Indice

Introduzione

Sono detti cloni software coppie di segmenti di codice (sequenze di linee di codice) sorgente simili tra loro, secondo una qualche definizione di similarità [Baxter, 2002], da un punto di vista lessicale, sintattico o semantico. Due segmenti di codice, quindi, sono cloni se uno è un duplicato dell'altro (più o meno esatto dal punto di vista lessicale e sintattico), o se presentano una similarità semantica in quanto implementano una stessa funzione (hanno pre- e post-condizioni simili).

La presenza di cloni in un sistema software può inficiarne la qualità, soprattutto per quanto riguarda la sua manutenzione ed evoluzione. Si pensi, ad esempio, al caso in cui è rilevato un errore in un segmento di codice clonato più volte nel sistema: in tal caso la stessa modifica per rimuovere l'errore deve essere applicata a tutti i segmenti clonati. Ciò richiede la precisa conoscenza dell'esistenza dei cloni e della loro localizzazione nel sistema, e comunque comporta un maggior onere della necessaria modifica (che va, appunto, ripetuta per ciascun clone).

In letteratura sono presenti varie metodologie, tecniche e tool per individuare cloni in un sistema software al fine di documentarne e segnalarne la presenza, in modo da poter poi eliminarne o ridurne la presenza attuando adeguate attività di manutenzione, quali ad esempio la loro rifattorizzazione in moduli.

Nella gestione dei cloni software presenti in un sistema software (in particolare per sistemi open source), aspetti interessanti sono:

- capire chi, tra sviluppatori e/o manutentori (i committer) di un sistema, ha introdotto i cloni in esso presenti, col fine di capire se c'è qualche committer maggiormente propenso ad introdurne, se clona proprio codice o quello prodotto da altri, quali possono essere motivazioni inducenti i committer all'introduzione dei cloni;
- capire come un clone o un insieme di cloni evolvono tra le successive release del sistema per capire se un clone continua ad esistere nella versione

successiva, se è aggiunto un nuovo clone a qualcuno di quelli già esistenti nella versione precedente, se è aggiunto un clone completamente nuovo non esistente nella release precedente.

- capire se i cloni introdotti nella nuova release sono stati introdotti da committer che ne avevano già inseriti nelle release precedenti.

Tali informazioni sono utili a migliorare la gestione dello sviluppo ed evoluzione del sistema.

Questo lavoro di tesi si interessa degli ultimi due punti del precedente elenco, basandosi sui risultati di un precedente lavoro di tesi relativo agli aspetti descritti nel primo punto dell'elenco.

Lo scopo. quindi, è di analizzare come i cloni software presenti in una release di un sistema software (individuati secondo quanto sviluppato nel un precedente lavoro) evolvono nelle release successive, al fine di verificare se un clone continua a persistere nelle release successive, se sono introdotti nuovi cloni non presenti nella release precedente, se sono introdotti nuovi cloni di uno già esistente in precedenza. Inoltre i cloni individuati sono associati ai committer che li hanno introdotti e, quindi, si vuole anche analizzare se vi sono commiter che sono maggiormente propensi ad introdurre cloni nelle varie release.

I cloni sono individuati usando l'approccio definito nel precedente lavoro di tesi in cui era stata definita una tool-chain che permette di analizzare i file sorgenti registrati in un repository per la gestione di sistemi sw open-source, insieme con le informazioni relative ai vari commit relativi a tali sorgenti, riportanti i rispettivi committer, di individuare i cloni esistenti nei file sorgenti ed 'fondere' le informazioni ottenute con quelle relative ai commit e committers, in modo da associare ciascun clone al committer che aveva manipolato il file e quindi poter essere ritenuto (l'ultimo) responsabile dell'introduzione del clone. Le informazioni così elaborate sono memorizzate in un data base relazionale che può essere interrogato per ottenere sia le liste di cloni e committers che le associazioni tra essi. Il

tool Clo_tter permette di gestire le operazioni della tool-chain e di interrogare la base di dati.

In questo lavoro di tesi sono state estese le funzionalità della tool-chain ed in particolare del tool Clo_tter in modo da poter perseguire gli obiettivi prefissati per tenere traccia dell'evoluzione dei cloni dalla prima versione del sistema fino all'ultima disponibile per l'analisi.

In particolare, data una release del sistema sw, si è interessati a conoscerne le differenze in termini di cloni o classi di cloni (insieme di cloni con una definita soglia di similarità) con le altre versioni per individuare, dove possibile, nuove istanze o di cloni o di classi di questi. Ad esempio, si considera un particolare clone o classe di cloni per una certa versione e si valuta la sua persistenza nella versione successiva e se la classe del clone analizzato presenta una cardinalità diversa da quella che la stessa classe aveva nella versione precedente ciò è indice di una eliminazione o introduzione di nuovi cloni nella classe. Di ciascun clone o classe di cloni introdotto in ciascuna release del sistema è riportata l'associazione con il committer ad esso relativo per poterne delineare il comportamento dello stesso rispetto all'evoluzione dei cloni.

Sono stati effettuati esperimenti in cui sono state analizzate alcune release successive di sistemi software open source prelevati da GitHub per verificare e validare le estensioni realizzate alla tool-chain ottenendo risultati significativi.

La tesi è strutturata secondo i seguenti capitoli:
Capitolo 1: Cloni software; il capitolo riporta i concetti relativi ai cloni software ed alla loro identificazione.
Capitolo 2: Il tool di Clone Detection NiCad; è descritto il tool NiCad usato per la identificazione di cloni software nei sistemi
Capitolo 3: Il sistema di controllo di versione Git; è descritta la struttura del sistema di hosting di sistemi open source GitHub

Capitolo 4: Il tool Sourcetree; è descritto descrive il tool Sourcetree utilizzato per prelevare le informazioni di interesse da GitHub.

Capitolo 5: Il tool Clo_tter esteso; è descritto il tool che permette di effettuare l'associazione tra committer e cloni di una release di un sistema software, arricchito con le nuove funzionalità per analizzare come i cloni evolvono tra le varie release del sistema.

Capitolo 6: Conclusioni; riporta le principali conclusioni del lavoro svolto e possibili sviluppi futuri.

Capitolo 1: Cloni software

1.1 Introduzione

Sono detti cloni software coppie di segmenti di codice (sequenze di linee di codice) sorgente simili tra loro secondo una qualche definizione di similarità [Baxter], [Koske] da un punto di vista lessicale, sintattico o semantico. In un sistema software è possibile avere cloni software all'interno di uno stesso programma o tra programmi/moduli differenti.

Due segmenti di codice, quindi, sono cloni se uno è un duplicato dell'altro (più o meno esatto dal punto di vista lessicale e sintattico), o se presentano una similarità semantica in quanto implementano una stessa funzione (hanno pre- e post-condizioni simili). Nonostante i linguaggi di programmazione offrono vari meccanismi di astrazione per facilitare l'incapsulamento e riutilizzo di codice, il metodo del "copia-ed-incolla" ("copy-and-paste"), è ancora molto usato per copiare e incollare un segmento di codice in un'altra posizione dello stesso programma o in un altro modulo per replicare lo stesso comportamento in più punti del sistema sw. Ciò causa la coesistenza di più copie di segmenti di codice esatti o molto simili nel sistema. Spesso questa operazione di copia è accompagnata da leggere modifiche nel codice clonato come la ridenominazione delle variabili o l'inserimento/eliminazione di parte del codice, per meglio adattarlo al nuovo contesto.

Vari sono i motivi per cui si effettua la duplicazione/clonazione di codice.

Tra questi:

- uno stretto e rigido vincolo temporale, o un'urgenza, che spinge gli utilizzatori a riusare, copiandolo, codice che implementa un comportamento simile a quello richiesto, rimandando a tempi successivi una migliore soluzione

progettuale;

- il riuso di codice affidabile, per evitare il rischio di introdurre difetti con la produzione di codice nuovo
- la mancanza nel linguaggio di programmazione di costrutti che favoriscono e permettono l'incapsulamento e riuso del codice stile e modello mentale dello sviluppatore che lo porta, più o meno inconsapevolmente, a definire ed usare stessi/simili pattern di programmazione.

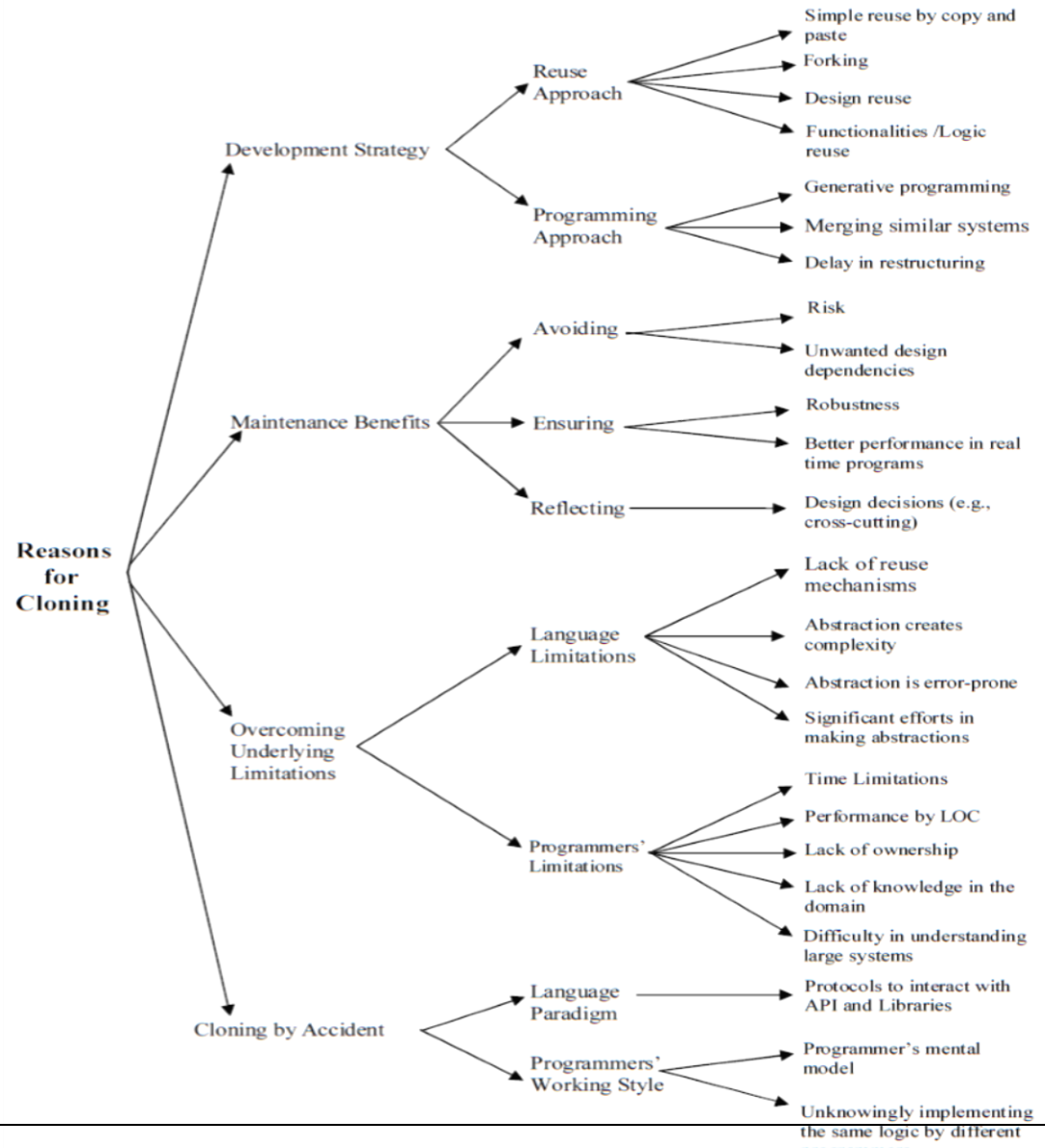


Figura 1.1 Principali Motivi della clonazione del codice

La figura 1.1 riassume i vari motivi che portano alla clonazione di codice

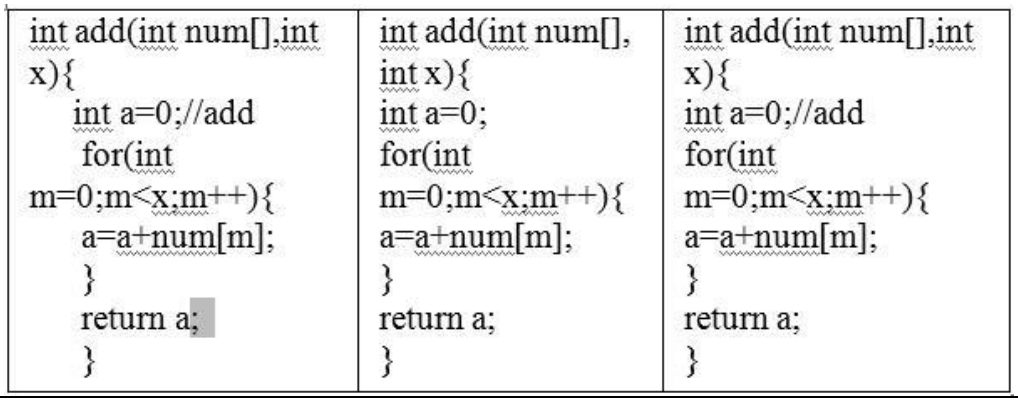
La *clonazione* può presentare alcuni vantaggi come il risparmio di tempo di sviluppo e aumento della percentuale di riuso del codice, ma essa è in considerata dannosa per la qualità specie la presenza di cloni può rendere più oneroso il testing, la manutenzione ed evoluzione del software,

Si pensi, ad esempio, al caso in cui è rilevato un errore in un segmento di codice clonato più volte nel sistema: in tal caso la stessa modifica per rimuovere l'errore deve essere applicata a tutti i segmenti clonati. Ciò richiede la precisa conoscenza dell'esistenza dei cloni e della loro localizzazione nel sistema, e comunque comporta un maggior onere della necessaria modifica (che va, appunto, ripetuta per ciascun clone).

