonsCsv - Ereditarietà di specifica

RQ1₂. I grafici in Figura 4.11 condividono con i grafici in Figura 4.10 la crescita iniziale e l'inizio della decrescita dopo il picco. Tuttavia, laddove l'ereditarietà di implementazione crolla, l'eredità di specifica cresce, banalmente perché il numero dei file/linee di codice al denominatore della normalizzazione è calato, mentre il numeratore, ovvero il numero di classi che utilizzano la keyword 'implements' resta invariato. Il finale è diverso a seconda del grafico che si osserva: in crescita se si osserva il numero dei file, in decrescita altrimenti.

In CommonsCsv, **l'ereditarietà di specifica viene utilizzata mediamente il doppio rispetto a quella di implementazione.**

RQ1₃. La Figura 4.12 mostra le due normalizzazioni applicate sull'evoluzione della delegazione. L'andamento, oltre ad essere molto diverso tra le due, è anche differente da quanto visto per l'ereditarietà. Il rapporto con il numero dei file era intuibile, in quanto questo

**Figura 4.12:** CommonsCsv - Delegazione

resta relativamente basso e ci sono tanti commit di rework/modifica di classi già esistenti. Il rapporto con le linee di codice invece non è propriamente stabile, ma vive un primo momento di aumento seguito da una leggera discesa. Tra i due grafici, però, mostra un insieme di punti di minimo in comune. Questi nascono con il commit 112. Questo commit rimuove da 3 classi alcune variabili d'istanza e le rimpiazza con classi statiche, andando in seguito a diminuire di molto le chiamate a metodi che vengono effettuate. Dopo 40 commit, tuttavia, i rapporti si risolvono. Il finale, in entrambi i grafici in crescendo, nasce da un rework abbastanza importante su poche classi, che aumenta molto la metrica in questione.

4.1.5 Compress

Per quanto dai grafici non sia palesemente visibile, Compress soffre leggermente del problema del branching. Tuttavia questo non sembra aver invalidato in alcun modo i grafici, che si presentano abbastanza chiari.

RQ1₁. L'andamento dei due grafici in Figura 4.13 è quasi identico. Inizialmente gli aumenti avengono per step, in maniera non continua. Inizialmente dunque, ci sono pochi commit, ma modificano molte cose. Il picco viene raggiunto con il commit 108, che importa un branch di redesign. Successivamente si introduce un nuovo branch. La normalizzazione però rende quasi invisibile questa differenza. Ciò significa che anche il branch secondario segue il trend del main branch.

La variabile da questo momento fatica ad aumentare di valore, mentre l'introduzione di nuovi file e un significativo lavoro di aggiunta di metodi alle classi già esistenti portano

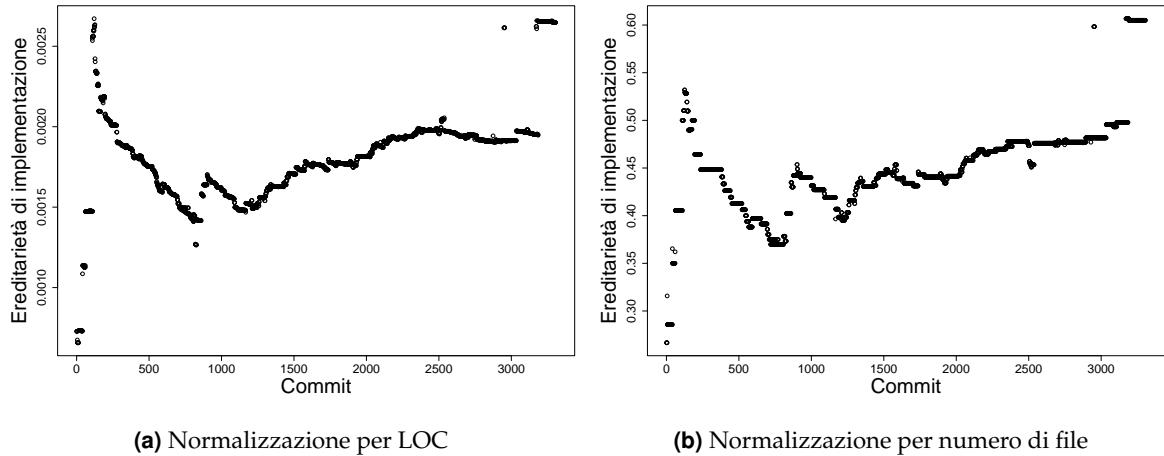


Figura 4.13: Compress - Ereditarietà di implementazione

a un rapido declino della presenza dell'ereditarietà di implementazione. Il commit 860 è quello che pone fine alla discesa, permettendo una rapida risalita. Questo commit introduce la compatibilità con un particolare package, e per arrivare a ciò è necessario estendere diverse classi. Anche i successivi commit aggiungono il supporto ad altri package, facendo giungere la metrica al secondo picco.

Dopo un breve periodo di discesa, l'ereditarietà di implementazione vive una crescita piuttosto costante, con un picco finale che avviene a seguito di una catena di merge, che porta su tutti i valori. Tuttavia, ci sono troppi pochi elementi per poter dire se l'ereditarietà di implementazione assume effettivamente un ruolo più importante, come sembra dal grafico, o se si tratta di un picco momentaneo.

RQ1₂. Anche l'ereditarietà di specifica non presenta sostanziali differenze tra le due normalizzazioni, visibili nei grafici in Figura 4.14. L'evoluzione iniziale è uguale a quella dell'ereditarietà di implementazione, tuttavia, mentre quest'ultima ha una rapida ripresa, l'ereditarietà di specifica decresce di più. In tutte le modifiche effettuate in questo lasso di tempo, molto raramente la metrica subisce dei cambiamenti ma sia il numero dei file che le linee di codice aumentano.

Dopo una ripresa non costante (il fattore branching non aiuta in questo caso), c'è uno scalino verso l'alto, dovuto alla creazione di una gerarchia con alla radice l'interfaccia InputStreamStatistics. Possiamo notare come però l'ereditarietà di specifica non ha un salto verso l'alto alla fine, e dunque è plausibile che venga utilizzata con ruolo centrale l'ereditarietà di implementazione nei branch di cui è stato fatto il merge. Alla fine dei commit analizzati, l'eredità

di implementazione è utilizzata più del doppio dell'eredità di specifica, con questa che si è inoltre dimostrata meno costante nel processo evolutivo.

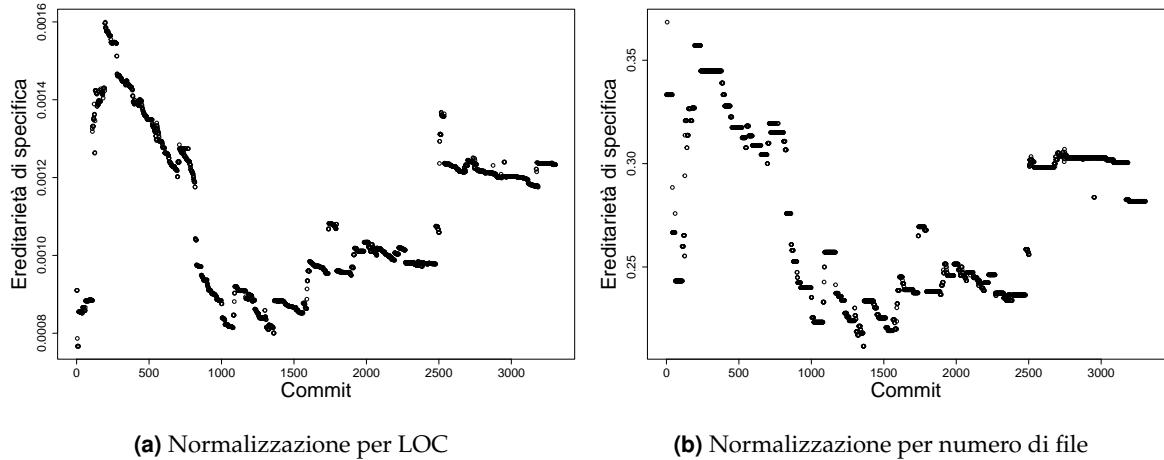


Figura 4.14: Compress - Ereditarietà di specifica

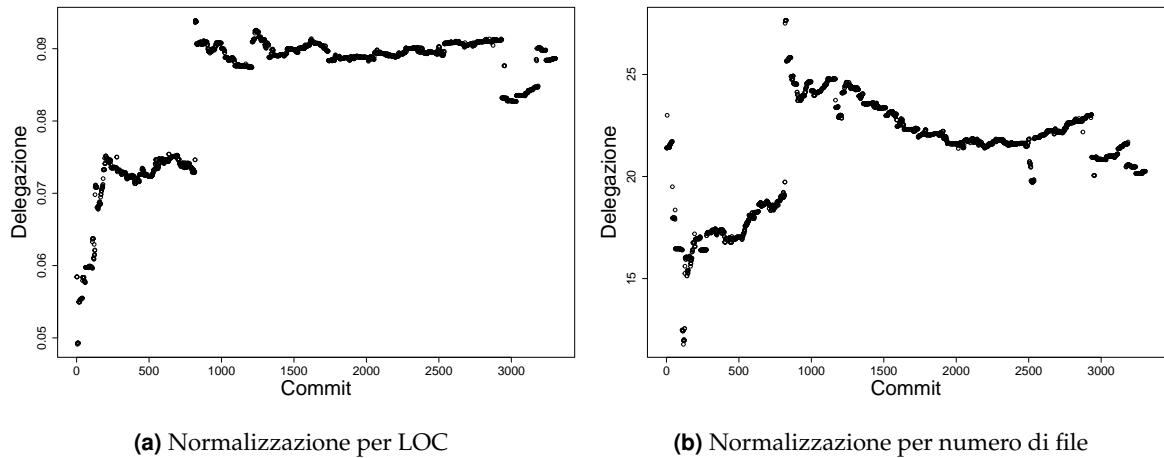


Figura 4.15: Compress - Delegazione

RQ1₃. Per la delegazione di Compress c'è poco da dire: dopo un'iniziale crescita graduale, è presente uno scalino, causato dallo spostamento di un file di test al di fuori del pacchetto di test, che ha comportato il suo ingresso nei file analizzati, per poi continuare con un andamento piuttosto stabile e lineare (al netto del fattore branching). Il finale è leggermente in discesa, ma l'analisi è resa complessa dal branching e dal fatto che i merge non sono presenti nel dataset.

4.1.6 Gson

Le pratiche di gestione della repository applicate per Gson hanno probabilmente contribuito fortemente alla poca leggibilità di alcuni dei grafici. Non è raro vedere aggiunte e rimozioni di grandi porzioni di codice e merge manuali.

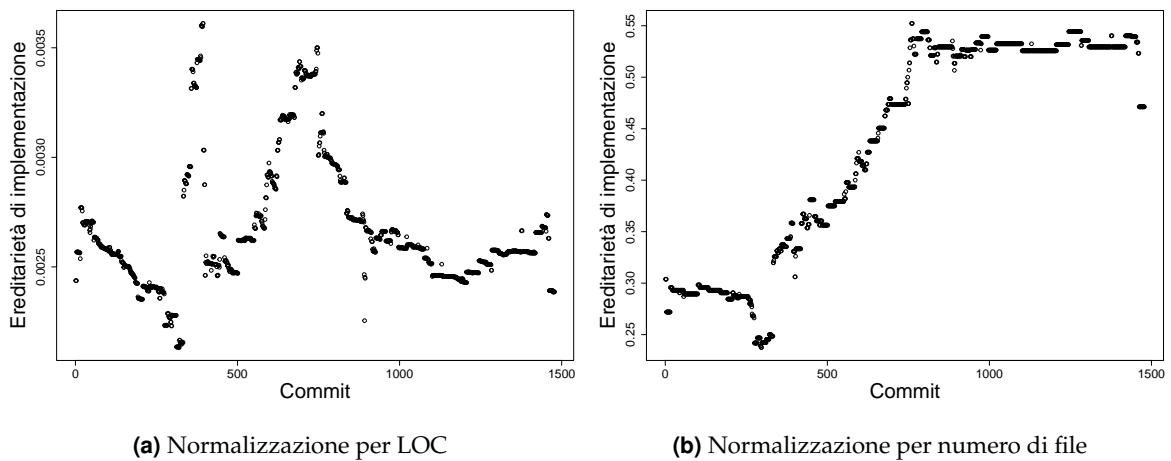


Figura 4.16: Gson - Ereditarietà di implementazione

RQ1₁. Certamente, per l'ereditarietà di implementazione, il grafico più semplice da osservare è quello relativo alla normalizzazione sul numero dei file in Figura 4.16(b). In questo caso c'è un avvio piuttosto stabile, seppur con una breve decrescita, per poi proseguire con un periodo di crescita graduale che, arrivata all'apice, si stabilizza.