1.2 Tipi di clone

In base al modo con cui un segmento di codice può essere clonato sono state definite, in letteratura, quattro tipi di cloni software [Koske] [Kaur]:

1. *Cloni esatti (exact clones)*: frammenti di codice identici eccetto per variazioni di spazi, commenti e layout; la figura 1.2 riporta un esempio di cloni esatti (tipo 1); in essi troviamo la stessa sequenza di istruzioni con stessi nomi delle variabili e con differenze relative ad aggiunta/eliminazione di commenti/spazi, o differente indentazione delle istruzioni.



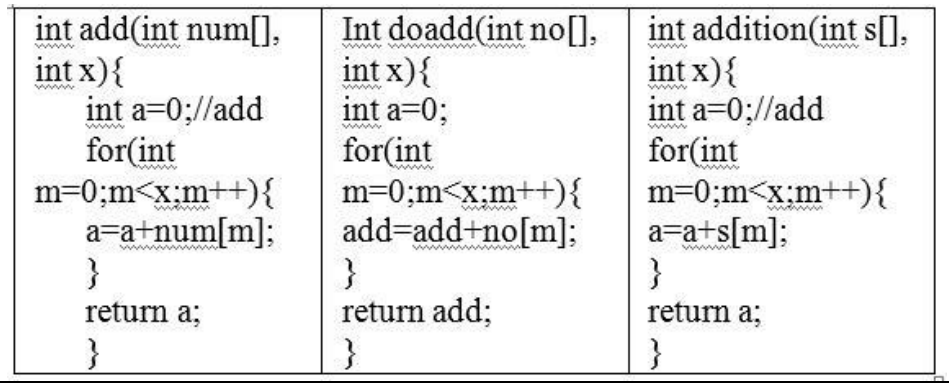
```
int add(int num[],int x){
    int a=0;//add
     for(int m=0;m<x;m++){
     a=a+num[m];
     }
     return a;
     }
```

```
int add(int num[], int x){
int a=0;
for(int m=0;m<x;m++){
a=a+num[m];
}
return a;
}
```

```
int add(int num[],int x){
int a=0;//add
for(int m=0;m<x;m++){
a=a+num[m];
}
return a;
}
```

Figura 1.2 – Esempio di cloni esatti (tipo 1)

2. Cloni rinominati o parametrizzati (renamed clones): frammenti di codice strutturalmente/sintatticamente identici eccetto per variazioni dei nomi degli identificatori, letterali, tipi, per gli spazi bianchi, il layout e commenti. La figura 1.3 mostra un esempio di cloni di tipo 2; in essi troviamo la stessa sequenza di istruzioni ma i nomi di identificatori e/o variabili sono differenti e con differenze relative ad aggiunta/eliminazione di commenti/spazi, o differente indentazione delle istruzioni, mentre il comportamento è lo stesso.



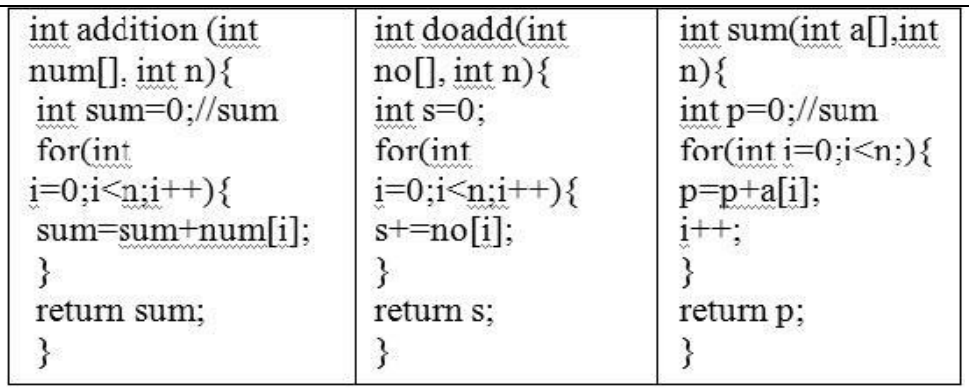
```
int add(int num[], int x){
    int a=0;//add
    for(int m=0;m<x;m++){
    a=a+num[m];
    }
    return a;
    }
```

```
Int doadd(int no[], int x){
int a=0;
for(int m=0;m<x;m++){
add=add+no[m];
}
return add;
}
```

```
int addition(int s[], int x){
int a=0;//add
for(int m=0;m<x;m++){
a=a+s[m];
}
return a;
}
```

Figura 1.3 – Esempio di cloni di tipo 2

3. *Near-Miss Clones:* frammenti di codice simili tra loro ma con modifiche come l'aggiunta o rimozione di istruzioni, identificatori e/o modifiche come nei tipi 1 o 2 . La figura 1.4 riporta un esempio di cloni di tipo 3



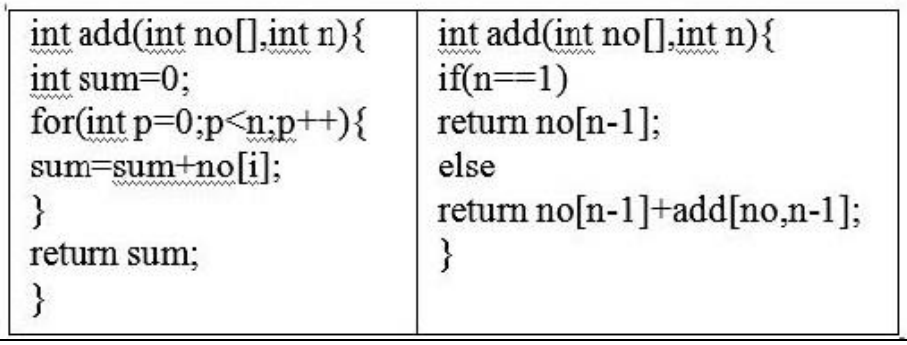
```
int addition (int
num[], int n){
 int sum=0;//sum
 for(int
i=0;i<n;i++){
 sum=sum+num[i];
 }
 return sum;
 }
```

```
int doadd(int
no[], int n){
int s=0;
for(int
i=0;i<n;i++){
s+=no[i];
}
return s;
}
```

```
int sum(int a[],int
n){
int p=0;//sum
for(int i=0;i<n;){
p=p+a[i];
i++;
}
return p;
}
```

Figura 1.4 – Esempio di cloni di tipo 3

4. *Cloni semantici (tipo 4)*: frammenti di codice che eseguono la stessa elaborazione (hanno lo stesso comportamento funzionale) ma implementati con differenti varianti sintattiche. La figura 1.5 riporta un esempio di cloni di tipo 4



```
int add(int no[],int n){
int sum=0;
for(int p=0;p<n;p++){
sum=sum+no[i];
}
return sum;
}
```

```
int add(int no[],int n){
if(n==1)
return no[n-1];
else
return no[n-1]+add[no,n-1];
}
```

Figura 1.5 - Esempio di cloni di tipo 4

La maggior parte degli studi per il rilevamento di cloni in un sistema software sono relativi all'individuazione di cloni di tipo 1, 2 e 3.

1.3 Tecniche di individuazione cloni

L'operazione di "rilevazione dei cloni" può essere conseguita attraverso diverse tecniche di classificazione, successivamente riportate:

- *Text based*: la rilevazione viene eseguita in base alla somiglianza testuale piuttosto che sulla base della somiglianza sintattica e semantica
-
- *Abstract Syntax Tree based*: i cloni sono rilevati comparando gli alberi astratti della sintassi che rappresentano i frammenti da confrontare, è quindi effettuato un confronto basato sulla struttura sintattica del codice.

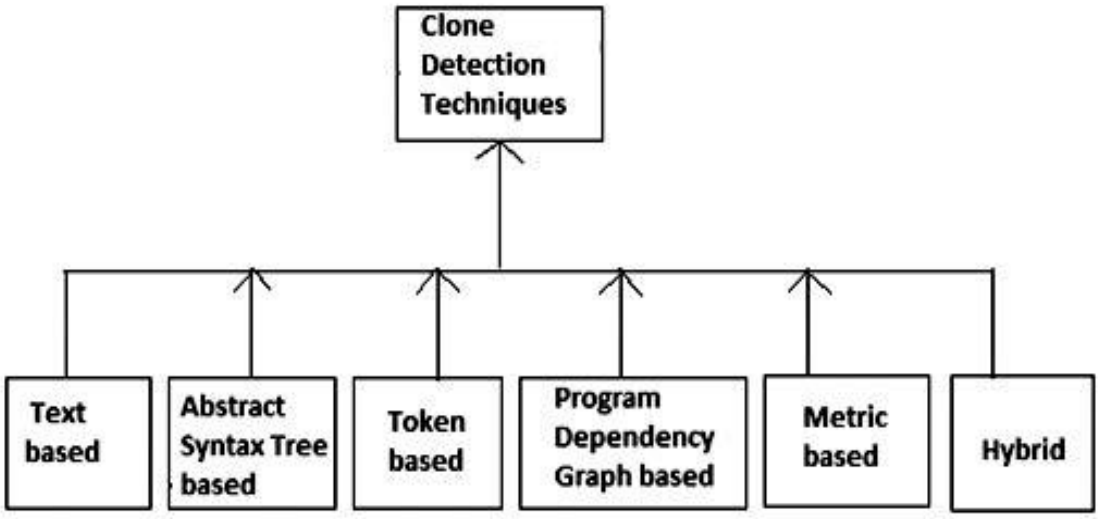


Figura 1.6 – Tecniche di classificazione cloni

- *Token based*: è effettuata un'analisi lessicale del codice che è convertito in una sequenza di token. La sequenza di token è analizzata per individuare sottosequenze uguali che corrispondono a frammenti. Cloni esatti e cloni sintattici sono tipicamente individuati con questa tecnica.

- *Programm Dependency Graph based*: partendo dal codice sorgente viene realizzato il grafo delle dipendenze del programma che include il controllo del flusso e il flusso dei dati. Il grafo viene analizzato per individuare sottografi simili che corrispondono a frammenti clonati

- *Metric based*: sono calcolate significative misure del codice che tengono conto di vari aspetti circa la struttura del codice e dei dati, i nomi di variabili, i letterali. Le parti del codice che evidenziano specificate caratteristiche metriche comparabili sono considerate come cloni.

- *Hybrid*: combinano due o più tecniche prima citate per rilevare i cloni; questa tecnica fornisce un risultato migliore rispetto alle tecniche normali. Ad esempio: tecniche basate su grafi e su metriche possono essere utilizzate in combinazione per ottenere risultati più precisi.

1.4 Processo di individuazione clone

La figura 1.7 rappresenta un tipico processo per la identificazione di cloni in un sistema.

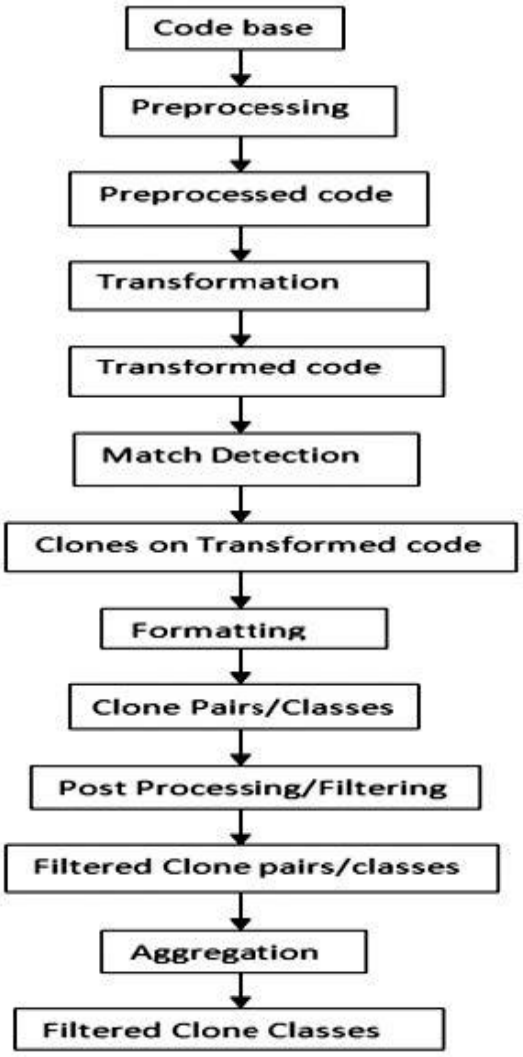


Figura 1.7 – Processo identificazione cloni

Questo processo è piuttosto costoso e richiede una buona velocità di calcolo. I cloni vengono rilevati in base alla somiglianza.