Provando ad analizzare anche l'altro grafico, tuttavia, è possibile notare come inizialmente aumentino le linee di codice ma non l'uso dell'ereditarietà. Il commit numero 333 aggiunge 7 file che sfruttano l'eredità di implementazione e modifica altre 7 classi per far sì che queste estendano da una delle nuove classi create, facendo risalire la metrica. Gli incrementi saranno successivi, per un tempo di circa 70 commit. Tuttavia, la totalità di questi file sarà eliminata, riportando la situazione quasi a quella di partenza.

Segue un periodo di crescita graduale, che permette di arrivare all'apice, per poi iniziare la discesa con il commit 749, che elimina 12 file, di cui 8 utilizzavano l'ereditarietà. Il trend negativo non arriva fino alla fine, bensì un breve periodo positivo chiude la timeline. Questo periodo è dettato da una crescita graduale, senza immissioni eccessive.

RQ1₂. Per l'ereditarietà di specifica, la situazione è opposta rispetto all'ereditarietà di implementazione. Infatti, in questo caso il grafico più facile da analizzare è quello normalizzato

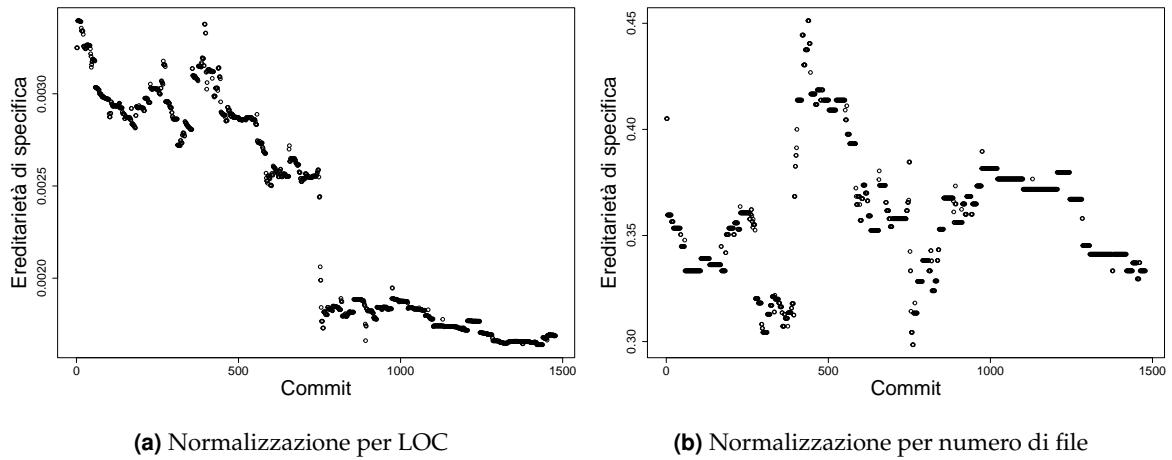


Figura 4.17: Gson - Ereditarietà di specifica

per LOC, in Figura 4.17(a). In questo vediamo una decrescita netta, che culmina in uno scostamento verso il basso netto dal quale la metrica non risale. Il salto è dovuto a un refactoring del codice relativo alle classi che implementavano l’interfaccia `ExclusionStrategy`, che vede l’eliminazione di queste per poi trasformarle in classi anonime, e prosegue per più di un commit.

Il salto è visibile anche nel grafico in Figura 4.17(b), a seguito del quale però, è presente una iniziale ripresa, dovuta a poche immissioni di nuovi file, ma con una buona media di utilizzo dell’ereditarietà di specifica. Il picco invece, è del tutto simile a quello del grafico in Figura 4.16(a), con una decrescita leggermente più graduale.

Restando sul confronto tra le due metriche, possiamo osservare come sebbene inizialmente l’ereditarietà di specifica sia più utilizzata (di circa il 30%) rispetto all’ereditarietà di implementazione, quest’ultima riprende velocemente terreno, e si arriva intorno ai 300 commit con un utilizzo praticamente pari delle due tecniche di riuso. In seguito, con la rimozione di un gran numero di file, entrambe le metriche perdono quota, ma a risentirne di più è l’ereditarietà di implementazione, che però si riprende più velocemente. Per il resto dell’evoluzione **l’ereditarietà di implementazione viene utilizzata in maniera più massiccia rispetto all’ereditarietà di specifica**.

RQ1₃. La delegazione presenta, invece, due grafici più simili tra loro. Gli scalini di crescita sono gli stessi che caratterizzano anche l’andamento delle due metriche di ereditarietà e le motivazioni sono le stesse. Tuttavia, a differenza dell’analisi fatta in precedenza, la delegazione tende a non perdere molto valore quando questo succede alle misure di ereditarietà.

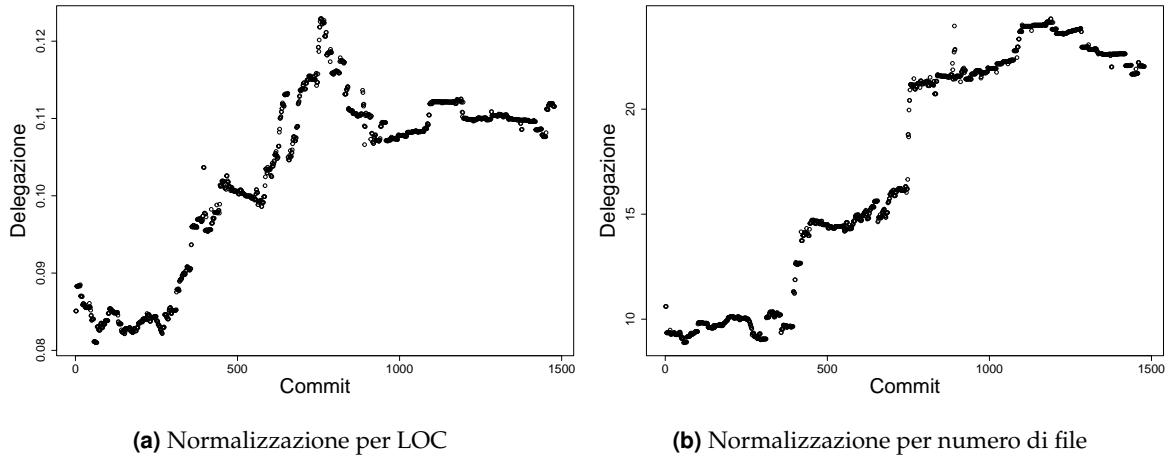


Figura 4.18: Gson - Delegazione

Ciò significa che in realtà **la delegazione è meglio distribuita tra le classi**. Il trend generale è di crescita, con situazioni di stallo intermedie (o breve decrescita, come nel caso della normalizzazione per LOC).

4.1.7 JacksonCore

JacksonCore, come tutti i progetti Jackson, utilizza un sistema di gestione della repository che prevede di utilizzare come main branch il branch dell'ultima versione disponibile. Dunque, se l'ultima versione disponibile è v , viene creato un branch apposito a partire dal branch della versione $v-1$. Il ramo relativo a $v-1$ potrebbe continuare nella sua evoluzione in maniera indipendente dal ramo di v , tuttavia frequentemente ci sono dei merge di $v-1$ in v , che portano i grafici ad essere, a volte, di difficile interpretazione. La maggior parte delle volte il fenomeno è rappresentato da dei punti sparsi che tendono a non seguire il flusso principale di punti.

RQ1₁. Gli andamenti dei due grafici sull'evoluzione dell'ereditarietà in Figura 4.19 di implementazioni sono piuttosto simili, seppure la normalizzazione per numero di file risulti più leggibile. Ignorando i punti provenienti da branching, si può osservare una breve flessione dell'utilizzo dell'ereditarietà dopo una crescita iniziale, che poi però tende a riprendersi. Dal grafico in Figura 4.19(b) vediamo come la crescita sia graduale, seppur con qualche punto di stallo. In questi punti non vengono aggiunti nuovi file alla repository, ma spesso viene aggiunto del codice (cosa che abbassa la curva del grafico in Figura 4.19(a)). Inoltre, i punti di stallo sono, come visibile, punti in cui si effettuano i merge dagli branch e si fa refactoring.

Sul finale la metrica scende leggermente, per poi iniziare un processo di risalita. Più in generale, il trend è positivo, dunque l'utilizzo l'ereditarietà di implementazione sembra crescere. Su un totale di poco più di 100 file, l'ereditarietà di implementazione è usata in oltre il 50% delle classi,

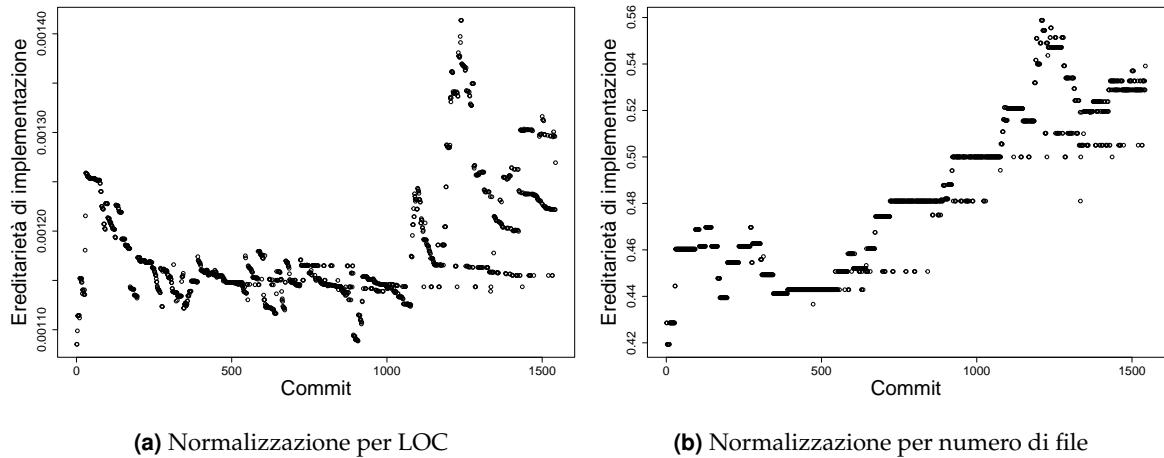
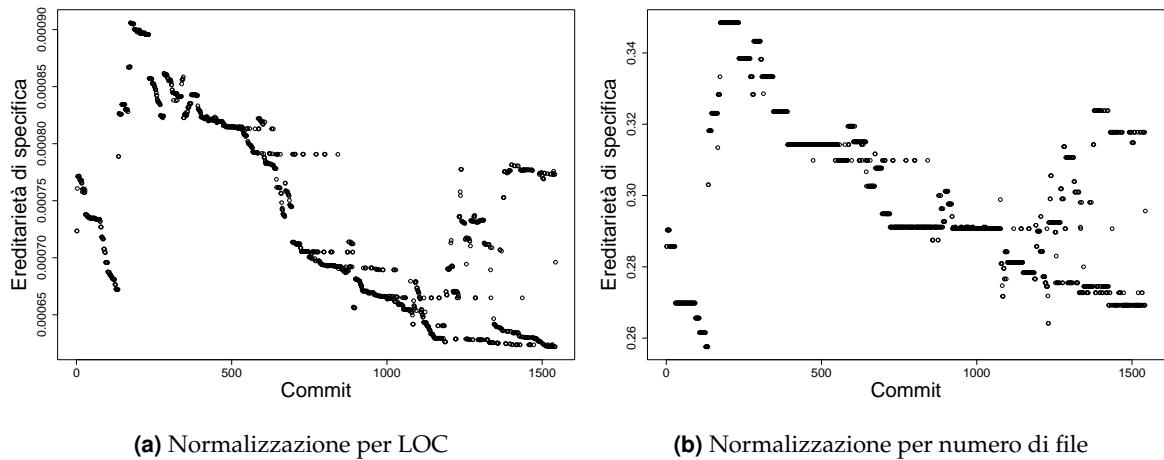
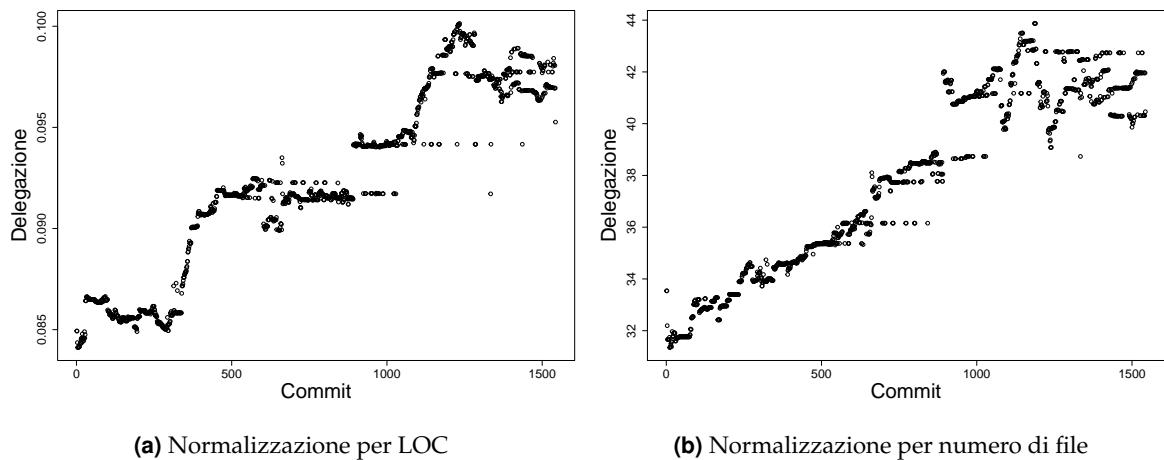