Preprocessing: questa fase prevede due passaggi: nel primo, conosciuto come segmentazione, viene suddiviso il codice sorgente in sezioni, nel secondo vengono individuate le aree di confronto. Questa fase comprende:

1) Eliminazione di parti indesiderate: il codice sorgente è segmentato e le parti non di interesse vengono rimosse, il che può generare falsi valori positivi.

2) Definire le unità sorgenti: una volta completata la rimozione del codice indesiderato, il resto del codice sorgente è partizionato in modo tale da poter ottenere una porzione comune. Per esempio: in un programma; file, classi, funzioni / metodi, blocchi di inizio o fine o sequenza di linee sorgente.

3) Definire le unità da confrontare: segmentazione delle unità sorgenti per ottenere ulteriori unità più piccole per il confronto.

Trasformation: in questa fase le unità del codice sorgente vengono convertite in particolari rappresentazioni intermedie. Questo passaggio è ulteriormente suddiviso in:

1) Estrazione: per rendere il codice sorgente appropriato come input per l'algoritmo reale viene effettuata la conversione

2) Tokenizzazione: ogni riga del codice sorgente è isolata in token.

3) Parsing: per individuare i cloni nell'approccio sintattico, viene utilizzato un albero astratto della sintassi per il confrontare sottoalberi. È anche possibile utilizzare l'approccio basato su metrica.

Match detection: il codice trasformato ottenuto dai passaggi precedenti viene immesso in un algoritmo di confronto in cui tutte le unità trasformate da confrontare vengono valutate in base alla loro similarità per determinare eventuali corrispondenze. L'algoritmo produce un insieme di coppie di candidati cloni. Gli algoritmi utilizzati in questa fase sono: suffix tree dynamic pattern matching e hash esteem examination.

Formatting: l'elenco di coppie di cloni ottenuto dall'algoritmo di confronto viene trasformato nel relativo elenco di coppie di cloni del codice originale.

Post processing/filtering: questo passaggio è ulteriormente suddiviso in due parti:

1) analisi manuale: qui i falsi positivi vengono filtrati da esperti umani.

2) euristica automatizzata: alcuni parametri sono già impostati in base agli scopi di filtraggio. Per esempio: lunghezza, frequenza, diversità ecc.

Aggregation: per ridurre la quantità di informazioni e/o facilitarne l'analisi, i cloni possono essere raccolti in classi di cloni per eseguire successivi esami o misurazioni.

Capitolo 2: Il tool di Clone Detection NiCad

2.1 Principali caratteristiche di NiCad

NiCad [Nicad] è uno strumento scalabile e flessibile di rilevamento di cloni, utilizzabile da riga di comando, che riceve come input una directory sorgente, in cui sono contenuti i file sorgente da esaminare, e fornisce come output un file .XML e un file .HTML in cui sono riportate le informazioni relative ai cloni trovati.

Il processo di rilevamento di NiCad prevede tre fasi principali: l'analisi, la normalizzazione e il confronto.

Nella prima fase vengono analizzati i file sorgenti per estrarre tutti i frammenti di codice secondo una predefinita granularità, come ad esempio funzioni o blocchi dove ciascun frammento estratto è un "potenziale clone".

Nella seconda fase, i frammenti di codice estratti possono essere normalizzati, filtrati o astratti prima del confronto.

Nella fase di confronto, i frammenti sono confrontati a livello lineare usando un algoritmo LCS (Longest Common Subsequence) per rilevare frammenti di codice simili.

Il confronto è parametrizzato rispetto ad un valore soglia che consente il rilevamento di cloni near-miss. Per esempio, una soglia di 0.0 rileva solo cloni esatti, 0.1 rileva quelli che possono differire fino al 10% di linee normalizzate, 0,2 fino al 20%, e così via.

A differenza di molti altri programmi, NiCad crea direttamente classi di cloni che contengono cloni dello stesso tipo che si riferiscono allo stesso frammento di codice in file sorgenti diversi.

2.2 Applicazione di NiCad

NiCad è stato utilizzato per rilevare i cloni di due sistemi: Dnsjava e Tika. NiCad è stato avviato dal terminale con il comando “nicad5 functions java” seguito dal percorso della cartella contenente i file sorgenti.

La figura 2.1 riporta un esempio del report generato da NiCad circa l’analisi dei cloni in un sistema software analizzato (il sistema Dnsjava) con la soglia pari al 30%, la granualità uguale a quella di una funzione, linguaggio java e intervallo di linee consecutive da 10 a 2500. Il report indica che sono state analizzate 2118 funzioni e trovate 72 coppie di cloni distribuite in 34 classi.

```
NiCad Clone Detector v5.0 (1.10.18)

config=./config/type3-1-report.cfg
system=examples/dnsjava-2.1.6
threshold=0.30
granularity=functions
language=java
transform=none
rename=none
filter=none
abstract=none
normalize=none
cluster=yes
report=yes
include=
exclude=

mer 03 apr 2019 09:22:39

Using previously extracted functions from java files in examples/dnsjava-2.1.6

Extracted 2118 functions

Finding clones between 10 and 2500 lines at UPI threshold 0.30

real    0m0,545s
user    0m0,076s
sys     0m0,182s

Found 72 clone pairs

Clustering clone pairs into classes

real    0m0,471s
user    0m0,092s
sys     0m0,185s

Clustered clone pairs into 34 classes

Getting original sources for clones

real    0m1,077s
user    0m0,061s
sys     0m0,246s

Making HTML reports

real    0m0,392s
user    0m0,060s
sys     0m0,106s

Done.

Detailed log in examples/dnsjava-2.1.6_functions-clones-2019-04-12-09:22:39.log
Results in examples/dnsjava-2.1.6_functions-clones

mer 03 apr 2019 09:22:42
```

Figura 2.1 – Analisi cloni di Dnsjava

La figura 2.2 riporta un esempio del report relativo a due cloni individuati da NiCad nel sistema Dnsjava indicando anche che essi sono formati da 10 linee e con una

similarità del 70%. Sono anche riportate i due frammenti formanti la coppia di cloni individuati.

Clone class 3, 2 fragments, nominal size 10 lines, similarity 70%

Lines 170 - 180 of examples/dnsjava-2.1.6/org/xbill/DNS/Lookup.java

```
public static synchronized void
setDefaultSearchPath(String [] domains) throws TextParseException {
        if (domains == null) {
                defaultSearchPath = null;
                return;
        }
        Name [] newdomains = new Name[domains.length];
        for (int i = 0; i < domains.length; i++)
                newdomains[i] = Name.fromString(domains[i], Name.root);
        defaultSearchPath = newdomains;
}
```

Lines 329 - 339 of examples/dnsjava-2.1.6/org/xbill/DNS/Lookup.java

```
public void
setSearchPath(String [] domains) throws TextParseException {
        if (domains == null) {
                this.searchPath = null;
                return;
        }
        Name [] newdomains = new Name[domains.length];
        for (int i = 0; i < domains.length; i++)
                newdomains[i] = Name.fromString(domains[i], Name.root);
        this.searchPath = newdomains;
}
```

Figura 2.2 – Esempio di cloni in Dnsjava

Il tool NiCad produce anche report in formato .xml indicanti informazioni circa i cloni identificati. Nella Tabella 2.1 è riportato un estratto da tale file .xml, relativa all'analisi del sistema Dnsjava, con la descrizione delle varie parti.

Il codice in formato .xml riportato in seguito è un esempio di output di più cloni analizzati in Dnsjava suddivisi in classi di cloni caratterizzate dal numero di linee dei cloni e dalla loro similarità

<class classid="1" nclones="2" nlines="22" similarity="77"> <source file="examples/Dnsjava/org/xbill/DNS/Zone.java" startline="305" endline="328" pcid="136"></source> …. </class>
• classid: indica la classe del clone • nclones: il numero di cloni costituenti quella classe • nlines: il numero di linee di codice formanti i cloni della classe • similarity: la percentuale di similarità dei cloni nella classe • file: nome del file sorgente in cui è stato rilevato il clone • startline: numero di linea di codice da dove inizia il clone nel file sorgente • endline: numero di linea di codice in cui termina il clone nel file sorgente • pcid: codice identificatore del clone

Tabella 2.1: Estratto del file xml prodotto da NiCad e relativa legenda

Capitolo 3: Il sistema di controllo versione Git e di hosting GitHub

3.1 Principali caratteristiche di Git

Git [Git] è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. La sua progettazione si ispirò a strumenti (allora proprietari) analoghi come BitKeeper e Monotone.

Git nacque per essere uno strumento per facilitare lo sviluppo del kernel Linux ed è diventato uno degli strumenti di controllo versione più diffusi.

Il progetto di Git è la sintesi dell'esperienza di Torvalds nel mantenere un grande progetto di sviluppo distribuito, della sua intima conoscenza delle prestazioni del file system, e di un bisogno urgente di produrre un sistema soddisfacente in poco tempo. Queste influenze hanno condotto alle seguenti scelte implementative:

- *Forte supporto allo sviluppo non lineare*: Git supporta diramazione e fusione (branching and merging) rapide e comode, e comprende strumenti specifici per visualizzare e navigare una cronologia di sviluppo non lineare. Un'assunzione centrale in Git è che una modifica verrà fusa più spesso di quanto sia scritta, dato che viene passata per le mani di vari revisori.

- *Sviluppo distribuito*: Git dà a ogni sviluppatore una copia locale dell'intera cronologia di sviluppo, e le modifiche vengono copiate da un tale repository a un altro. Queste modifiche vengono importate come diramazioni aggiuntive di sviluppo, e possono essere fuse allo stesso modo di una diramazione sviluppata localmente.

- *Gestione efficiente di grandi progetti*: Git è molto veloce e scalabile. È tipicamente un ordine di grandezza più veloce degli altri sistemi di controllo versione, e due ordini di grandezza più veloce per alcune operazioni.

- *Autenticazione crittografica della cronologia*: la cronologia di Git viene memorizzata in modo tale che il nome di una revisione particolare (secondo la terminologia Git, una "commit") dipende dalla completa cronologia di sviluppo che conduce a tale commit. Una volta che è stata pubblicata, non è più possibile cambiare le vecchie versioni senza che ciò venga notato.

- *Progettazione del toolkit*: Git è un insieme di programmi di base scritti in linguaggio C e molti script di shell che forniscono comodi incapsulamenti. È facile concatenare i componenti per fare altre cose utili.

- *Strategie di fusione intercambiabili*: come parte della progettazione del suo toolkit, Git ha un modello ben definito di una fusione incompleta, e ha più algoritmi per tentare di completarla. Se tutti gli algoritmi falliscono, tale fallimento viene comunicato all'utente e viene sollecitata una fusione manuale. Pertanto, è facile sperimentare con nuovi algoritmi di fusione.

3.2 Implementazione

Git ha due strutture dati, un *indice* modificabile che mantiene le informazioni sul contenuto della prossima revisione, e un *database di oggetti* a cui si può solo aggiungere e che contiene quattro tipi di oggetti:

- Un oggetto blob è il contenuto di un file. Gli oggetti blob non hanno nome, data, ora, né altri metadati. Git memorizza ogni revisione di un file come un oggetto blob distinto.

- Un oggetto *albero* è l'equivalente di una directory: contiene una lista di nomi di file, ognuno con alcuni bit di tipo e il nome di un oggetto blob o albero che è il file, il link simbolico, o il contenuto di directory. Questo oggetto descrive un'istantanea dell'albero dei sorgenti.

- Un oggetto *commit* (revisione) collega gli oggetti albero in una cronologia. Contiene il nome di un oggetto albero (della directory dei sorgenti di livello più alto),

data e ora, un messaggio di archiviazione (log message), e i nomi di zero o più oggetti di commit genitori. Le relazioni tra i blob si possono trovare esaminando gli oggetti albero e gli oggetti commit.

- Un oggetto *tag* (etichetta) è un contenitore che contiene riferimenti a un altro oggetto, può tenere metadati aggiuntivi riferiti a un altro oggetto. Il suo uso più comune è memorizzare una firma digitale di un oggetto commit corrispondente a un particolare rilascio dei dati gestiti da Git.