Figura 4.19: JacksonCore - Ereditarietà di implementazione

RQ1₂. Il trend dell'ereditarietà di specifica è opposto a quello dell'ereditarietà di implementazione. Viene raggiunto velocemente il picco massimo, per poi assistere a una discesa piuttosto graduale, con il finale non semplice da inquadrare causa branching. Il picco viene raggiunto insieme ad una serie di commit volti ad implementare l'interfaccia 'Serializable' per varie classi già esistenti, cosa che però non verrà più effettuata in seguito. In generale, **la presenza e il trend sono peggiori rispetto all'altra tipologia di ereditarietà.**

RQ1₃. L'andamento della delegazione è positivo in entrambi i grafici, ma quello con normalizzazione su numero di file che tende ad essere più lineare. Nel grafico in Figura 4.21(a), invece, sono presenti più punti di stallo, in cui il rapporto tra delegazione e linee di codice resta pressoché costante.

**Figura 4.20:** JacksonCore - Ereditarietà di specifica**Figura 4.21:** JacksonCore - Delegazione

4.1.8 JacksonDatabind

JacksonDatabind, come tutti i progetti Jackson, utilizza un sistema di gestione della repository non tradizionale, descritto nella Sezione 4.1.7. Anche in questo caso, i grafici sono influenzati da un talvolta visibile utilizzo del branching. In generale, si tratta di un progetto con un numero abbastanza alto di file e con oltre 5000 commit, il che lo rende il software più ampio analizzato.

RQ1₁. I due grafici in Figura 4.22 hanno andamento molto diverso. La ragione è la seguente: nonostante il progetto gestisca tra i 250 e i 450 file nel corso della sua storia - e dunque ci sia spesso incremento nel numero di file - questo numero è dominato dal numero di linee di

codice che vengono aggiunte alla repository.

Data la mole del progetto, concentrarsi sulle linee di codice potrebbe non essere pienamente indicativo dell'andamento. Detto ciò, vediamo che **l'ereditarietà di implementazione è usata in maniera massiccia**: di media, **oltre il 70% delle classi presenti nel sistema utilizza questo meccanismo**. Dopo un'iniziale decrescita, si può osservare come ci sia una crescita costante nell'arco di 1000 commit, che porta al picco del grafico. In seguito, nonostante il branching renda più complessa la lettura, si può notare una leggera discesa, uno stallo, una seconda discesa e un secondo stallo. Non sembra esserci un motivo vero e proprio dietro, semplicemente vengono impiegate meno classi che utilizzano la keyword 'extends' rispetto alla media precedente.

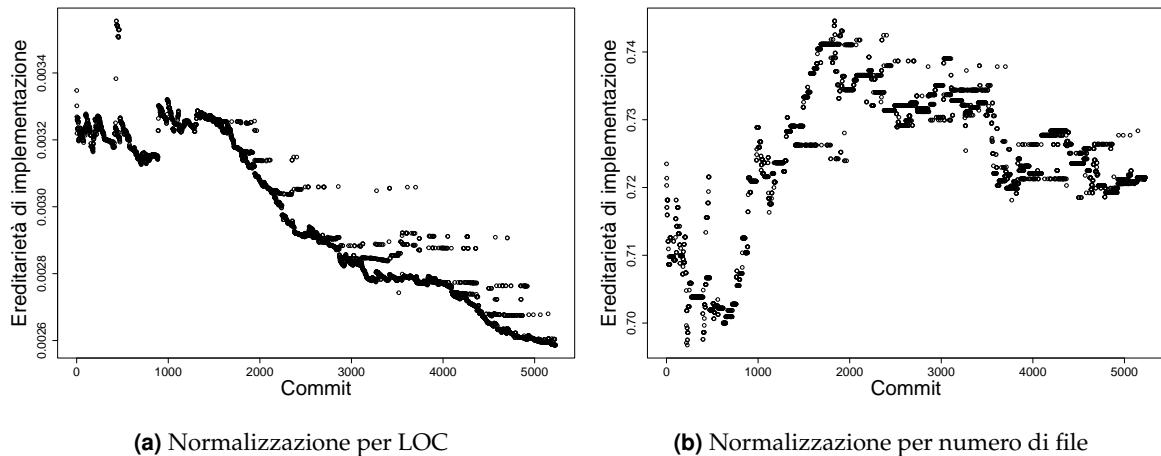
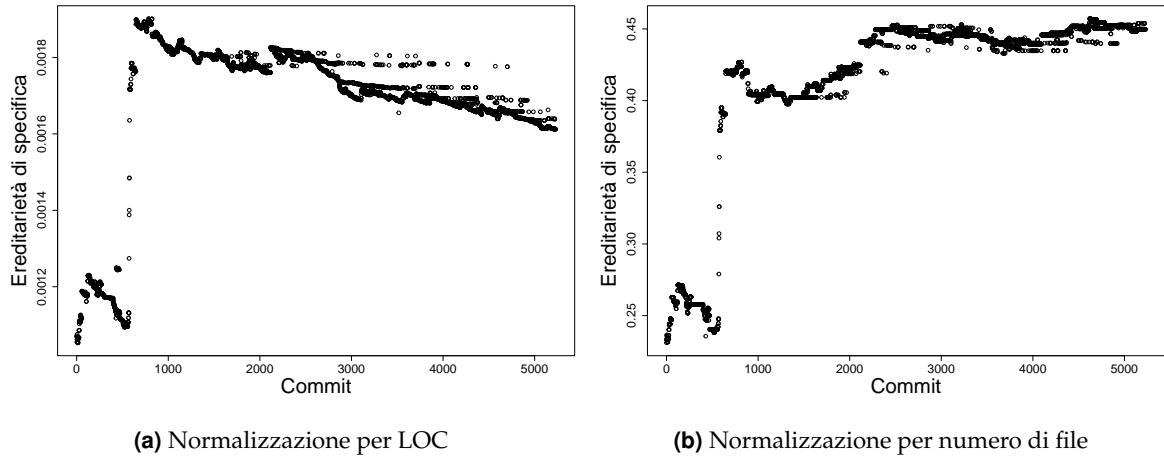
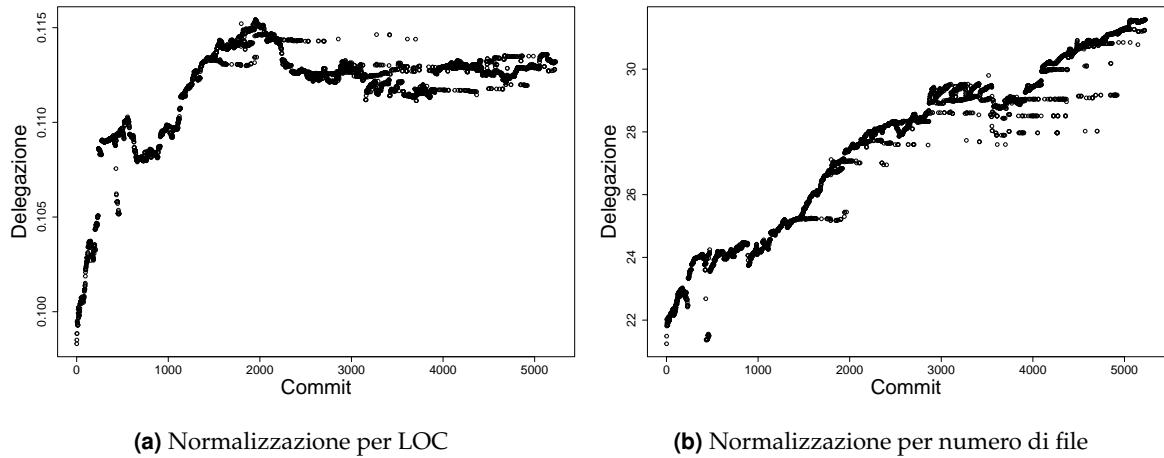


Figura 4.22: JacksonDatabind - Ereditarietà di implementazione

RQ1₂. L'ereditarietà di specifica, invece, viene utilizzata molto meno rispetto all'ereditarietà di implementazione. Vediamo che in Figura 4.23 c'è un punto in cui con pochi commit cresce molto la presenza di classi che utilizzano questo tipo di ereditarietà. Ciò avviene perché il proprietario della repository ha deciso di rendere quante più classi serializzabili per favorire il supporto alle applicazioni Android. C'è dunque un lavoro di refactoring che porta un gran numero di classi ad implementare l'intefaccia '*Serializable*'. In seguito, possiamo vedere come **il trend resti stabile** nel grafico che normalizza la metrica per il numero dei file, e tenda ad una leggera decrescita nel grafico che normalizza per linee di codice. La motivazione è la stessa evidenziata per l'ereditarietà di implementazione.

**Figura 4.23:** JacksonDatabind - Ereditarietà di specifica

RQ1₃. La delegazione, rispetto alle altre due metriche, segue un andamento più costante. Questa cresce in maniera piuttosto lineare laddove la si metta in rapporto con il numero di file, mentre, dopo una crescita graduale, resta stabile quando la si rapporta al numero di linee di codice, come visibile in Figura 4.24.

**Figura 4.24:** JacksonDatabind - Delegazione

4.1.9 JacksonXml

JacksonXml, come tutti i progetti Jackson, utilizza un sistema di gestione della repository non tradizionale, descritto nella Sezione 4.1.7. Anche in questo caso, i grafici sono influenzati da un talvolta visibile utilizzo del branching. In generale, si tratta di un progetto piuttosto ristretto, con poco più di 1000 commit e un massimo di circa 40 file e 10000 linee di codice.

RQ1₁. L'avvio della metrica di ereditarietà di implementazione è lo stesso per entrambi i grafici in Figura 4.25. In questa fase vengono aggiunti pochi file e poche linee di codice, ma laddove viene introdotto un nuovo file, questo fa uso di ereditarietà di implementazione. Vediamo come la presenza di queste classi è molto significativa, tanto da coprire oltre il 70% delle classi totali. A questo punto inizia una decrescita, nella quale si aggiungono nuovi file e nuovo codice con più costanza. Le classi che utilizzano il meccanismo di ereditarietà in esame non vengono rimosse, ma non aumentano con la stessa frequenza di prima. Successivamente, però la metrica riprende presenza. La differenza nell'andamento è presto spiegata: laddove nel grafico (b) vediamo uno stallo, nella repository non vengono inseriti nuovi file e la metrica (e il conseguente rapporto) restano costanti. In corrispondenza, però, in (a) sono presenti dei decrementi; ciò significa che nella repository è in atto refactoring/modifica di classi già esistenti, che questa sia una modifica manutentiva o evolutiva.

A seconda del grafico che si osserva si potrebbe avere una diversa visione dell'andamento dell'ereditarietà di implementazione. Il discorso verrà approfondito nelle conclusioni.

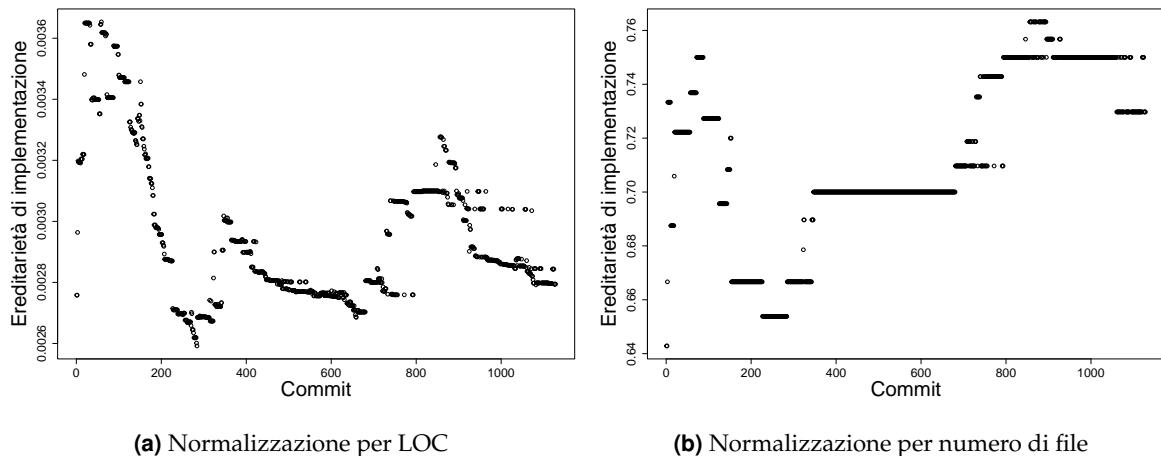


Figura 4.25: JacksonXml - Ereditarietà di implementazione

RQ1₂. Nel caso dell'ereditarietà di specifica i due grafici in Figura 4.26 si somigliano molto di più. L'incremento iniziale avviene in maniera non dissimile a quanto osservato per l'ereditarietà di implementazione. Tuttavia, in questo caso, il successivo trend calante è rappresentato in entrambi i grafici. È bene notare come la presenza dell'ereditarietà di specifica sia più bassa (del 50% circa) rispetto a quella dell'altra metrica di ereditarietà. Non ci sono particolari punti su cui indagare in questo caso.

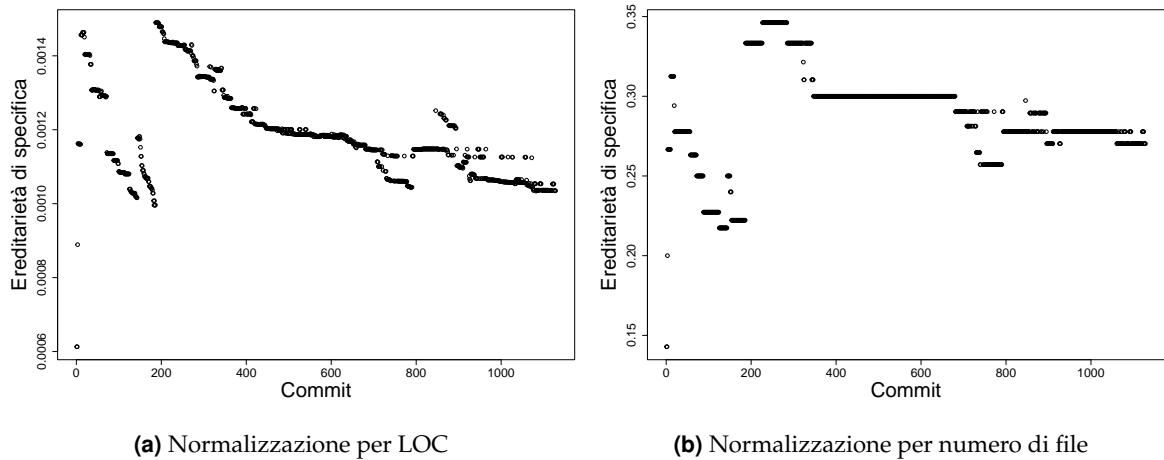


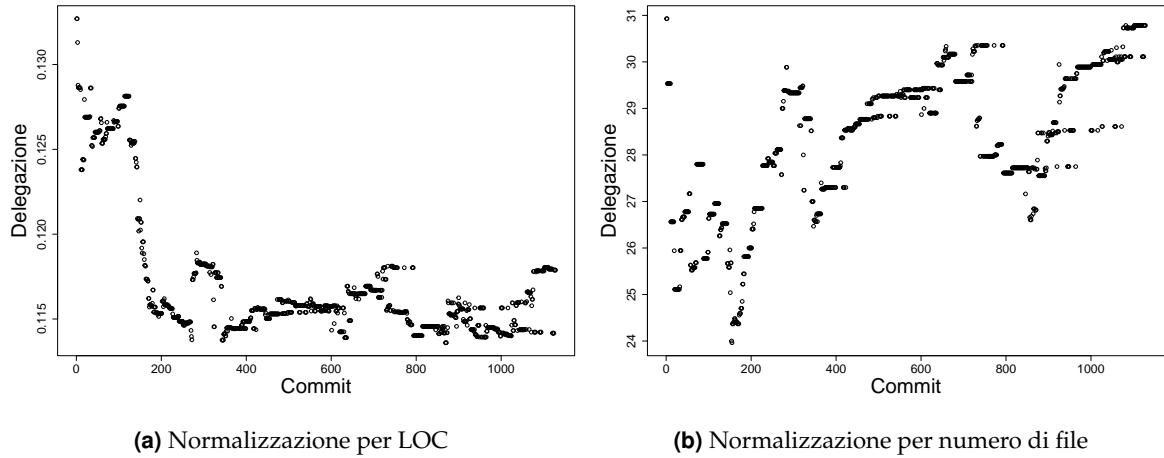
Figura 4.26: JacksonXml - Ereditarietà di specifica

RQ1₃. Per la delegazione il discorso è differente rispetto a quelli fatti per le due metriche di ereditarietà. Vediamo come i due grafici in Figura 4.27 siano sostanzialmente diversi tra loro. Come spiegato in precedenza, i file aumentano in maniera sporadica in questo progetto, mentre le linee di codice vengono sicuramente modificate con più frequenza. Proprio per come viene calcolata la delegation, questa dovrebbe sentire di più l'influenza delle linee di codice rispetto al numero dei file. Qui sembra essere così. La normalizzazione per numero di file, al netto di qualche drop (branching) segue un percorso di aumento. Tuttavia, quando andiamo ad analizzare il grafico normalizzato per LOC troviamo una discesa netta della presenza della delegazione, seguita poi da un periodo di piccole crescite e decrescite che, rimossa la componente branching, dovrebbe rappresentare un'andamento piuttosto costante.

4.1.10 JXPath

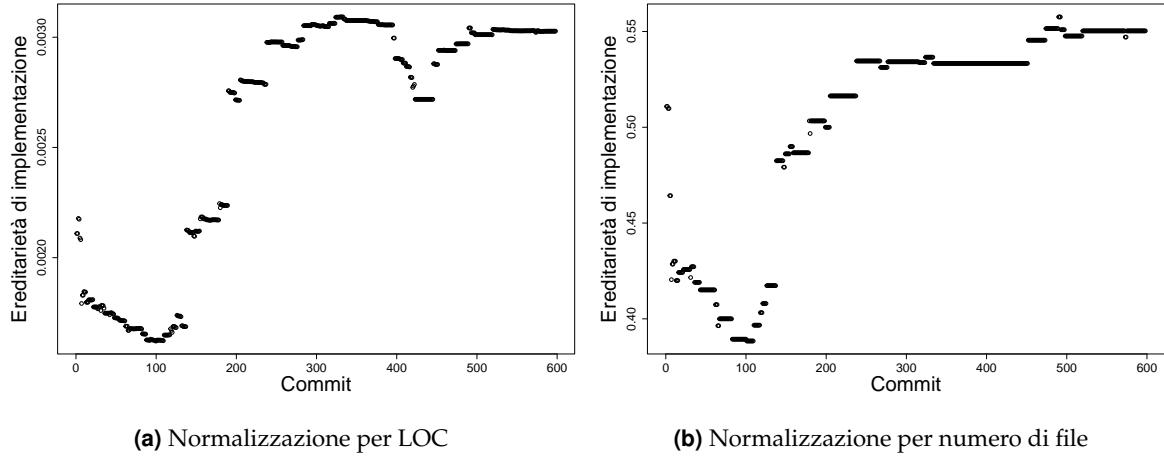
JXPath è un progetto che ha meno commit (circa 600) rispetto agli altri progetti esaminati. Tuttavia, il numero di file non è basso: si parte da 90 file per arrivare a 170, per un totale di circa 30000 linee di codice.

RQ1₁. I grafici dell'ereditarietà di implementazione in Figura 4.28 sono piuttosto simili. Dopo una breve discesa iniziale, entrambi trovano un salto verso l'alto, causato dal commit 138; questo commit aggiunge 16 classi per effettuare delle operazioni fondamentali (unione, sottrazione, divisione, ma anche confronti) e ognuna di queste classi estende '*CoreOperation*'. Mentre poi il grafico (b) continua con una crescita graduale, nel grafico (a) possiamo notare

**Figura 4.27:** JacksonXml - Delegazione

un ulteriore salto, questa volta dovuto all'eliminazione di 6500 linee di codice a causa di un refactoring a seguito di un cambio di licenza. Infine, i due grafici tendono ad assestarsi verso un trend piuttosto stabile.

In JxPath l'utilizzo dell'ereditarietà di implementazione si assesta al di sopra del 50%, ovvero più della metà delle classi sfrutta questo meccanismo di riuso.

**Figura 4.28:** JxPath - Ereditarietà di implementazione

RQ1₂. I due grafici in Figura 4.30, relativi all'ereditarietà di specifica, sono diversi l'uno dall'altro. Fino al commit 200 l'andamento è simile, seppur modulato diversamente. Nel caso della normalizzazione per numero di file, il commit 138 che fa incrementare di molto l'ereditarietà di implementazione, qui fa crollare quella di specifica: nessuno dei 16 file

aggiunti utilizza il meccanismo in questione. Il commit che elimina 6500 linee di codice è quello che fa riprendere quota al grafico (a). I due grafici da questo punto in poi seguono un trend di decrescita intervallato da situazioni di stallo.

Più in generale, l'**ereditarietà di specifica viene utilizzata** da circa il 30% delle classi, dunque **in maniera significativamente minore rispetto all'ereditarietà di implementazione**.

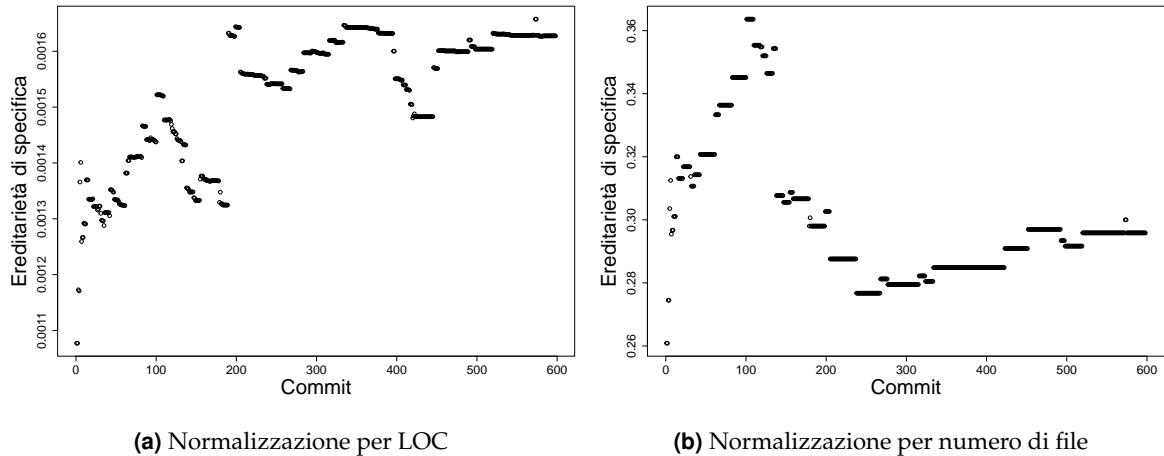


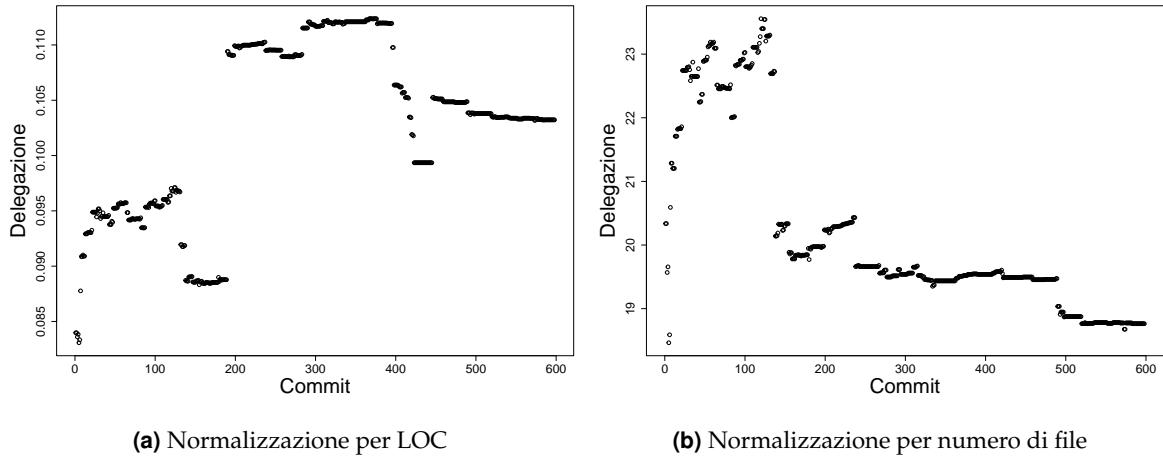
Figura 4.29: JxPath - Ereditarietà di specifica

RQ1₃. La particolarità dei grafici della delegazione, in Figura 4.30, è che somigliano ai grafici dell'eredità di specifica, in Figura 4.29, seppur mantenendo alcune differenze. Si tratta di una situazione unica, che non si verifica per gli altri progetti. I punti salienti e l'andamento sono dunque già stati analizzati precedentemente. Nonostante la somiglianza, non si evince una relazione diretta con l'ereditarietà di specifica, almeno non dai commit.