Ogni oggetto è identificato da un codice hash SHA-1 del suo contenuto. Git calcola tale codice hash, e usa questo codice come nome dell'oggetto.

L'*indice* è uno strato intermedio che serve da punto di collegamento fra il database di oggetti e l'albero di lavoro.

Il database ha una struttura semplice. L'oggetto viene messo in una directory che corrisponde ai primi due caratteri del suo codice hash; Il resto del codice costituisce il nome del file che contiene tale oggetto.

3.3 GitHub

GitHub [GitHub] è un servizio di hosting per progetti software. Il nome "GitHub" deriva dal fatto che GitHub è una implementazione dello strumento di controllo versione distribuito Git. Il sito è principalmente utilizzato dagli sviluppatori, che caricano il codice sorgente dei loro programmi e lo rendono scaricabile dagli utenti. Questi ultimi possono interagire con lo sviluppatore tramite un sistema di issue tracking, pull request e commenti che permette di migliorare il codice del repository risolvendo bug o aggiungendo funzionalità. Inoltre, GitHub elabora dettagliate pagine che riassumono come gli sviluppatori lavorano sulle varie versioni dei repository.

Nel caso di questo lavoro di tesi, GitHub è stato usato per accedere e scaricare i sistemi utilizzati nell’esperimento per associare i cloni software presenti in essi ai committers. In particolare, i sistemi recuperati da GitHub, sono:

- Dnsjava, è un’implementazione del DNS in java, può essere utilizzata per query, trasferimenti di zona e aggiornamenti dinamici
- AdminBus, è un sistema software per consentire ad un utente l’acquisto di un biglietto o di un abbonamento per la corsa selezionata.
- Tika, è un toolkit per il rilevamento e l'estrazione di metadati e il contenuto di testi strutturati in diversi documenti utilizzando le librerie di parser esistenti.

Capitolo 4: Il tool Sourcetree

4.1 Principali caratteristiche di Sourcetree

Le informazioni relative ai commit e i committer sono state recuperate utilizzando Sourcetree [Sourcetree], un client di Git gratuito che permette di interagire con i repository in modo più semplice ed efficace.

Esso permette di visualizzare e gestire i repository tramite la semplice GUI Git di Sourcetree oppure tramite terminale Git.

Nel caso specifico è stato utilizzato il comando git log tramite terminale Git che mostra i commits eseguiti nel repository in ordine cronologico inverso. In questo modo il commit più recente è il primo ad apparire. Ogni commit è elencato riportando il suo codice SHA-1, il nome e l'e-mail dell'autore (committer), la data di salvataggio e la descrizione del commit. Sono disponibili moltissime opzioni da passare al comando "git log" per vedere esattamente altre informazioni specifiche.

L'opzione utilizzata è "-p", che mostra le differenze introdotte da ciascuna commit rispetto allo stato precedente del file revisionato.

Un'altra opzione interessante è "format", che permette di specificare la formattazione dell'output di log. Questa è specialmente utile quando si genera un output che sarà analizzo da una macchina, come nel caso specifico, perché quest'ultimo non cambierà con gli aggiornamenti di Git.

Di seguito alcuni esempi di formattazione dell'output:

Opzione *Descrizione dell'output*

%H Hash della commit

%h Hash della commit abbreviato

%T Hash dell'albero

%P Hash del genitore

%an Nome dell'autore

%ae e-mail dell'autore

%ad Data di commit dell'autore (il formato rispetta l'opzione --date=)

%ar Data relativa di commit dell'autore

%cn Nome di chi ha fatto la commit (committer, in inglese)

%ce e-mail di chi ha fatto la commit

%cd Data della commit

%s Oggetto

E' stata infine aggiunta l'opzione "--date=iso" per ottenere come output la data nel formato *ISO 8601*, uno standard internazionale per la rappresentazione di date e orari, ovvero "yyy-MM-dd HH:mm:ss".

Il risultato è stato salvato in file di testo aggiungendo alla fine l'opzione ">" seguito dal nome del file.

Nello specifico è stata utilizzata la seguente istruzione per ogni versione analizzata:

```
git log -p --date=iso --after="dateFirstCommit" --before="dateLastCommit"

--pretty=format:"Commit:%n%H%n%cn%n%ce%n%cd%n%s" > nomeFile
```

La figura 4.1 riporta un esempio di commit del progetto Dnsjava, in cui sono indicati:

- il codice identificatore del commit
- il nome e l'indirizzo di e-mail del committer
- la data in cui è stato effettuato il commit
- la descrizione/motivazione del commit
- il file sorgente oggetto del commit
- intervallo di modica all'interno del file

```
Commit:
849bde970e2349d828147453807898ed761cc2c7
bwelling
bwelling@c76caeb1-94fd-44dd-870f-0c9d92034fc1
2016-08-14 17:21:17 +0000
Fix typo.
diff --git a/org/xbill/DNS/OPENPGPKEYRecord.java b/org/xbill/DNS/OPENPGPKEYRecord.java
index 696eda7..359fb67 100644
--- a/org/xbill/DNS/OPENPGPKEYRecord.java
+++ b/org/xbill/DNS/OPENPGPKEYRecord.java
@@ -33,7 +33,7 @@ getObject() {
 public
 OPENPGPKEYRecord(Name name, int dclass, long ttl, byte [] cert)
 {
-   super(name, Type.CERT, dclass, ttl);
+   super(name, Type.OPENPGPKEY, dclass, ttl);
    this.cert = cert;
 }
```

Figura 4.1 – Esempio di report di un commit relativo al progetto Dnsjava

Capitolo 5: Il tool Clo_tter esteso per l'analisi dell'evoluzione dei cloni in release successive di un sistema sw

5.1 Architettura del tool Clo_tter

E' stata realizzata una tool-chain, di cui il tool Clo_tter è l'elemento finale, con lo scopo di:

1. accedere alle informazioni di progetti open-source ospitati in GitHub, selezionare uno di tali progetti, scaricarne i file con il codice sorgente di una delle versioni del file e quelli relativi ai commit effettuati e memorizzare tali file in un repository locale;
2. analizzare il codice sorgente per l'individuazione in esso di cloni software;
3. analizzare il file con le informazioni relative ai commit per filtrare ed estrarre quelle di interesse;
4. analizzare le informazioni ottenute nei due punti precedenti in modo da riuscire ad associare ciascun clone al committer che ha effettuato un commit su quel frammento di codice clonato; tale committer è candidato ad essere considerato come il responsabile dell'introduzione del clone. Tali informazioni vanno registrate in una base di dati per permettere le elaborazioni al punto successivo;
5. analizzare l'evoluzione dei cloni o di una classe di cloni tra le successive release del sistema per capire se un clone continua a persistere nella nuova versione, se è aggiunto un nuovo clone a quelli preesistenti di una certa classe di cloni, se è aggiunto un clone completamente nuovo non esistente nella release precedente andando a formare una nuova classe di cloni.

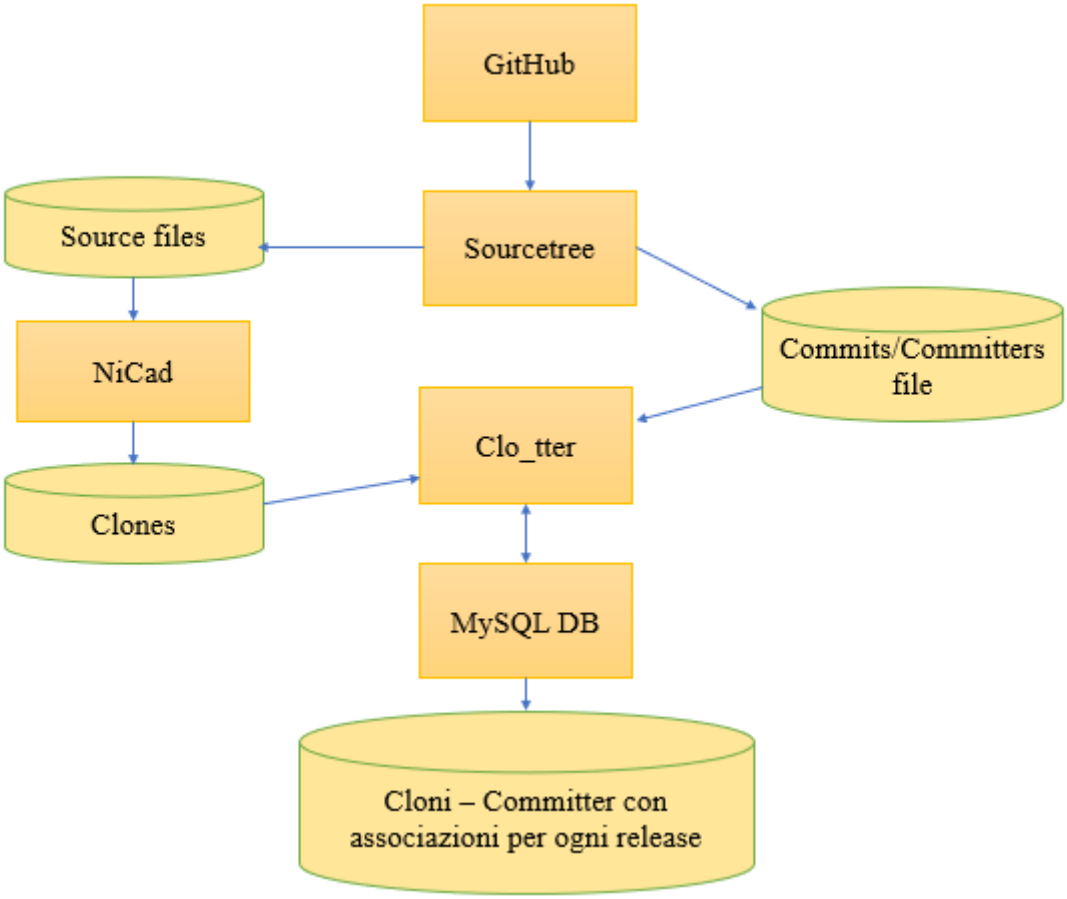


Figura 5.1 – Architettura generale

5.2 Descrizione dei principali casi d'uso

In figura 5.2 si riporta un UML Use Case Diagram utile a descrivere le funzioni offerte dal tool Clo_tter in relazione all'attore che interagisce con il sistema.

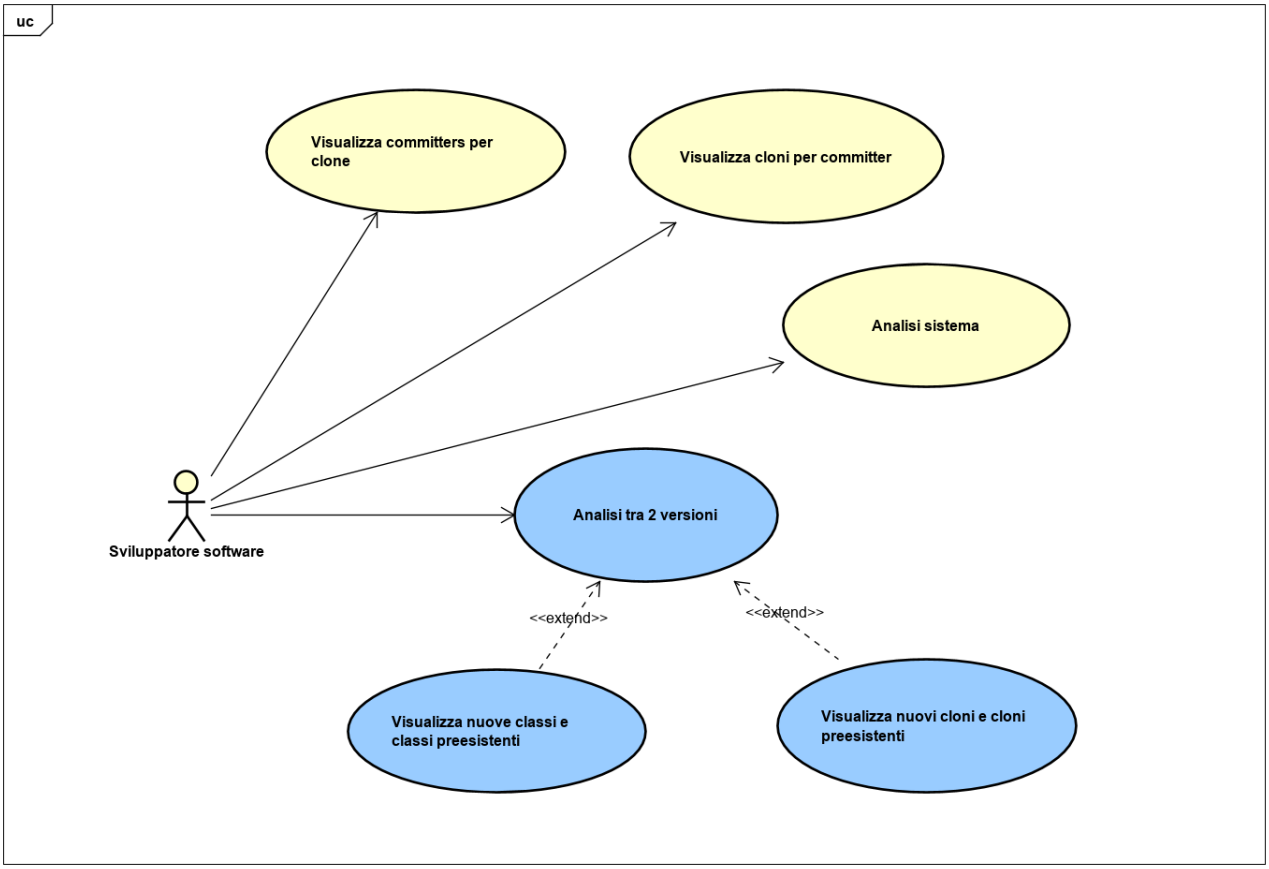


Figura 5.2 – UML Use Case Diagram

Vengono descritti i principali casi d'uso oggetti del presente lavoro di tesi:

- UC1 – Analisi tra 2 versioni del sistema
- UC2 – Visualizza lista nuovi cloni e lista cloni preesistenti
- UC3 – Visualizza lista nuove classi e lista classi preesistenti

Nome: Analisi tra 2 versioni [UC1]

Attore: Sviluppatore software

Flusso eventi [individuazione delle nuove classi di cloni e dei nuovi cloni nella versione successiva]

1. L'utente seleziona due versioni consecutive di un sistema
2. Clo_tter interroga la base di dati per ottenere i cloni e le classi di cloni delle due versioni
3. Clotter interroga la base di dati per ottenere le associazioni cloni-committer per la versione successiva
4. Clotter elabora le associazioni per individuare i nuovi cloni e quelli preesistenti
5. **Extension point 1 [UC2]**: Viene visualizzata la lista dei nuovi cloni e di quelli preesistenti
6. Clotter analizza ogni classe della versione successiva per individuare quali sono nuove e quali preesistenti.
7. **Extension point 2 [UC3]**: Viene visualizzata la lista delle nuove classi e di quelle preesistenti

Flusso Alternativo A1 [interruzione esecuzione Clo_tter]

*. Clo_tter smette di funzionare per problemi di elaborazione o relativi al collegamento con il DB.

Pre-condizioni: sono stati analizzate le due versioni e popolato la relativa base di dati

Input: 2 versioni consecutive

Post-condizioni: il contenuto della base di dati resta invariato

Output: lista nuovi cloni e cloni preesistenti; lista nuove classi e classi preesistenti

Nome: Visualizza lista nuovi cloni e lista cloni preesistenti [UC2]
Attore: Sviluppatore software
Flusso eventi [visualizzazione nuovi cloni e cloni preesistenti di una versione]
1. Il sistema visualizza tutti i ‘nuovi cloni’ associati ai committer e i ‘cloni preesistenti’
Flusso Alternativo A1 [interruzione esecuzione Clo_tter]
*. Clo_tter smette di funzionare per problemi di elaborazione o relativi al collegamento con il DB.
Pre-condizioni: sono disponibili i nuovi cloni e i cloni preesistenti
Input: lista nuovi cloni e lista cloni preesistenti
Post-condizioni: il contenuto della base di dati resta invariato
Output: visualizzazione delle liste in input

Nome: Visualizzazione lista nuove classi di cloni e lista classi preesistenti [UC3]
Attore: Sviluppatore software
Flusso eventi [visualizzazione nuove classi e classi preesistenti di una versione]
1. Il sistema visualizza tutte le nuove classi di cloni e quelle preesistenti
Flusso alternativo A1 [interruzione esecuzione Clo_tter]
*. Clo_tter smette di funzionare per problemi di elaborazione o relativi al collegamento con il DB.
Pre-condizioni: sono disponibili le nuove classi di cloni e le classi preesistenti
Input: lista nuove classi di cloni e classi preesistenti
Post-condizioni: il contenuto della base di dati resta invariato
Output: visualizzazione delle liste in input

Si rimanda al paragrafo 5.3.3 per una più dettagliata descrizione dei processi attuati da Clo_tter in riferimento all’evoluzione dei cloni tra due versioni successive.

5.3 Il design del tool Clo_tter

5.3.1 Design architetturale

Dovendo interagire con l'utente, l'architettura di Clo_tter è incentrata sul pattern **MVC**: (Model, View, Controller) che è adatto ad applicazioni interattive e fornisce un metodo efficiente per disaccoppiare la componente HCI con il resto dell'applicazione.

In particolare, esso divide il sottosistema in tre componenti che gestiscono indipendentemente input, elaborazione, output:

- Model: rappresenta il modello dei dati di interesse per l'applicazione;
- View (GUI): fornisce una rappresentazione grafica del model;
- Controller: definisce la logica di controllo e le funzionalità applicative.

5.3.2 Design di Clo_tter

La figura 5.3 riporta lo UML Class Diagram di Clo_tter, riportante le principali classi che lo costituiscono e le loro relazioni.

Per motivi di spazio nella figura non sono riportati tutti i dettagli progettuali, quali quelli relativi ai Design Pattern utilizzati.

Uno è il **Singleton,** utilizzato per le classi di gestione ovvero Controller, GestoreCommits, GestoreClones, GestoreDB per avere una e una sola istanza di esse.

Un atro è il **Facade,** utilizzato nella classe GestoreDB insieme al package java.sql utilizzando un'interfaccia di più alto livello rendendo il sottosistema relativo all'accesso al database, più semplice.

Nel seguito è riportata una sintetica descrizione delle classi componenti la parte Model del pattern MVC il class diagram della figura 5.4:

- Commit: rappresenta la revisione effettuata sul sistema software analizzato; è caratterizzata dai seguenti attributi: identificativo, data, descrizione e committer;
- Committer: rappresenta i committer che hanno agito sul sistema software analizzato; è caratterizzata dai seguenti attributi: nome ed e-mail;
- Clone: rappresenta i cloni identificati nel sistema analizzato; è caratterizzata dai seguenti attributi: identificativo, file, prima linea del clone, ultima linea del clone, identificativo della classe del clone;
- Classe del clone: raggruppa tutte i frammenti di codice simili tra loro per un'elevata soglia di similarità; è caratterizzata dai seguenti attributi: identificativo, numero di cloni, numero di linee e similarità.
- File: rappresenta un file sorgente del sistema analizzato, oggetto delle operazioni di change in un commit; è caratterizzata dal nome del file;
- Change: rappresenta le varie operazioni di modifica che sono eseguite in un commit; è caratterizzata da un identificativo, identificativo del relativo Commit e del file cui è applicato;
- Range: rappresenta l'intervallo delle linee di codice di un file sorgente a cui è stato applicato un change; è caratterizzata da un identificativo, identificativo del relativo Change, la prima riga e l'intervallo di modifica

Un Commit può essere formato da diversi *changes* a più file sorgenti differenti, mentre il *change* può essere formato da più *ranges* riferiti allo stesso file.

Inoltre, il class diagram riporta le classi costituenti la parte Controller dello MVC.

La classe GestoreDB ha la responsabilità di interfacciare l'applicazione con il database, permettendo di leggere o scrivere dati persistenti in memoria.

La classe Controller ha il compito di determinare il modo in cui l'applicazione risponde agli input dell'utente.

La classe GestoreCommits ha la responsabilità di leggere i commits, committers e files dal report generato da Sourcetree e di istanziare le relative liste.

La classe GestoreClones ha la responsabilità della lettura dei cloni e delle classi dei cloni dal report generato da NiCad e di istanziare le relative liste.

Il class diagram riporta anche un package GUI, che rappresenta la parte view dello MVC, raggruppante tutte le classi implementate per realizzare l'interfaccia utente.

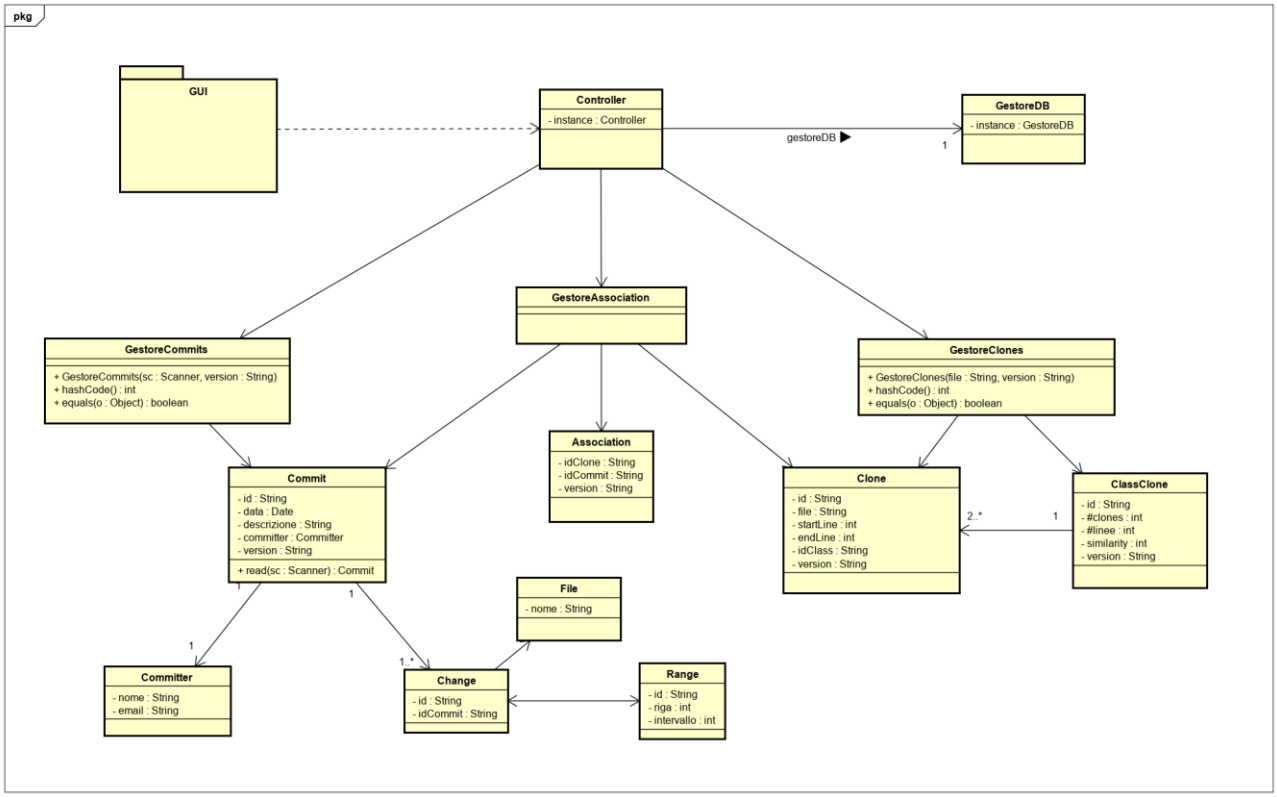


Figura 5.3 – UML Class Diagram

5.3.3 Sequence Diagram

Gli scenari precedentemente illustrati in figura 5.2 con i diagrammi di casi d'uso, vengono ora descritti, in figura 5.4, utilizzando un UML Sequence Diagram che dettagliatamente riporta la sequenza di azioni svolte dal sistema:

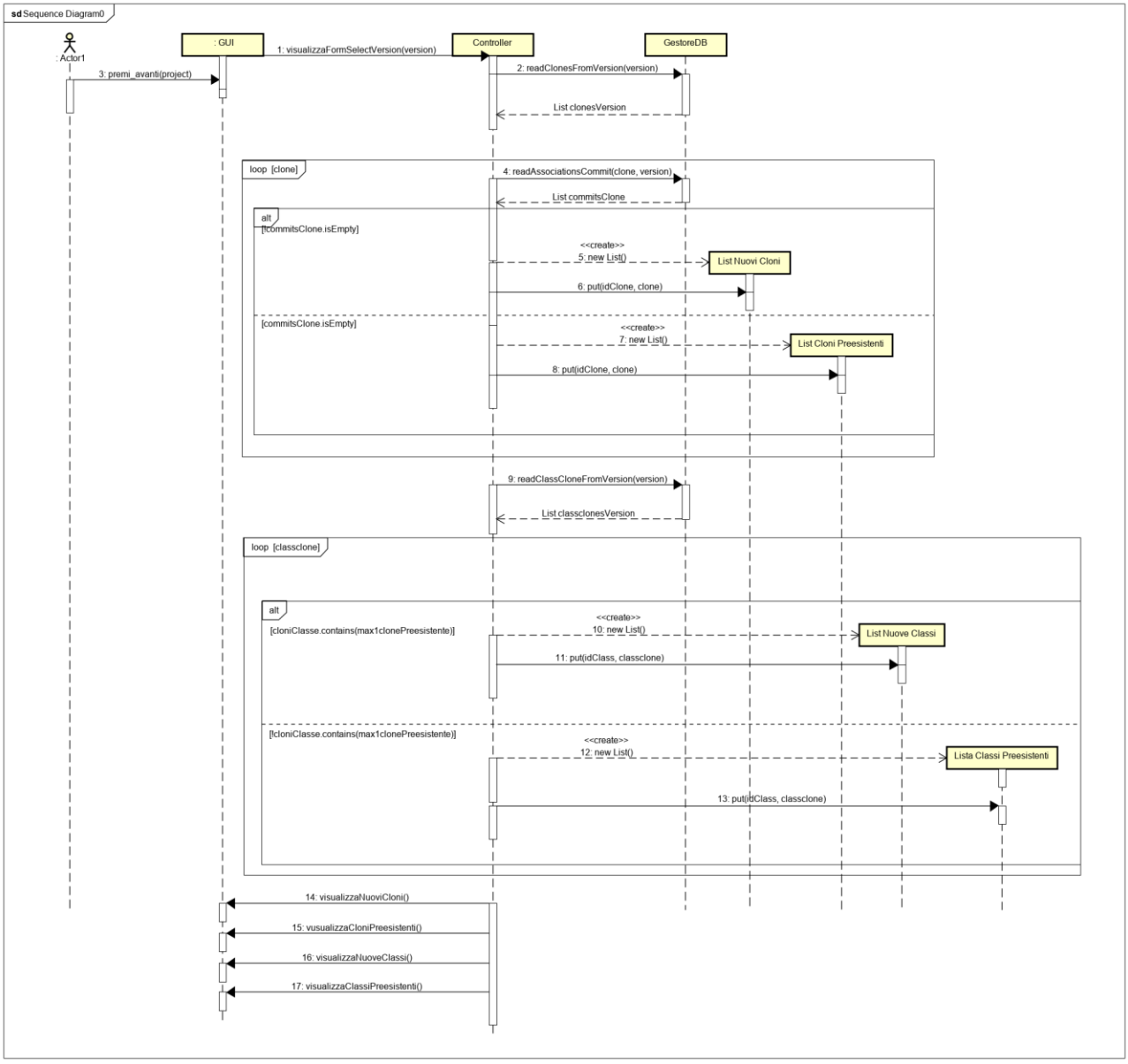


Figura 5.4 – UML Sequence Diagram

5.4 Implementazione

5.4.3 Interfacciamento database - applicazione

Il database è gestito tramite il DBMS *MySQL*.

L'interfacciamento con l'applicativo Java è eseguito attraverso il connettore *JDBC*.

In particolare, è la Singleton class **GestoreDB** che provvede alla comunicazione con il database.

5.4.1 Component Diagram

La figura 5.5 riporta un component diagram modellante i principali componenti di Clo_tter: Clo_tter.management, relativo all'insieme delle classi di gestione; Clo_tter.baseClass, relativo all'insieme delle classi base e Clo_tter.gui, relativo al l'insieme delle classi realizzanti l'interfaccia utente. Clo_tter può connettersi direttamente al DBMS MySQL attraverso l'utilizzo di JDBC API, gestore di driver che permette ad un driver di terze parti di connettersi a un DB specifico.

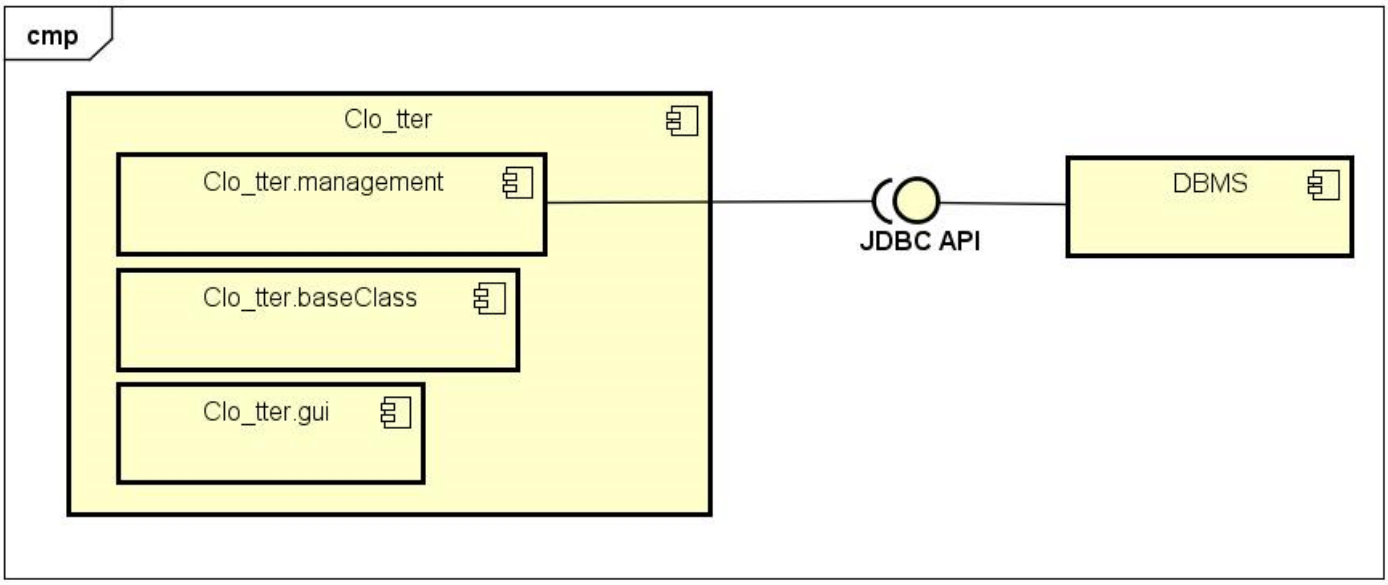


Figura 5.5 – Component Diagram

5.4.2 Schema logico DB

Nel seguito è riportato lo schema logico delle tabelle costituenti il data base di Clo_tter:

Commits= {id, emailCommitter, data, descrizione, version}

Committers= {email, nome}

Changes= {id, idCommit, file}

Ranges= {id, idChange, riga. intervallo}

Files={nomeFile}

Clones= {file, startline, endline, pcid, classid, version}

ClassClones= {id, clones, righe, similarity, version}

Associations= {idClone, idCommit, version}

La figura 5.6 riporta uno screen shot relativo al contenuto della tabella Clones del progetto Dnsjava.

file	startline	endline	pcid	classid	version
org/xbill/DNS/Lookup.java	170	180	520	3	2.1.6
org/xbill/DNS/Lookup.java	329	339	530	3	2.1.6
org/xbill/DNS/Lookup.java	170	180	531	3	2.1.7
org/xbill/DNS/Lookup.java	339	349	542	3	2.1.7
org/xbill/DNS/Lookup.java	170	180	543	3	2.1.8
org/xbill/DNS/Lookup.java	339	349	554	3	2.1.8
org/xbill/DNS/NSEC3PARAMRecord.java	92	108	725	4	2.1.6
org/xbill/DNS/NSEC3PARAMRecord.java	92	108	737	4	2.1.7
org/xbill/DNS/NSEC3PARAMRecord.java	92	108	750	4	2.1.8
org/xbill/DNS/NSEC3Record.java	137	155	739	4	2.1.6
org/xbill/DNS/NSEC3Record.java	137	155	751	4	2.1.7
org/xbill/DNS/NSEC3Record.java	137	155	764	4	2.1.8
org/xbill/DNS/SMIMEARecord.java	112	124	1065	5	2.1.8
org/xbill/DNS/TLSARecord.java	110	122	1157	5	2.1.8
org/xbill/DNS/Zone.java	305	328	1352	1	2.1.6
org/xbill/DNS/Zone.java	305	328	1376	1	2.1.7
org/xbill/DNS/Zone.java	305	328	1405	1	2.1.8

Figura 5.6 – Esempio tabella dei cloni

5.5 Descrizione del processo per l'analisi dell'evoluzione dei cloni in release successive di un sistema sw

Nel seguito sono riportati i metodi definiti per attuare l'analisi di come i cloni software presenti in un sistema evolvono nelle release successive. L'obiettivo, come detto in precedenza, è di verificare se un clone continua a persistere nelle release successive, se sono introdotti nuovi cloni non presenti nella release precedente, se sono introdotti nuovi cloni di uno già esistente in precedenza. Inoltre i cloni individuati sono associati ai committer che li hanno introdotti e, quindi, si vuole anche analizzare se vi sono commiter che sono maggiormente propensi ad introdurre cloni nelle varie release.

La figura 5.7 riporta un UML activity diagram modellante l'intero processo realizzato tramite la tool-chain descritta.

Dopo il download dei file sorgenti di release successive di un sistema sw da GitHub, queste vengono analizzate (con NiCad) per individuare in ciascuna di esse i cloni, mentre per associare a ciascun di essi il relativo committer si è utilizzato il tool Clo_tter andando a formare, utilizzando anche le informazioni estratte dal tool Sourcetree, le associazioni tra i cloni e i commit/committer per ciascuna versione.

È poi effettuata l'analisi delle differenze tra due successive versioni relativamente ai cloni individuati in ciascuna di esse con l'obbiettivo di individuare i nuovi cloni rispetto a quelli preesistenti che possano formare nuove classi di cloni.

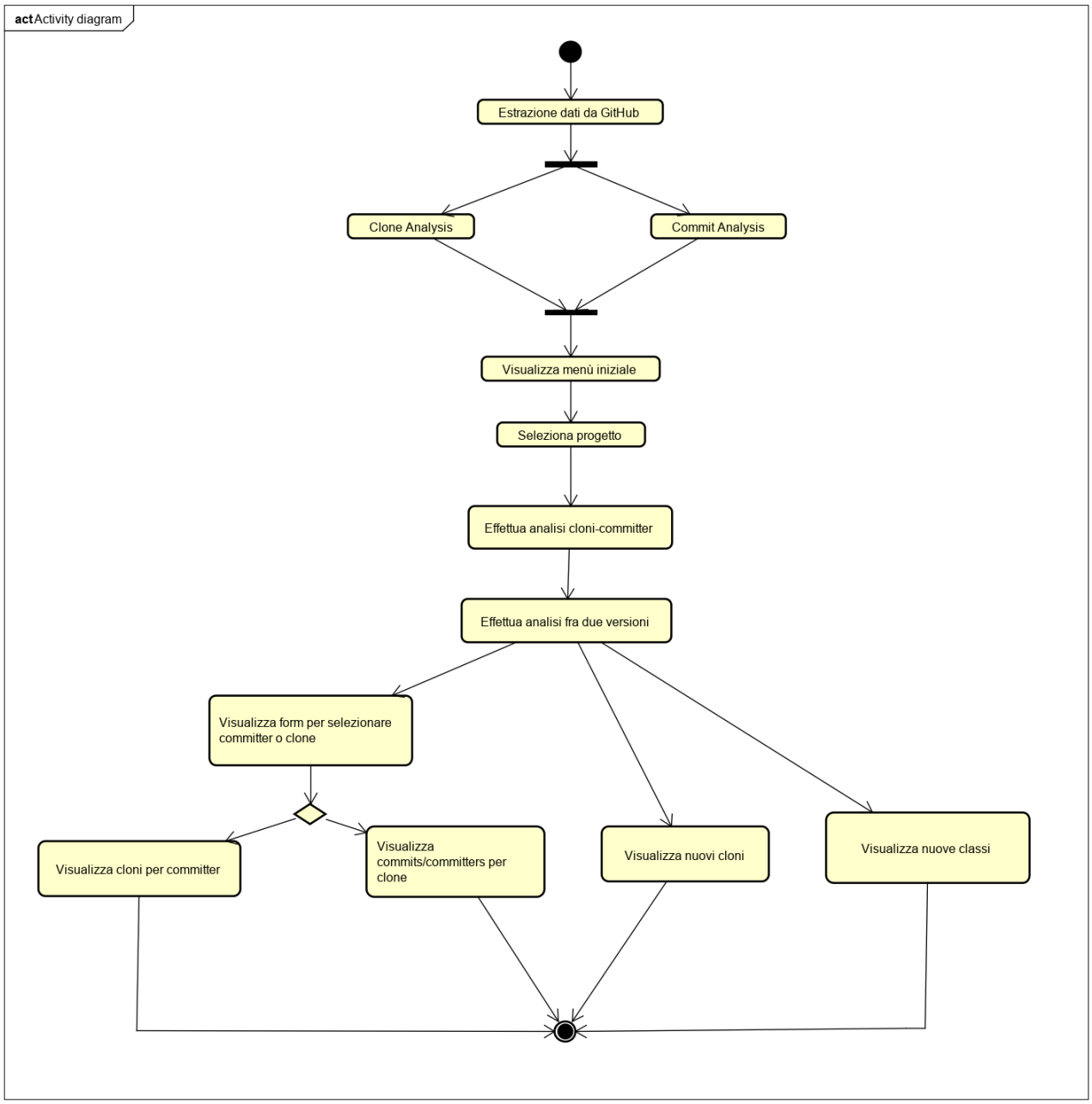


Figura 5.7 – Activity Diagram

In figura 5.8 si descrive con un UML activity diagram il processo di individuazione dei cloni, classi di cloni e commits/committers associati alla singola versione per ciascun sistema open source. Per ottenere informazioni riguardanti l'evoluzione dei cloni nelle varie release è necessario che il DB sia popolato adeguatamente per effettuare le interrogazioni riguardanti ciascuna versione.