4.1.11 Time

Time è un sistema di buone dimensioni, che subisce molte modifiche e sostanziose aggiunte/rimozioni di file. Ha oltre 2000 commit e contiene dai 60 ai 300 file, pur terminando con meno di questa cifra.

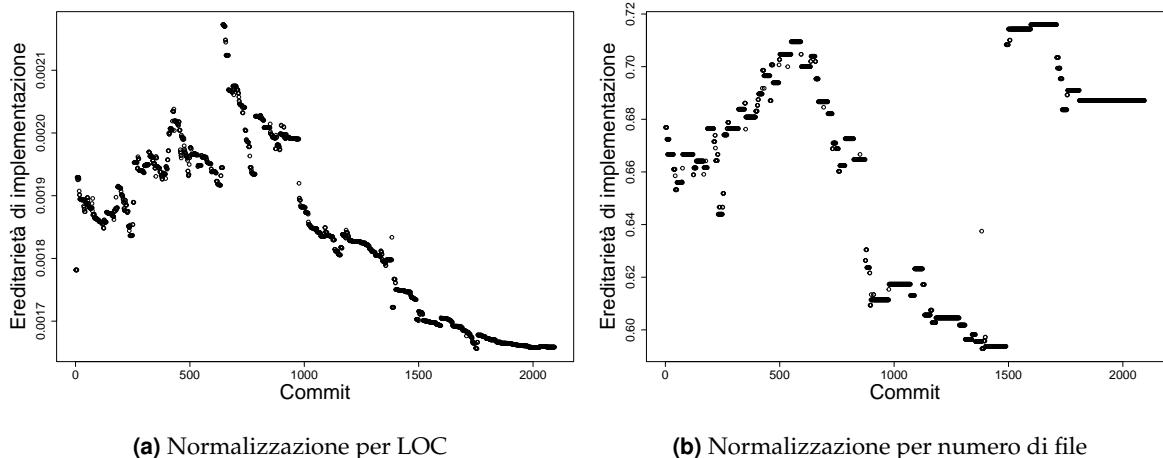
RQ1₁. I due grafici in Figura 4.31, che rappresentano l'andamento dell'ereditarietà di implementazione, condividono una prima fase di crescita, che culmina in un picco intorno ai 650 commit, dove un cambio di licenze porta a un refactoring che riduce le linee di codice senza toccare la metrica di ereditarietà. Quest'ultima subisce un calo graduale in termini di presenza percentuale nel progetto, con un ulteriore ribasso al commit 874, dove vengono

**Figura 4.30:** JxPath - Delegazione

aggiunti molti file ma nessuno utilizza l'ereditarietà.

Mentre per il grafico (a) la discesa prosegue senza particolarità, in (b) c'è un salto molto importante; vengono eliminati 120 file e 30000 linee di codice e resi (si tratta della rimozione della directory dedicata ai contributi della comunità). Questo causa, naturalmente, anche un calo in termini assoluti della metrica, che però in rapporto al numero dei file ne esce rafforzata.

Anche qui l'eredità di implementazione è utilizzata in buona parte dei file, qui oltre il 60%, con spesso valori al di sopra del 65%

**Figura 4.31:** Time - Ereditarietà di implementazione

RQ1₂. I grafici in Figura 4.32 condividono la crescita iniziale. A differenza dell’andamento dell’eredità di implementazione, qui non c’è una decrescita netta, anzi i valori si discostano poco rappresentando una quasi stabilità. Questa volta il commit che elimina 120 file ha impatto sul grafico (a) più che sul grafico (b). Mentre il secondo resta in stallo, il primo ha una decrescita netta.

In generale, la presenza dell’eredità di specifica si attesta tra il 35% e il 40%, venendo dunque superata ampiamente dall’eredità di implementazione.

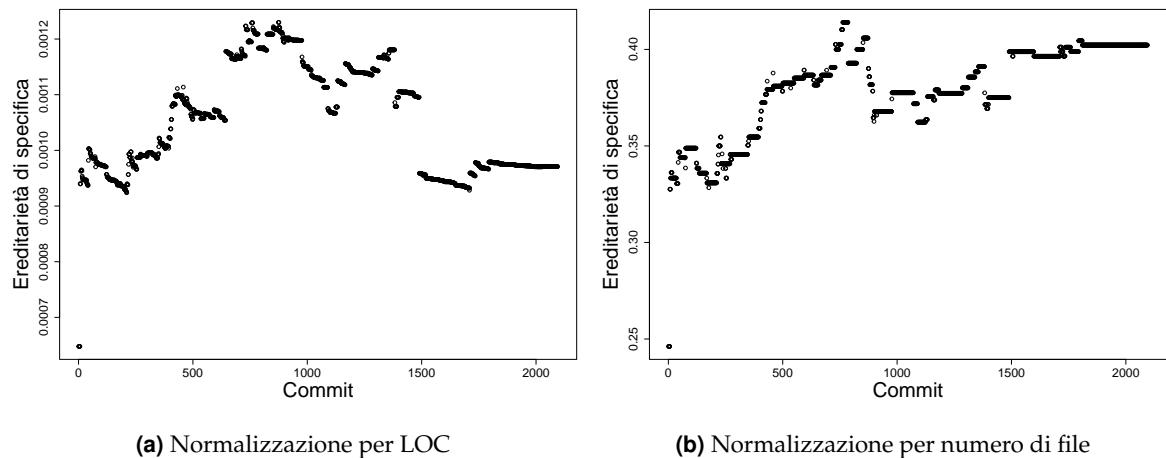
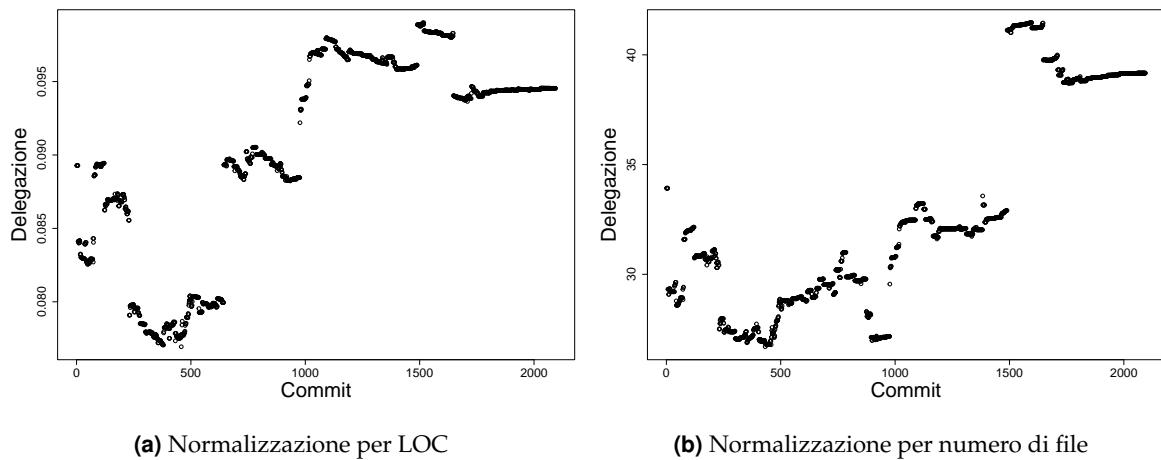


Figura 4.32: Time - Ereditarietà di specifica

RQ1₃. Il primo calo della delegazione, visibile in Figura 4.33 è dovuto alla rimozione, intorno al commit 230, di 4000 linee di codice per mezzo dell’eliminazione di 9 file, che erano una parte di software non più supportata. Le altre variazioni dei grafici sono giustificate allo stesso modo di quanto accade per l’ereditarietà di implementazione e di specifica.

**Figura 4.33:** Time - Delegazione

4.1.12 Risposta alla domanda di ricerca RQ1

Prima di concludere, è necessario fare una considerazione sulle due normalizzazioni utilizzate per lo studio. Spesso, i grafici hanno mostrato andamento molto simile, tuttavia non sempre sono risultati entrambi leggibili. Come già detto in fase di modellazione, la normalizzazione più adatta alle metriche analizzate, almeno per le due di ereditarietà, è quella fatta per numero di file. Con questa, infatti, possiamo osservare la presenza delle classi che utilizzano l'ereditarietà sul numero totale di classi, e ciò permette di ottenere una prospettiva anche in termini percentuali della diffusione del fenomeno. L'altra tipologia di normalizzazione, invece, sulle linee di codice, è stata studiata per continuità con analisi empiriche precedenti, dove, però, la misurazione delle metriche avveniva in maniera diversa. Nella seguenti conclusioni, dunque, laddove ci siano differenze sostanziali si tenderà a valutare i grafici sull'andamento delle metriche normalizzate per numero di file.

Evoluzione dell'ereditarietà di implementazione L'ereditarietà di implementazione, che mediamente viene utilizzata di più rispetto all'ereditarietà di specifica, non segue lo stesso trend tra i vari progetti. In quasi tutti i grafici, però, si osserva come ci sia una fase iniziale in cui l'utilizzo del meccanismo cresce, per poi raggiungere un picco. A quel punto, la maggior parte dei progetti riporta un andamento piuttosto stabile, con crescita e decrescita non significative. Altri, invece, mostrano dei crolli. Più in generale, ci sono molti alti e bassi nei grafici, frutto spesso di interventi specifici sui progetti, per cui risulta complesso trovare una linea comune per tutto il corso dell'evoluzione.

Se tuttavia consideriamo i primi commit e gli ultimi commit, ad eccezione di un solo progetto,

tutti mostrano un incremento - in proporzione al numero dei file - dell'utilizzo dell'ereditarietà di implementazione.

Evoluzione dell'ereditarietà di specifica L'eredità di specifica ha ancora più variabilità dell'ereditarietà di implementazione, con grafici molto diversi tra loro. Quello che accomuna la maggior parte di essi, però, è una tendenza alla stabilità finale, sia che si provenga da un incremento, sia che si provenga da un decremento. Alcuni grafici evolutivi sono molto simili alle controparti relative all'ereditarietà di implementazione, mentre altri sono completamente diversi.

Anche in questo caso, considerando solo i primi commit e gli ultimi commit c'è effettivamente un incremento (ad eccezione di un progetto), anche se questo è mediamente meno significativo rispetto all'ereditarietà di implementazione.

Evoluzione della delegazione La delegazione, invece, ha un trend che, in 9 progetti su 11, prevede una generale crescita della metrica, in alcuni casi anche con abbassamenti temporanei o stalli seguiti da riprese. Sono solo due i progetti per i quali, invece, c'è una decrescita, che in un caso è molto netta rispetto ai valori iniziali, e nell'altro caso è più graduale.

4.2 Risultati RQ2

In RQ2 viene utilizzato il Multinomial Log-Linear model per osservare l'impatto delle variabili indipendenti sulla variazione del numero dei difetti nel codice. Per ogni variabile indipendente si ottengono due coefficienti: uno indica l'impatto che un incremento unitario di quella variabile ha sulla probabilità che aumenti il numero di bug, l'altro invece indica l'impatto che un incremento unitario di quella variabile ha sulla probabilità che diminuisca il numero di bug. La presenza di asterischi (da 1 a 3, in ordine crescente di importanza) indica che la variabile è statisticamente significativa.

Tabella 4.1: Variabili rimosse dai progetti a causa della multicollinearità

Progetto	Variabili scartate
Codec	RFC, NOC
Cli	DIT, NOC, Ereditarietà di implementazione
CommonsCollections	WMC
CommonsCsv	RFC
Compress	RFC
Gson	RFC
JacksonCore	WMC, RFC, DIT, Ereditarietà di implementazione
JacksonDatabind	RFC
JacksonXml	WMC, RFC, DIT
JxPath	DIT
Time	WMC, RFC, DIT

La Tabella 4.1 riporta le variabili scartate dai vari progetti a causa della multicollinearità. Per semplificare la lettura dei risultati, di seguito vengono riportate solo le tabelle ottenute dai primi due progetti tramite l'applicazione del modello multinomiale. Le restanti tabelle sono presenti in Appendice A.

4.2.1 Codec

Per il sistema Codec sono state scartate, causa correlazione, le variabili RFC e NOC. Come si può evincere dalla Tabella 4.2, le metriche di ereditarietà definite in questo studio non impattano in maniera statisticamente significativa la presenza dei bug nel codice. È bene notare come, però, le metriche Object Oriented di CK abbiano una certa rilevanza. Ciò che sorprende è vedere come il Coupling Between Objects incide sulla variabile dipendente: nonostante ci si potesse aspettare che aumentando l'accoppiamento fosse più probabile introdurre un bug, avviene l'esatto contrario.

La metrica CK più legata allo studio dell'ereditarietà, ovvero DIT, è statisticamente significativa. Al crescere della profondità dell'albero di ereditarietà si riduce la probabilità di incrementare il numero dei bug e si aumenta, in maniera preponderante, la probabilità di far decrescere il numero di bug.

4.2.2 Cli

Per Cli sono state scartate la DIT, la NOC e una delle metriche definite, ovvero l'ereditarietà di implementazione, correlata all'eredità di specifica. Come mostrato nella Tabella 4.3, tra le metriche CK solo la Lack of Cohesion Of Methods è statisticamente significativa; a sorpresa, al crescere di questa metrica diminuisce la probabilità di incrementare il numero dei bug nel sistema. La metrica relativa all'ereditarietà di specifica è significativa, seppur non quanto LCOM. Al crescere dell'ereditarietà di specifica aumenta la probabilità di introdurre un bug. Questo è un primo risultato che mette in correlazione riuso e presenza di difetti.

Tabella 4.2: Codec - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	−10.098 (7.495)	2.280 (10.981)
diffLCOM	0.092 (0.140)	0.054 (0.261)
diffDIT	11.927*** (0.269)	−0.183*** (0.033)
diffCBO	−5.434 (5.898)	−9.729*** (0.243)
diffLOC	0.005 (0.302)	0.075 (0.346)
diffdelegations	0.058 (0.049)	−0.060 (0.077)
diffintInh	−1.791 (1.685)	−1.510 (3.395)
diffimpInh	−0.060 (0.940)	0.046 (2.134)
churns	−0.002 (0.003)	−0.002 (0.004)
Constant	−4.762*** (0.248)	−4.717*** (0.246)
Akaike Inf. Crit.	449.237	449.237

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella 4.3: Cli - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	-0.691 (3.434)	2.416 (3.602)
diffLCOM	0.166 (0.256)	-0.744*** (0.244)
diffCBO	-0.645 (3.617)	-5.947 (3.821)
diffRFC	-0.014 (1.027)	0.271 (1.118)
diffLOC	0.002 (0.139)	0.056 (0.200)
diffdelegations	0.017 (0.022)	0.001 (0.025)
diffintInh	0.070 (0.618)	1.382** (0.542)
churns	-0.002 (0.002)	-0.005 (0.004)
Constant	-3.472*** (0.187)	-3.448*** (0.187)
Akaike Inf. Crit.	604.909	604.909

Note: *p<0.1; **p<0.05; ***p<0.01

4.2.3 CommonsCollections

Il sistema CommonsCollections ha portato a scartare solo la WMC. Nessuna delle metriche (vedi Tabella A.1), se non la CBO, è rilevante per il problema. Di nuovo, sorprendentemente, un aumento dell'accoppiamento porta alla riduzione della probabilità di introdurre bug.

4.2.4 CommonsCsv

Per CommonsCsv (vedi Tabella A.2) viene scartata la sola RFC. Nonostante le metriche di riuso non abbiano impatto sulla presenza di bug nel codice, le due metriche di CK più legate all'uso dell'ereditarietà, ovvero DIT e NOC lo sono e si comportano allo stesso modo. Un incremento di una di queste due metriche causa una diminuzione della probabilità che venga risolto un bug, e inoltre fa aumentare la probabilità che ne venga introdotto uno.

4.2.5 Compress

Compress (vedi Tabella A.3) ha richiesto la rimozione della sola RFC dal modello. Anche qui, le metriche definite dallo studio non hanno rilevanza, ma è comunque interessante notare come siano di nuovo significative DIT e NOC, ma con un impatto inverso rispetto a quanto visto per CommonsCsv. Qui infatti un loro aumento causa la diminuzione della probabilità di introdurre un bug e aumentano le probabilità di rimuoverlo. Su Compress il CBO e anche il WMC si comportano come ci si aspetterebbe, almeno in parte. Se è vero che non sembrano essere statisticamente significativi per l'introduzione di bug, sono sicuramente significativi per la riduzione del numero dei bug: un loro aumento fa decrescere la probabilità che questo accada.

Le Lines Of Code hanno invece un piccolo impatto positivo sul decrease del numero di bug, al contrario di WMC e CBO.

4.2.6 Gson

Gson (vedi Tabella A.4) ha richiesto la rimozione della sola RFC. Di nuovo, le nuove metriche non sono significative, mentre DIT e NOC hanno una buona significatività. Entrambe - anche se la NOC ha maggiore impatto - con un loro aumento causano un incremento della probabilità di introdurre bug.

4.2.7 JacksonCore

JacksonCore (vedi Tabella A.5) è stato il progetto che ha richiesto più rimozioni: DIT, RFC, WMC e anche l'ereditarietà di implementazione, correlata a quella di specifica. L'unica metrica statisticamente significativa è la NOC, che, con un suo incremento, diminuisce sia la probabilità di rimuovere un bug sia (in maniera preponderante) quella di introdurlo.

4.2.8 JacksonDatabind

JacksonDatabind (vedi Tabella A.6) ha richiesto la rimozione della sola RFC. Ci sono diverse metriche statisticamente significative. La DIT e la NOC hanno lo stesso comportamento: al crescere di una delle due diminuisce sia la probabilità di rimuovere un bug sia (in maniera preponderante) quella di introdurlo. La mancanza di coesione invece (LCOM) è significativa "in positivo", ovvero si comporta in maniera esattamente inversa rispetto a DIT e NOC. Come ci si può aspettare, la crescita della complessità ciclomatica dei metodi, misurata dalla WMC, è uno dei fattori che può contribuire all'introduzione di un difetto.

La cosa che interessa più lo studio, però, è l'impatto delle metriche del riuso, dato che questa volta sia l'ereditarietà di implementazione che la delegazione sono statisticamente significative. Un aumento della prima incrementa la probabilità di introdurre un bug, mentre un incremento della seconda diminuisce questa probabilità.

Significativi, seppur non quanto gli altri, sono anche i churns, il cui aumento è negativamente correlato all'incremento del numero di bug nel sistema.

4.2.9 JacksonXml

Per JacksonXml (vedi Tabella A.7) sono stati rimossi WMC, RFC e DIT a causa della correlazione. Le metriche statisticamente significative solo la NOC, un cui aumento causa la decrescita della probabilità sia di rimuovere che di introdurre un bug, e i churns, un cui aumento fa diminuire la probabilità che si incrementi il numero dei bug.

4.2.10 JxPath

Per JxPath (vedi Tabella A.8) è stata rimossa la sola metrica DIT a causa della collinearità. Molte delle metriche sono significative. Un aumento della WMC incrementa la probabilità di introdurre un bug e al contempo decrementa le probabilità di effettuare un fix. La NOC fa esattamente il contrario. La metrica dell'accoppiamento, la CBO, ha stranamente una correlazione negativa con la probabilità di introdurre bug, stessa cosa vale per la RFC,

churns e, soprattutto, per l'ereditarietà di implementazione. La delegazione, invece, ha rapporto positivo con la probabilità di introdurre difetti. La metrica delle LOC è correlata positivamente con la probabilità di aggiungere bug e negativamente (anche se leggermente) con la probabilità di ridurli. Infine, anche l'ereditarietà di specifica ha una lieve correlazione negativa con la probabilità di risolvere un difetto.

4.2.11 Time

Sul sistema Time (vedi Tabella A.9) vengono scartati WMC, RFC e DIT per la correlazione tra variabili indipendenti.

Un aumento della NOC comporta un decremento delle probabilità sia di introduzione di un bug sia di rimozione di un bug, mentre all'aumentare della CBO, inaspettatamente, incrementa la probabilità di eliminare un difetto e decresce la probabilità di introdurlo.

Infine, l'ereditarietà di specifica ha una correlazione negativa con l'incremento del numero dei bug.

4.2.12 Risposta alla domanda di ricerca RQ2

La RQ2 si pone come obiettivo quello di stabilire se esiste una relazione tra i meccanismi di riuso e la presenza di difetti nel codice. Dopo aver applicato il modello multinomiale per capire quali variabili fossero significative per l'increase / decrease dei bug, risulta che solo in tre casi alcune delle metriche di ereditarietà e delegazione sono significative. Inoltre, il loro impatto non è lo stesso tra i diversi software. Ciò significa che, ad esempio, l'eredità di specifica ha un impatto positivo sulla probabilità che venga introdotto un bug in un progetto, mentre in un altro ha impatto negativo sulla stessa probabilità.

Si può dunque concludere che, con il dataset di bug a disposizione, è raro che i meccanismi di riuso impattino sulla presenza di difetti nel codice. È possibile anche aggiungere che anche laddove ci sia un impatto, non è possibile generalizzare sulla tipologia dell'influenza che questi meccanismi hanno. Ciò vuol dire che, dipendentemente dal progetto, un aumento del riuso può portare sia ad una maggiore probabilità di incrementare il numero dei difetti sia ad una minore probabilità di farlo.

4.3 Risultati RQ3

Per RQ3 è stato utilizzato il Generalized Linear Model. Essendo la variabile dipendente, ovvero i code churns, numerica, otteniamo un coefficiente per ogni variabile indipendente che rappresenta l'impatto che ha l'incremento di una unità di quella variabile sul numero di churns. Laddove siano presenti degli asterischi (da 1 a 3, in ordine crescente di importanza) vuol dire che la variabile è statisticamente significativa per il modello.

Le variabili scartate sono le stesse già scartate da RQ2, e sono state riportate precedentemente (vedi Tabella 4.1).

Per semplificare la lettura dei risultati, di seguito vengono riportate solo le tabelle ottenute dai primi due progetti tramite l'applicazione del modello lineare generalizzato. Le restanti tabelle sono presenti in Appendice B.

4.3.1 Codec

Per il sistema Codec (vedi Tabella 4.4), le uniche metriche che impattano l'effort sono il CBO e il DIT. Dunque, non essendo possibile differenziare tra incremento e decremento del bug, non è possibile predire il costo dell'intervento manutentivo.

4.3.2 Cli

Cli (vedi Tabella 4.5) mostra dei risultati interessanti, in quanto il modello considera la delegazione e l'ereditarietà di interfaccia (oltre alla RFC e in piccola parte alla LOC) come significative per la predizione dell'effort. Di nuovo però, lo stato della variabile relativa ai bug non incide sull'effort, non rendendo possibile la predizione del costo di un intervento correttivo.

Tabella 4.4: Codec - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffWMC	163.951 (105.295)
diffLCOM	1.383 (2.713)
diffDIT	3,341.228*** (903.732)
diffCBO	1,021.357*** (150.161)
diffLOC	1.293 (2.158)
delegations	0.017 (0.045)
intInh	-1.217 (3.145)
impInh	-0.131 (1.747)
Bug2Decrease	1.433 (37.641)
Bug2Increase	-20.191 (37.620)
Constant	52.948*** (15.918)
Observations	2,134
Log Likelihood	-13,837.350
Akaike Inf. Crit.	27,696.690

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella 4.5: Cli - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffWMC	26.263 (227.457)
diffLCOM	-12.154 (16.169)
diffCBO	-108.063 (134.420)
diffRFC	192.611*** (57.094)
diffLOC	-9.992* (5.254)
delegations	-0.697*** (0.229)
intInh	39.595*** (11.915)
Bug2Decrease	-74.159 (102.978)
Bug2Increase	-70.799 (101.593)
Constant	126.288** (51.413)
Observations	1,099
Log Likelihood	-8,532.446
Akaike Inf. Crit.	17,084.890

Note: *p<0.1; **p<0.05; ***p<0.01

4.3.3 CommonsCollections

Il modello applicato al sistema CommonsCollections (vedi Tabella B.1) non riporta una corrispondenza statisticamente significativa né tra effort e metriche di riuso, né tra effort e variazione del bug.