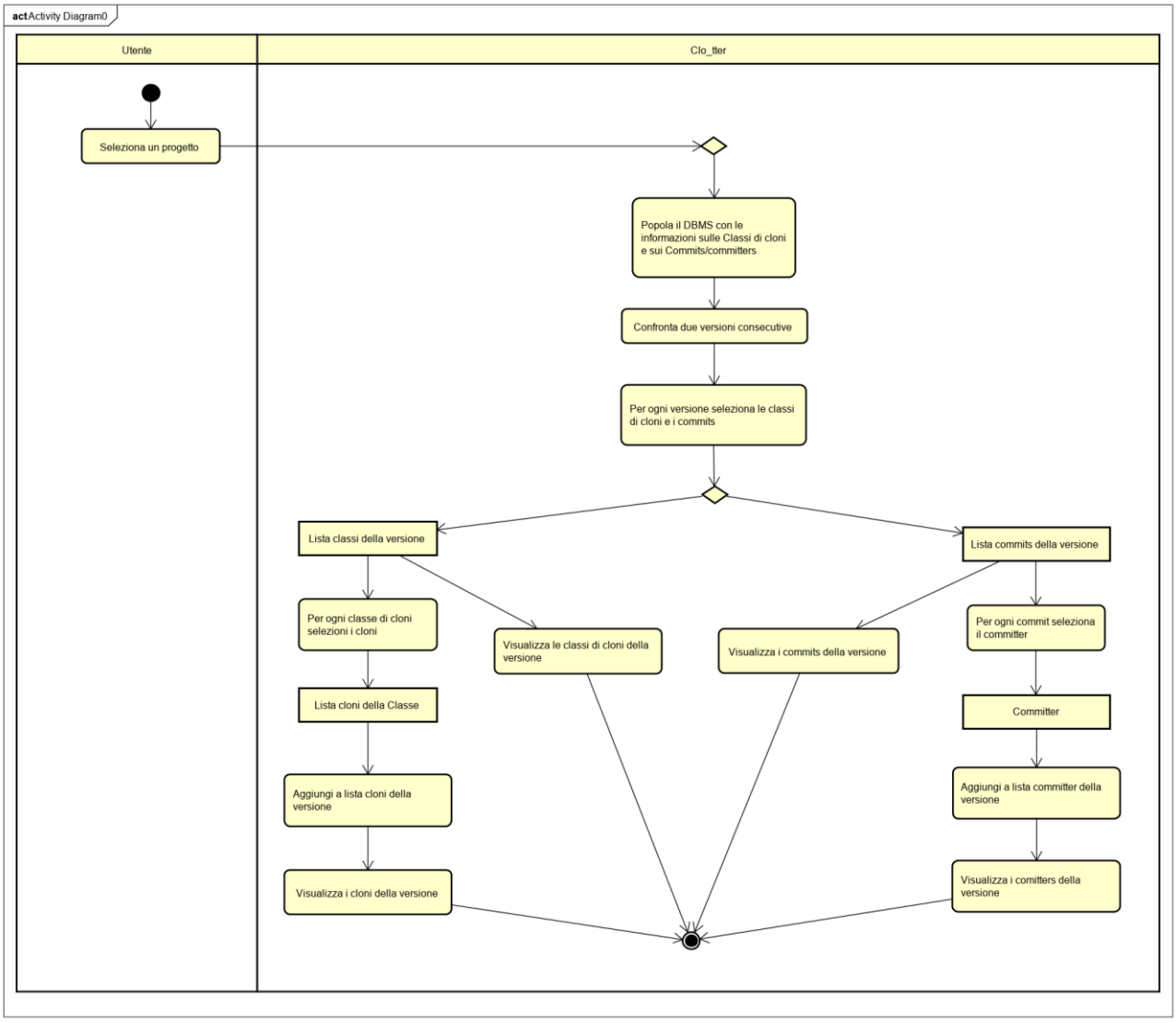


Figura 5.8 – UML activity diagram relativo all'analisi tra più versioni

In ogni versione del sistema si vuole individuare i cloni e le classi di cloni aggiunte e i committer associati che hanno contribuito all'introduzione di codice clonato.

In figura 5.9 si descrive il processo di individuazione dei nuovi cloni e delle nuove classi di cloni. Si considera una versione successiva per la quale si vuole individuare i nuovi cloni e le nuove classi di cloni aggiunte nella release:

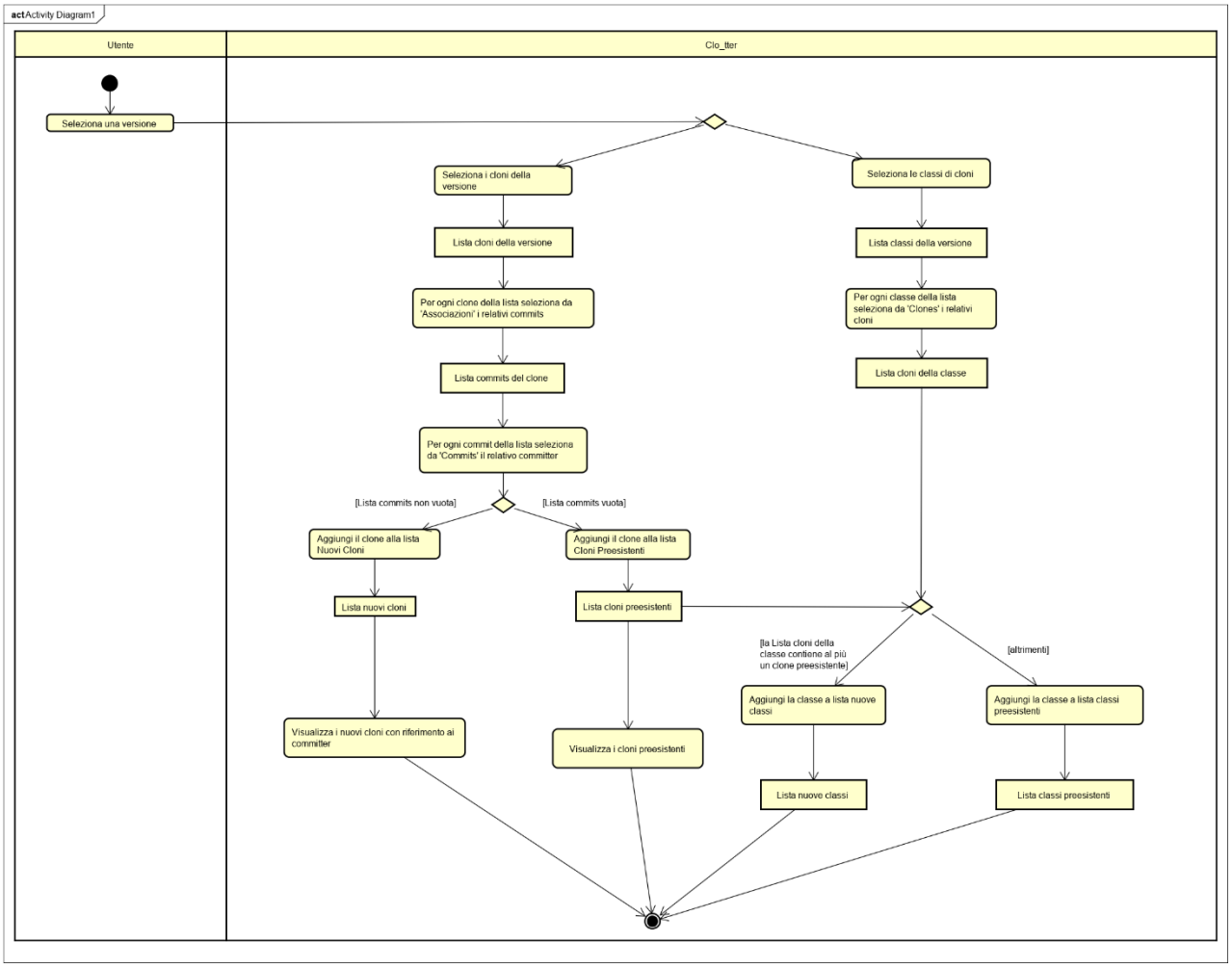


Figura 5.9 UML activity diagram relativo all'individuazione di nuovi cloni/classi di cloni

Il seguente esempio descrive il processo per l'individuazione di nuovi cloni e classi di cloni, associandoli ai relativi committer.

- La tabella 1 riporta i cloni totali e le classi totali individuali in due successive versioni di un ipotetico sistema analizzato

Cloni/Classi		
ID clone	*ID classe*	*Versione*
1	1	1.0
2	2	1.0
1	1	2.0
2	2	2.0
3	2	2.0
4	1	2.0
5	2	2.0
6	2	2.0

Tabella 1 – Associazione cloni-classi di cloni

- Si selezionano i cloni e le classi di cloni per la versione 2.0:

Cloni	
ID clone	*Versione*
1	2.0
2	2.0
3	2.0
4	2.0
5	2.0
6	2.0

Classi di Cloni	
ID classe	*Versione*
1	2.0
2	2.0

- Per ogni clone della versione 2.0 sono ricavate le associazioni con i relativi commit/committer in base al contenuto della seguente tabella 2:

Cloni – Commit/Committer			
ID clone	*ID commit*	*ID committer*	*Versione*
1	X	234	1.0
2	Y	123	1.0
1	A	123	2.0
2	A	123	2.0
2	B	234	2.0
4	A	123	2.0
4	C	234	2.0
6	B	234	2.0

Tabella 2 – Associazioni cloni-commit/committer

Le seguenti tabelle riportano per ogni clone le associazioni con il relativo committer per la versione successiva considerata:

Clone ID=1 versione=2.0		
ID clone	*ID commit*	*ID committer*
1	A	123

Clone ID=2 versione=2.0		
ID clone	*ID commit*	*ID committer*
2	A	123
2	B	234

Clone ID=3 versione=2.0		
ID clone	*ID commit*	*ID committer*
3	NULL	NULL

Clone ID=4 versione=2.0		
ID clone	*ID commit*	*ID committer*
4	A	123
4	C	234

Clone ID=5 versione=2.0		
ID clone	*ID commit*	*ID committer*
5	NULL	NULL

Clone ID=6 versione=2.0		
ID clone	*ID commit*	*ID committer*
6	B	234

Quando per un clone della versione successiva risulta almeno un'associazione con un commit/committer esso è considerato essere *'Nuovo Clone'*; mentre se non risulta essere associato a nessun commit/committer è considerato essere un clone già esistente nella versione precedente:

Nuovi Cloni	
ID clone	*Versione*
1	2.0
2	2.0
4	2.0
6	2.0

Cloni preesistenti	
ID clone	*Versione*
3	2.0
5	2.0

- Per individuare le nuove classi di cloni, con riferimento alla tabella 1 '*Cloni-Classi*', consideriamo le classi della versione 2.0:

Cloni-Classi	
ID Clone	*ID Classe*
1	1
4	1

Cloni-Classi	
ID Clone	*ID Classe*
2	2
3	2
5	2
6	2

La classe 1 è composta solamente da tutti nuovi cloni, per questo si definisce come *'Nuova Classe'*. Diversamente la classe 2 contiene più di un clone preesistente, quindi viene definita *'Classe preesistente'*.

- In definitiva i nuovi cloni e le nuove classi per la versione 2.0 sono:

Nuovi Cloni	
ID Clone	*versione*
1	2.0
2	2.0
4	2.0

Nuove Classi	
ID Classe	*versione*
2	2.0

5.6 Applicazione dell'approccio

L'approccio definito è stato applicato a due progetti presenti in GitHub:

- Dnsjava
- Tika

Per individuare i cloni nelle varie versioni considerate di tali sistemi, è stato utilizzato NiCad configurato con i seguenti parametri:

dimensione clone min/max = 10-2500 #linee

soglia minima di similarità = 70%

Di ciascun sistema riportiamo le metriche, calcolate con il tool **Understand Scitools** [UND], in termini di classi, funzioni e linee di codice (LOC). Questi valori sono indicativi delle differenze tra i due sistemi:

- Dnsjava
    - v 2.1.6: 270 classi, 2146 funzioni, 24201 LOC
    - v 2.1.7: 275 classi, 2188 funzioni, 24662 LOC
    - v 2.1.8: 279 classi, 2212 funzioni, 24727 LOC
- Tika
    - v 1.0: 567 classi, 2930 funzioni, 36884 LOC
    - v 1.1: 591 classi, 3059 funzioni, 39481 LOC
    - v 1.2: 634 classi, 3371 funzioni, 44662 LOC
    - v 1.3: 640 classi, 3462 funzioni, 45932 LOC
    - v 1.4: 669 classi, 3696 funzioni, 49869 LOC

Il tool Clo_tter per ottenere le informazioni riguardanti cloni/classi di cloni e commit/committer ha estratto dal database creato per ogni sistema i dati utili all'analisi.

Le tabelle dei cloni, classi di cloni e commit includono l'attributo 'versione' necessario per effettuare l'operazione sui dati, mentre i committer vengono individuati dall'associazione con i commit attraverso l'attributo 'e-mail'.

Per il sistema "Dnsjava":

Dnsjava				
Release	# Classi_Cloni	# Cloni	# Commit	# Committer
2.1.6	34	87	1431	2
2.1.7	35	89	15	2
2.1.8	36	91	9	1

Tabella 5.2.1 – Evoluzione dei cloni e delle classi di cloni per Dnsjava

Nel progetto Dnsjava la numerosità delle classi e dei cloni è rimasta pressoché invariata dalla prima versione all'ultima disponibile per l'analisi.

Il tool Clo_tter ha individuato per il sistema "Tika":

Tika				
Release	# Classi_Cloni	# Cloni	# Commit	# Committer
1.0	31	76	954	13
1.1	36	88	117	6
1.2	44	103	120	7
1.3	49	114	106	7
1.4	54	128	166	10

Tabella 5.2.2 – Evoluzione dei cloni e delle classi di cloni per Tika

In questo sistema Clo_tter ha individuato, tra le 5 versioni analizzate, che le classi di cloni sono aumentate di 23 unità nella versione 1.4, per un totale di 52 cloni rispetto alla prima versione.

Si può osservare che, eccetto per la transizione dalla versione 1.1 alla 1.2 per cui si è avuto un aumento di 8 classi, le classi di cloni aumentano di 5 unità ad ogni transizione di versione. Una considerazione analoga può essere fatta anche per il numero di cloni che in media aumentano di 13 unità nelle varie transizioni, con un massimo per quella tra la 1.1 e 1.2 dove i cloni aumentano di 15 unità.

Sembra non esserci una correlazione tra il numero di committ ed il numero di cloni presenti in ciascuna versione.

Le differenze tra i due sistemi sono significative anche in termini di committer che hanno contribuito all'evoluzione dei cloni nelle varie versioni.

Per questo, soprattutto nei sistemi che presentano più di un committer, conoscere i cloni associati a ciascun di esso può fornire aspetti interessanti all'analisi complessiva.

Il metodo applicato consiste nell'interrogare il database sulle associazioni cloni-commit presenti nella tabella 'Associazioni' selezionando i commit legati al committer scelto.

In Clo_tter il metodo 'readAssociationsClones(String email)' ha il compito di interrogare il DB e salvare in una struttura dati di tipo HashMap i cloni associati al committer selezionato.

La query SQL che effettua l'interrogazione è la seguente:

```
SELECT * FROM associations as a, clones as cl, commits as co, committers as c WHERE
c.email='email' AND a.idclone=cl.pcid AND a.idcommit=co.id  AND
co.email=c.email;
```

L’analisi di Clo_tter ha prodotto i seguenti risultati:

Per Dnsjava al committer ‘bwelling’ con id: 1594128718 Clo_tter ha associato 90 cloni, distribuiti in tutte le versioni esaminate.

Per Tika la tabella 5.2.3 mostra i committer associati ai cloni da essi introdotti rispetto a tutte le release esaminate.

Tra i committer elencati i più propensi all’introduzione di nuovi cloni nel sistema complessivo sono:

- Nick Burch , id: 1125603941, 55 cloni
- Jukka Zitting, id: 1445070098, 52 cloni
- Michael McCandless, id: 2013372900, 17 cloni
- Ray Gauss II, id: 1125603941, 17 cloni

ID Committer	Nome	# Cloni
1125603941	Ray Gauss II	17
476781759	Nick Burch	55
236547980	Hong-Thai Nguyen	1
2013372900	Michael McCandless	17
675030415	Antoni Mylka	1
1587228132	Chris Mattmann	7
400370190	Tim Allison	6
978657549	David Meikle	1
689209543	Maxim Valyanskiy	4
1264350371	Sergey Beryozkin	1
1589042715	Oleg Tikhonov	4
1234144495	Kenneth William Krugler	3
1445070098	Jukka Zitting	52

Tabella 5.2.3 – Committer candidati all’introduzione dei cloni per Tika

Nel seguito sono riportati i risultati ottenuti secondo l'approccio utilizzato per l'individuazione di nuovi cloni in ciascuna versione dei sistemi analizzati.

In Clo_tter il metodo 'readAssociationsCommitsVersion' permette di interrogare il database per ogni clone della versione successiva.

Per Dnsjava nella versione 2.1.7 Clo_tter ha individuato 2 cloni in più rispetto alla versione precedente, ugualmente per la versione 2.1.8:

Dnsjava v.2.1.7					
ID Clone	*ID Classe*	*Start*	*End*	*Committer*	*File*
2036	35	25	49	bwelling	tests/org/xbill/DNS/TSIGTest.java
2037	35	51	78	bwelling	tests/org/xbill/DNS/TSIGTest.java

La classe con id 35 è formata proprio dai *'nuovi cloni'* 2036 e 2037 quindi è una *'nuova classe'*.

Dnsjava v.2.1.8					
ID Clone	*ID Classe*	*Start*	*End*	*Committer*	*File*
1065	5	112	124	bwelling	org/xbill/DNS/SMIMEARecord.java
1157	5	110	122	bwelling	org/xbill/DNS/TLSARecord.java

La classe con id 5 è formata dai *'nuovi cloni'* 1065 e 1157, confermando che anche questa è una *'nuova classe'*.

Per Tika si riportano per brevità solo i risultati riguardanti le ultime due versioni:

Tika v.1.3 - 15 nuovi cloni				
ID Clone	*ID Classe*	*Start*	*End*	*Committer*
3091	36	55	124	476781759
37	1	741	758	2013372900
3046	33	273	311	476781759
3112	36	273	311	2013372900
3121	41	232	265	476781759
3076	39	883	897	2013372900
3095	38	188	218	1125603941
3194	36	228	285	2013372900
3072	38	772	804	1125603941
3119	41	162	191	476781759
3118	41	127	156	476781759
234	3	513	539	1125603941
3113	39	262	276	476781759
1955	21	48	64	1125603941
2934	29	67	91	476781759

Tika v.1.4 - 16 nuovi cloni				
ID Clone	*ID Classe*	*Start*	*End*	*Committer*
3255	41	207	237	1445070098
3387	48	128	152	1445070098
3254	40	182	201	400370190
3386	48	98	122	1445070098
3253	40	158	178	400370190
3385	48	68	92	1445070098
3252	40	134	154	400370190
3384	48	38	62	1445070098
3340	46	37	70	476781759
3251	39	61	131	400370190
3493	52	97	116	1234144495
3502	52	74	96	1264350371
3203	36	47	64	1234144495
3235	37	992	1008	236547980
3389	48	188	212	1445070098
3388	48	158	182	1445070098

Tra le versioni 1.2 e 1.3 di Tika Clo_tter ha individuato 15 nuovi cloni associati a 3 committer:

ID committer	*# cloni*
1125603941	4
476781759	7
2013372900	5
Tika 1.3 nuovi cloni / Committer	

Tra la versione 1.3 e 1.4 Tika Clo_tter ha individuato 16 nuovi cloni associati a 5 committer:

ID committer	*# cloni*
1445070098	7
400370190	4
476781759	1
236547980	1
1234144495	2
1264350371	1
Tika 1.4 nuovi cloni / Committer	

Si riportano per entrambe le versioni anche le *'nuove classi'* rispetto alla precedente:

Classi nuove nella versione Tika 1.3: 8

-Class ID: 33, #Clones: 2, Lines: 27, Similarity: 77
-Class ID: 1, #Clones: 2, Lines: 15, Similarity: 93
-Class ID: 3, #Clones: 2, Lines: 22, Similarity: 100
-Class ID: 38, #Clones: 2, Lines: 28, Similarity: 78
-Class ID: 39, #Clones: 2, Lines: 12, Similarity: 75
-Class ID: 29, #Clones: 2, Lines: 17, Similarity: 76
-Class ID: 41, #Clones: 4, Lines: 25, Similarity: 88
-Class ID: 21, #Clones: 2, Lines: 13, Similarity: 92

Classi nuove nella versione Tika 1.4: 5

-Class ID: 36, #Clones: 2, Lines: 12, Similarity: 83
-Class ID: 48, #Clones: 6, Lines: 20, Similarity: 80
-Class ID: 37, #Clones: 2, Lines: 14, Similarity: 71
-Class ID: 52, #Clones: 2, Lines: 13, Similarity: 71
-Class ID: 41, #Clones: 2, Lines: 28, Similarity: 78

Conclusioni

Lo scopo del presente lavoro di tesi è stato analizzare come i cloni software presenti in una release di un sistema software evolvono nelle release successive. In particolare è stato analizzato se un clone continua a persistere nelle release successive, se sono introdotti nuovi cloni non presenti nella release precedente, se sono introdotti nuovi cloni di uno già esistente in precedenza. Inoltre, i cloni individuati sono stati associati ai committer che li hanno introdotti e, quindi, si è verificato anche se vi sono committer che sono maggiormente propensi ad introdurre cloni software nelle varie release.

In particolare, è stato esteso il tool, Clo_tter, per effettuare l'analisi di versioni successive con l'obbiettivo di conoscere lo stato di ciascuna versione successiva in termini di cloni e classi di cloni, associati ai committer responsabili della loro introduzione.

La principale complessità di tale sviluppo è stata proprio quella relativa a capire se un clone individuato in una release fosse uno già esistente nella release precedente o uno nuovo, in quanto non è facile effettuare tale confronto a causa delle (altre) modifiche apportate nella nuova release che rendono difficile se non impossibile fare un confronto basato, ad esempio, sulla localizzazione del clone in termini di numero delle linee di codice occupate dal clone nel file sorgente.

Sono stati effettuati alcuni esperimenti utilizzando sistemi software open source prelevati da GitHub per verificare e validare il tool Clo_tter così esteso, che è risultato soddisfare i requisiti per esso specificati ed ottenendo risultati significativi negli esperimenti condotti.

L'attuale implementazione presenta un limite principale dovuto al fatto che per le versioni precedenti di un sistema non possiamo individuare i cloni eliminati nella versione successiva, informazione da associare al committer legato alla cancellazione

del clone/i. Anche l’associazione dei cloni individuati ai committer più risultare imprecisa.

Uno sviluppo futuro può permettere al tool Clo_tter di estenderne le funzionalità superando il precedente limite.

Bibliografia

[Baxter] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In Koshgoftaar, T.M., Bennett, K., eds.: International Conference on Software Maintenance, IEEE Computer Society Press (1998) 368–378

[Kaur] Geetika Chatley, Sandeep Kaur and Bhavneesh Sohal "Software Clone Detection: A review", International Journal of Control Theory and Applications · January 2016, International Science Press

[Koske] Rainer Koschke. Survey of Research on Software Clones. In Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 24pp., Dagstuhl, Germany, July 2006

[Nicad] Cordy, James R., and Chanchal K. Roy. "The NiCad clone detector." *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011.

[Git] https://www.git-scm.com/

[GitHub] https://github.com/

[Sourcetree] https://www.sourcetreeapp.com/

[UND] https://scitools.com

Ringraziamenti

A conclusione di questo lavoro di tesi ho il piacere di ringraziare tutti coloro che in questi anni hanno fatto parte della mia vita.

A partire dai miei genitori, ai quali dedico questo traguardo.

Se siamo qui oggi è grazie a loro che, convivendo con tante difficoltà, hanno sempre posto davanti a queste il mio futuro.

A mia sorella dico grazie per la pazienza nei miei confronti. Le auguro di riuscire a raggiungere i suoi obbiettivi con la speranza di una nuova vita migliore.

Al relatore Prof. Giuseppe A. Di Lucca un sentito ringraziamento per la disponibilità e competenza prestatemi nonostante i tanti impegni di docente.

Ai miei amici del "*Universibar*" e della "*Vecchia guardia*" vi sono immensamente grato per l'amicizia di questi anni, durante i quali abbiamo condiviso tanti momenti felici.

A mia nonna, a tutti i miei zii vi voglio ringraziare perché il vostro aiuto è stato fondamentale per me e per la mia famiglia. Non smetterò mai di farlo.

Alle persone care, che oggi non possono essere qui, vi ricorderò sempre con molto affetto. Grazie.