4.3.4 CommonsCsv

Per CommonsCsv (vedi Tabella B.2) addirittura solo una metrica CK viene considerata come significativa per la predizione dei churns. Si tratta della LCOM e quindi della mancanza di coesione tra i metodi. Più questa aumenta, più tende ad aumentare il costo dell'intervento.

4.3.5 Compress

Compress (vedi Tabella B.3) ha una relazione statisticamente significativa tra l'effort e l'ereditarietà di implementazione (più questa sale, più i chrunsi aumentano) ma anche con la delegazione (cresce la delegazione, diminuiscono i churns). Inoltre, la variabile dipendente è in relazioni statisticamente significative con quasi tutte le variabili della suite CK, ad eccezione delle LOC (e di RFC, che è stata scartata).

4.3.6 Gson

Il modello applicato su Gson (vedi Tabella B.4) rileva una correlazione significativa tra l'effort e la delegazione, oltre che con NOC, LOC, CBO (verso concorde, quando aumentano fanno crescere l'effort) e DIT (verso discorde).

4.3.7 JacksonCore

JacksonCore (vedi Tabella B.5) è l'unico progetto per cui il modello considera statisticamente significativa la relazione tra effort e Decrease del bug, seppure questo rapporto non sia significativo quanto quello che i churns hanno con il NOC, l'accoppiamento (CBO) e la LOC.

4.3.8 JacksonDatabind

Su JacksonDatabind (vedi Tabella B.6) il modello individua relazioni statisticamente significative tra i churns e quasi tutte le variabili in gioco, eccetto per l'Increase/Decrease dei bug e l'ereditarietà di implementazione

4.3.9 JacksonXml

Per JacksonXml (vedi Tabella B.7) l'effort è collegato anche all'ereditarietà di specifica, oltre che a LCOM, CBO, LOC e delegazione.

4.3.10 JxPath

Su JxPath (vedi Tabella B.8) il modello segnala la delegazione come uno dei fattori statisticamente significativi che, quando aumenta, fa scendere il numero dei churns. Inoltre, c'è relazione con le metriche CK.

4.3.11 Time

Infine, su Time (vedi Tabella B.9) il modello individua le relazioni con le due ereditarietà e la delegazione: qui è interessante notare come la delegazione e l'ereditarietà di specifica abbiano impatto "negativo" sui code churns (quando le due metriche aumentano, i code churns diminuiscono), mentre l'ereditarietà di implementazione faccia esattamente l'opposto. È da notare come però la variazione per unità di incremento offerta dalla delegazione è piuttosto bassa. Tuttavia, c'è da considerare che il range di valori assunto dalla delegazione è molto più ampio rispetto a quello che le due metriche di ereditarietà assume.

4.3.12 Risposta alla domanda di ricerca RQ3

Per RQ3 ci si è chiesti se fosse possibile predire il costo in termini di effort per la correzione dei difetti. Dopo aver applicato il modello lineare generalizzato a tutti i progetti per capire quali fossero le variabili significative per la predizione dell'effort, scopriamo che l'incremento o il decremento del numero di difetti non ha peso sul numero di righe di codice modificate. Si può notare, inoltre, come l'errore standard sia sempre piuttosto elevato per le variabili che indicano la variazione nel numero di bug. Questo significa che il fatto che un commit risolva un difetto, naturalmente contestualizzato nel dataset di Defects4J, non è una discriminante sufficientemente significativa da far variare l'effort per quel commit.

Dunque, per RQ3 si può concludere che, dato il dataset di bug messo a disposizione da Defects4J e andando a considerare tutti i commit delle repository, non è possibile stimare l'effort necessario per la correzione dei difetti.

CAPITOLO 5

Conclusioni

In questa tesi è stata valutata l'evoluzione dei meccanismi di riuso e l'impatto che questi hanno sulla presenza di difetti nel codice. I risultati, data la granularità al *commit-level*, mostrano che non c'è una linea comune seguita da tutti i progetti software, e che ogni software ha una storia a sé, con il fattore di gestione della repository che assume un buon livello di importanza. Differenziando i risultati ottenuti sui tre meccanismi analizzati, si nota come l'ereditarietà di implementazione sia in linea di massima più utilizzata e abbia un trend di crescita più positivo rispetto all'ereditarietà di specifica, mentre la delegazione viaggia su un binario separato rispetto alle altre due metriche. L'adozione del riuso non sembra avere un impatto significativo sulla presenza di difetti sul codice, con solo tre progetti sugli undici analizzati che ricevono un contributo statisticamente significativo dalle metriche poste in esame. Tuttavia, anche sulla natura del contributo non è possibile tracciare una linea comune, in quanto l'impatto di ciascun meccanismo sul numero di difetti varia a seconda del software preso in esame.

Inoltre, non è possibile, per i progetti e il dataset di bug utilizzati, predire l'effort necessario per la correzione dei difetti, in quanto la diminuzione del numero di bug nei progetti non sembra essere una discriminante sufficiente a far variare l'effort rispetto ad altri interventi.

Lo studio ha messo in luce aspetti interessanti sui quali sarà possibile ricercare in futuro. In primis, il mining delle repository in ambito di studi sull'evoluzione del software può riportare più di qualche problema a causa della meccanica del branching. Se è vero che questo

fattore è stato già valutato nell'ambito della fault-prediction [39], non è detto che i risultati siano gli stessi quando si parla di studi che mirano ad analizzare la storia di un progetto. Anche il calcolo delle metriche relative al riuso, e in particolare di quelle relative all'ereditarietà, dovrebbe essere oggetto di studio più approfondito: in questa tesi, infatti, le due metriche hanno rappresentato il numero di classi ad utilizzare quel meccanismo, mentre un calcolo a più basso livello - ovvero a livello di classe, di utilizzo dei metodi e dello stato della superclasse - potrebbe condurre a risultati diversi da quelli ottenuti in questa sede.

Laddove è stata trovata una correlazione tra la presenza dei difetti e l'utilizzo del riuso, non è stato possibile trarre una conclusione comune per tutti i progetti, in quanto le metriche di riuso hanno avuto impatti a volte opposti tra un sistema e l'altro. Questo suggerisce che l'impatto che il riuso può avere sulla qualità del software non dipende meramente dal meccanismo in sé, ma anche da come questo viene utilizzato. Ciò significa che potrebbe essere necessario condurre analisi qualitative per stabilire i limiti entro i quali l'utilizzo dell'ereditarietà e della delegazione può considerarsi corretto e utile.

APPENDICE A

Appendice RQ2

Tabella A.1: CommonsCollections - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffNOC	-0.038 (0.050)	0.004 (0.013)
diffLCOM	0.422 (0.808)	0.476 (22.615)
diffDIT	0.012 (5.830)	-0.0003 (0.023)
diffCBO	-0.878 (58.069)	-0.495*** (0.103)
diffRFC	4.123 (5.784)	-0.030 (1.106)
diffLOC	-0.611 (1.053)	-0.099 (11.153)
diffdelegations	-0.069 (0.137)	0.004 (0.654)
diffintInh	1.013 (6.087)	1.924 (14.548)
diffimpInh	0.767 (3.457)	0.771 (13.557)
churns	-0.026 (0.038)	-0.100 (0.127)
Constant	-6.429*** (0.536)	-6.230*** (0.527)
Akaike Inf. Crit.	165.190	165.190

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.2: CommonsCsv - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	-3.539 (2.559)	0.627 (5.136)
diffNOC	-4.413*** (0.156)	1.052*** (0.176)
diffLCOM	-0.056 (0.066)	-0.040 (0.130)
diffDIT	-4.526*** (0.238)	0.661*** (0.169)
diffCBO	-0.994 (4.485)	-3.484 (8.728)
diffLOC	0.175 (0.121)	0.045 (0.236)
diffdelegations	0.031 (0.076)	-0.058 (0.147)
diffintInh	-0.571 (5.267)	-1.495 (10.178)
diffimpInh	-1.141 (2.923)	-0.094 (4.432)
churns	-0.004 (0.007)	-0.009 (0.013)
Constant	-4.605*** (0.265)	-4.520*** (0.264)
Akaike Inf. Crit.	399.415	399.415

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.3: Compress - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	-5.248*** (0.033)	-1.903 (4.387)
diffNOC	10.188*** (0.002)	-5.653*** (0.051)
diffLCOM	-0.066 (0.335)	0.046 (0.125)
diffDIT	12.511*** (0.002)	-5.151*** (0.125)
diffCBO	-4.163*** (0.021)	1.467 (2.717)
diffLOC	0.149* (0.090)	0.024 (0.104)
diffdelegations	0.013 (0.017)	-0.003 (0.013)
diffintInh	-0.187 (0.862)	-0.148 (0.637)
diffimpInh	-0.337 (0.488)	0.154 (0.383)
churns	-0.003 (0.002)	-0.0003 (0.001)
Constant	-4.236*** (0.160)	-4.327*** (0.159)
Akaike Inf. Crit.	948.921	948.921

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.4: Gson - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	3.261 (30.899)	-1.305 (25.907)
diffNOC	-0.159 (0.275)	1.536*** (0.415)
diffLCOM	0.013 (1.242)	-0.159 (1.157)
diffDIT	0.696 (0.466)	1.896** (0.798)
diffCBO	-17.977 (12.462)	-3.472 (12.553)
diffLOC	0.115 (1.273)	0.689 (1.373)
diffdelegations	0.068 (0.059)	-0.018 (0.078)
diffintInh	-0.356 (3.632)	0.226 (1.865)
diffimpInh	0.047 (2.105)	0.267 (1.713)
churns	-0.015 (0.014)	-0.007 (0.009)
Constant	-4.910*** (0.369)	-5.051*** (0.372)
Akaike Inf. Crit.	248.017	248.017

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.5: JacksonCore - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffNOC	−37.085*** (0.359)	−93.807*** (0.372)
diffLCOM	−0.024 (0.029)	−0.008 (0.031)
diffCBO	−2.840 (5.329)	−8.386 (5.859)
diffLOC	−0.039 (0.053)	0.039 (0.047)
diffdelegations	0.001 (0.003)	0.003 (0.003)
diffintInh	−0.371 (0.439)	0.491 (0.407)
churns	−0.001 (0.002)	−0.004 (0.004)
Constant	−4.183*** (0.237)	−4.082*** (0.237)
Akaike Inf. Crit.	509.727	509.727

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.6: JacksonDatabind - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	-2.330 (1.546)	3.344** (1.619)
diffNOC	-58.836*** (0.037)	-152.598*** (0.043)
diffLCOM	0.155*** (0.049)	0.179*** (0.045)
diffDIT	-70.763*** (0.028)	-124.104*** (0.029)
diffCBO	1.062 (0.979)	2.261* (1.194)
diffLOC	-0.045 (0.147)	-0.013 (0.141)
diffdelegations	-0.005 (0.003)	-0.010*** (0.003)
diffintInh	0.109 (0.095)	-0.065 (0.100)
diffimpInh	0.018 (0.099)	0.341*** (0.090)
churns	-0.001 (0.001)	-0.004** (0.002)
Constant	-4.048*** (0.118)	-4.043*** (0.127)
Akaike Inf. Crit.	1,804.498	1,804.498

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.7: JacksonXml - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffNOC	-0.066*** (0.020)	-0.235*** (0.007)
diffLCOM	-0.016 (0.244)	-0.420 (0.478)
diffCBO	-1.162 (3.528)	18.673* (10.142)
diffLOC	-0.009 (0.228)	0.077 (0.639)
diffdelegations	0.027 (0.043)	0.005 (0.066)
diffintInh	-0.489 (2.984)	1.159 (5.170)
diffimpInh	-0.461 (1.068)	-0.430 (2.108)
churns	0.002 (0.006)	-8.872*** (0.001)
Constant	-5.312*** (0.440)	-5.346*** (0.518)
Akaike Inf. Crit.	165.934	165.934

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.8: JxPath - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffWMC	−14.452*** (2.914)	41.577*** (3.503)
diffNOC	4.203*** (0.113)	−1.798*** (0.040)
diffLCOM	0.217 (1.161)	−1.167 (1.509)
diffCBO	−11.521 (12.928)	−28.867*** (9.482)
diffRFC	8.152 (7.483)	−47.303*** (8.690)
diffLOC	−3.004* (1.736)	5.674*** (1.428)
diffdelegations	0.201* (0.104)	0.399*** (0.100)
diffintInh	−4.686* (2.742)	0.161 (1.719)
diffimpInh	−2.314 (1.718)	−15.482*** (4.800)
churns	−0.015* (0.008)	−0.026*** (0.007)
Constant	−3.345*** (0.262)	−3.323*** (0.264)
Akaike Inf. Crit.	331.609	331.609

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella A.9: Time - applicazione del modello multinomiale

	<i>Dependent variable:</i>	
	Decrease	Increase
	(1)	(2)
diffNOC	-1.001*** (0.007)	-0.723*** (0.021)
diffLCOM	-0.835 (0.867)	-0.255 (2.234)
diffCBO	5.642*** (0.319)	-7.746*** (0.028)
diffLOC	0.645 (0.699)	1.564 (2.364)
diffdelegations	0.032 (0.051)	-0.074 (0.117)
diffintInh	-1.633 (3.875)	-6.170*** (0.250)
diffimpInh	-1.313 (2.317)	1.200 (4.509)
churns	-0.006 (0.006)	-0.019 (0.013)
Constant	-4.545*** (0.243)	-4.462*** (0.251)
Akaike Inf. Crit.	449.579	449.579

Note: *p<0.1; **p<0.05; ***p<0.01

APPENDICE B

Appendice RQ3

Tabella B.1: CommonsCollections - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffNOC	10,213.080*** (2,341.143)
diffLCOM	7.799* (4.680)
diffDIT	-2,378.489 (1,497.270)
diffCBO	6,717.225*** (652.191)
diffRFC	4.682 (60.012)
diffLOC	-58.840*** (10.760)
delegations	-0.003 (0.057)
intInh	1.143 (1.211)
impInh	-1.026 (0.870)
Bug2Decrease	-106.139 (620.995)
Bug2Increase	-111.854 (620.996)
Constant	103.349 (77.155)
Observations	3,560
Log Likelihood	-30,405.490
Akaike Inf. Crit.	60,834.970

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.2: CommonsCsv - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
	churns
diffWMC	−20.375 (56.965)
diffNOC	−132.074 (1,357.614)
diffLCOM	10.673*** (1.905)
diffDIT	−1,787.167 (1,192.199)
diffCBO	−56.282 (94.958)
diffLOC	2.994 (2.693)
delegations	0.165 (0.124)
intInh	−6.889 (6.984)
impInh	−0.386 (4.387)
Bug2Decrease	−1.272 (69.685)
Bug2Increase	−14.892 (69.652)
Constant	11.161 (20.003)
Observations	1,634
Log Likelihood	−11,501.520
Akaike Inf. Crit.	23,027.050

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.3: Compress - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffWMC	-1,988.919*** (210.722)
diffNOC	-10,740.970*** (1,699.369)
diffLCOM	26.285*** (6.021)
diffDIT	52,852.530** (2,813.141)
diffCBO	5,529.115*** (307.003)
diffLOC	5.769 (5.471)
delegations	-0.080*** (0.022)
intInh	-0.710 (1.950)
impInh	3.653*** (1.093)
Bug2Decrease	-26.208 (85.330)
Bug2Increase	-15.856 (86.311)
Constant	91.271*** (25.693)
Observations	3,305
Log Likelihood	-25,563.430
Akaike Inf. Crit.	51,150.860

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.4: Gson - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffWMC	-377.039 (269.203)
diffNOC	17,827.570*** (3,288.230)
diffLCOM	-15.488 (12.955)
diffDIT	-6,958.231** (2,826.673)
diffCBO	1,916.428*** (145.944)
diffLOC	46.604*** (10.894)
delegations	-0.119** (0.050)
intInh	1.415 (1.301)
impInh	2.080 (2.226)
Bug2Decrease	-6.128 (90.985)
Bug2Increase	-14.031 (96.408)
Constant	111.100* (63.054)
Observations	1,478
Log Likelihood	-10,372.340
Akaike Inf. Crit.	20,768.680

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.5: JacksonCore - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffNOC	21,588.520*** (1,212.595)
diffLCOM	1.241 (0.765)
diffCBO	1,682.687*** (131.899)
diffLOC	28.585*** (1.371)
delegations	-0.012 (0.017)
intInh	2.701 (3.677)
Bug2Decrease	83.885* (48.156)
Bug2Increase	-51.820 (48.181)
Constant	25.537 (47.505)
Observations	1,543
Log Likelihood	-10,602.590
Akaike Inf. Crit.	21,223.180

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.6: JacksonDatabind - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
	churns
diffWMC	853.627*** (71.347)
diffNOC	22,830.430*** (1,564.318)
diffLCOM	−28.862*** (1.208)
diffDIT	50,782.460*** (1,712.723)
diffCBO	3,147.449*** (74.440)
diffLOC	−8.353*** (3.029)
delegations	0.010* (0.006)
intInh	−1.281*** (0.305)
impInh	−0.140 (0.499)
Bug2Decrease	28.236 (19.492)
Bug2Increase	−7.043 (20.359)
Constant	182.664*** (57.987)
Observations	5,228
Log Likelihood	−34,772.800
Akaike Inf. Crit.	69,569.600

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.7: JacksonXml - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffNOC	333.786 (509.649)
diffLCOM	-8.269*** (0.992)
diffCBO	239.229*** (19.892)
diffLOC	8.568*** (0.946)
delegations	-0.096** (0.044)
intInh	-2.847 (2.324)
impInh	3.001** (1.408)
Bug2Decrease	19.831 (23.738)
Bug2Increase	-20.624 (26.005)
Constant	57.931*** (9.413)
Observations	1,128
Log Likelihood	-6,173.937
Akaike Inf. Crit.	12,367.870

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.8: JxPath - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffWMC	-889.089 (1,208.343)
diffNOC	24,786.920*** (5,760.288)
diffLCOM	21.501*** (5.505)
diffCBO	3,504.462*** (363.997)
diffRFC	1,358.532*** (363.735)
diffLOC	-158.255*** (12.875)
delegations	-0.598** (0.294)
intInh	6.773 (14.133)
impInh	0.942 (3.559)
Bug2Decrease	-41.147 (149.800)
Bug2Increase	14.086 (149.626)
Constant	1,561.449*** (387.616)
<hr/>	
Observations	598
Log Likelihood	-4,669.763
Akaike Inf. Crit.	9,363.526

Note: *p<0.1; **p<0.05; ***p<0.01

Tabella B.9: Time - applicazione del modello lineare generalizzato

<i>Dependent variable:</i>	
churns	
diffNOC	54,104.760*** (3,864.815)
diffLCOM	189.720*** (23.536)
diffCBO	-31,929.670*** (2,814.595)
diffLOC	344.715*** (42.978)
delegations	-0.580*** (0.107)
intInh	-156.445*** (22.026)
impInh	179.745*** (20.294)
Bug2Decrease	-625.949 (602.612)
Bug2Increase	-690.321 (618.444)
Constant	-6,729.363*** (706.442)
<hr/>	
Observations	2,094
Log Likelihood	-19,418.210
Akaike Inf. Crit.	38,856.420

Note: *p<0.1; **p<0.05; ***p<0.01

Bibliografia

- [1] G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino, "On the evolution of inheritance and delegation mechanisms and their impact on code quality," (Citato alle pagine 1, 2, 3, 9 e 17)
- [2] D. Castro and C. Werner, "Systematic mapping on software reuse teaching," in *2021 12th International Conference on Information and Communication Systems (ICICS)*, pp. 257–264, 2021. (Citato a pagina 1)
- [3] J. M. Bieman and J. X. Zhao, "Reuse through inheritance: A quantitative study of c++ software," *SIGSOFT Softw. Eng. Notes*, vol. 20, p. 47–52, aug 1995. (Citato a pagina 2)
- [4] V. Rajlich, "Software evolution and maintenance," in *Future of Software Engineering Proceedings*, FOSE 2014, (New York, NY, USA), p. 133–144, Association for Computing Machinery, 2014. (Citato a pagina 2)
- [5] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "A Comprehensive Technique to Predict the Size of Maintenance Issues," 2 2020. (Citato alle pagine 2 e 7)
- [6] D. Galin, *Software quality assurance: from theory to implementation*. Pearson education, 2004. (Citato a pagina 3)
- [7] L. H. Rosenberg and L. E. Hyatt, "Software quality metrics for object-oriented environments," *Crosstalk journal*, vol. 10, no. 4, pp. 1–6, 1997. (Citato a pagina 3)
- [8] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994. (Citato a pagina 3)

- [9] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005. (Citato alle pagine 3 e 8)
- [10] E. Nasser, S. Counsell, and M. Shepperd, "An empirical study of evolution of inheritance in java oss," in *19th Australian Conference on Software Engineering (aswec 2008)*, pp. 269–278, 2008. (Citato a pagina 3)
- [11] F. Brito e Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," *Proceedings of the 3rd International Software Metrics Symposium*, 03 1996. (Citato alle pagine 3 e 6)
- [12] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994. (Citato a pagina 6)
- [13] L. Prechelt, B. Unger, M. Philippson, and W. Tichy, "A controlled experiment on inheritance depth as a cost factor for code maintenance," *Journal of Systems and Software*, vol. 65, no. 2, pp. 115–126, 2003. (Citato a pagina 6)
- [14] J. M. M. R. J. Daly, A. Brooks and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Software Engineering*, vol. 1, no. 2, 1996. (Citato a pagina 6)
- [15] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using oo metrics. an industrial case study," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pp. 99–107, 2002. (Citato a pagina 6)
- [16] A. Chhikara, R. Chhillar, and S. Khatri, "Evaluating the impact of different types of inheritance on the object oriented software metrics," *International Journal of Enterprise Computing and Business Systems*, vol. 1, no. 2, pp. 1–7, 2011. (Citato a pagina 6)
- [17] V. A, S. S., and C. Babu, "Evaluation of reusability in aspect oriented software using inheritance metrics," 05 2014. (Citato a pagina 7)
- [18] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69–82, 2006. (Citato a pagina 7)

- [19] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, "Is lines of code a good measure of effort in effort-aware models?," *Information and Software Technology*, vol. 55, no. 11, pp. 1981–1993, 2013. (Citato a pagina 7)
- [20] F. Thung, "Automatic prediction of bug fixing effort measured by code churn size," in *Proceedings of the 5th International Workshop on Software Mining*, SoftwareMining 2016, (New York, NY, USA), p. 18–23, Association for Computing Machinery, 2016. (Citato a pagina 7)
- [21] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 432–441, IEEE Press, 2013. (Citato a pagina 7)
- [22] Y. Kamei, E. Shihab, B. Adams, A. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013. Copyright: Copyright 2013 Elsevier B.V., All rights reserved. (Citato a pagina 7)
- [23] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, 2006. (Citato a pagina 7)
- [24] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software Quality Journal*, vol. 18, pp. 3–35, 03 2010. (Citato a pagina 8)
- [25] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 908–911, Association for Computing Machinery, 2018. (Citato a pagina 8)
- [26] M. Aniche, "Repodriller." <https://github.com/mauricioaniche/repodriller>. (Citato a pagina 8)
- [27] A. Trautsch, F. Trautsch, S. Herbold, B. Ledel, and J. Grabowski, *The SmartSHARK Ecosystem for Software Repository Mining*, p. 25–28. New York, NY, USA: Association for Computing Machinery, 2020. (Citato a pagina 8)

- [28] N. Riquet, X. Devroey, and B. Vanderose, "Presentation abstract : A first case study about key socio-technical software indicators at Forem." 20th Belgium-Netherlands Software Evolution Workshop, BENEVOL '21, 2021. (Citato a pagina 8)
- [29] P. Shrivastava, "Neural code summarization," 2021. (Citato a pagina 8)
- [30] M. Steinhauer and F. Palomba, *Speeding up the Data Extraction of Machine Learning Approaches: A Distributed Framework*, p. 13–18. New York, NY, USA: Association for Computing Machinery, 2020. (Citato a pagina 8)
- [31] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019. (Citato a pagina 9)
- [32] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014. (Citato a pagina 11)
- [33] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *CoRR*, vol. abs/1811.02429, 2018. (Citato a pagina 11)
- [34] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset." working paper or preprint, June 2015. (Citato a pagina 11)
- [35] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science China Information Sciences*, vol. 62, Oct. 2019. (Citato a pagina 11)
- [36] D. Yang, K. Liu, D. Kim, A. Koyuncu, K. Kim, H. Tian, Y. Lei, X. Mao, J. Klein, and T. F. Bissyandé, "Where were the repair ingredients for defects4j bugs? exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems," *Empirical Softw. Engg.*, vol. 26, nov 2021. (Citato a pagina 11)
- [37] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 130–140, 2018. (Citato a pagina 11)

- [38] D. Spadini, "Modified files are not fetched for merge commits." <https://github.com/ishepard/pydriller/issues/89>. (Citato a pagina 14)
- [39] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: Should we consider branches?," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 202–213, 2018. (Citato alle pagine 21 e 65)