Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Dipartimento di informatica – Scienza e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di uno strumento per la scansione di progetti *software* alla ricerca di potenziali segni di plagio

Elaborato in
Programmazione ad Oggetti

Relatore
Chiar.mo Prof.
Danilo Pianini

Presentata da
Luca Tassinari

III Sessione di Laurea
Anno Accademico 2021-2022

Sommario

Questa tesi nasce dalla necessità di sviluppare uno strumento capace d'individuare potenziali plagi in progetti *software*. Il plagiarismo nel *software* è pratica che, nel tempo, ha visto numerosi scontri legali (ad esempio, Oracle contro Google per Android), e per il quale sono pochi i progetti *open source* di facile utilizzo pratico. L'elaborato, quindi, si addentra nelle tecniche di analisi e d'individuazione di possibili plagi, presentando il processo di progettazione e sviluppo dello strumento.

La tesi è strutturata in capitoli. Nel primo viene introdotto il contesto, vengono esposti i problemi da affrontare durante lo sviluppo di strumenti antiplagio e viene presentata una panoramica delle principali tecniche presenti in letteratura. Nel secondo e terzo capitolo ci si addentra nell'analisi dei requisiti, nella progettazione e nel processo d'implementazione dello strumento. Un ultimo capitolo è dedicato all'analisi dei risultati ottenuti.

Indice

Elenco delle figure

Elenco dei listati

Elenco delle tabelle

Capitolo 1

Contesto e motivazioni

Negli ultimi decenni, con la rapida crescita di dati e informazioni facilmente accessibili sul *web*, è diventato sempre più semplice poter utilizzare, in parte o in tutto, le risorse reperite. Tuttavia, l'uso improprio di tali risorse e il loro appropriamento senza attribuire i necessari crediti agli autori, in violazione della legge sul *copyright*, costituisce un **plagio** [Bri22] che, oltre ad essere una pratica scorretta che contravviene a qualsiasi ordine deontologico, rappresenta un illecito punito a norma di legge [Uff].

Tra tutte, le due tipologie di plagio più comuni sono i plagi testuali, ovvero quelli commessi in documenti scritti in linguaggio naturale, e quelli del codice sorgente, che riguardano progetti *software*, su cui verte la trattazione di questa tesi.

1.1 Il problema del plagio nel *software*

Anche nel mondo dell'informatica il problema del plagio è un fenomeno in crescita, incoraggiato per lo più dalla sempre maggiore quantità di progetti *software open source*, che induce gli sviluppatori a copia incollare frammenti di codice, talvolta neppure conoscendo le relative condizioni e termini di licenza.

Prima di addentrarci nell'analisi dei metodi e delle tecnologie utilizzate per l'identificazione di plagi nel contesto dei progetti informatici, è di fondamentale importanza fornire una visione d'insieme sulle strategie che vengono impiegate da parte degli sviluppatori per cercare di offuscare la copiatura quando questa è un atto deliberato.

In generale, non è possibile classificare tutti i possibili metodi con cui un programma può essere trasformato in un altro mantenendo inalterate le sue funzionalità. Tuttavia, è possibile distinguere due macro categorie di modifiche: **lessicali** e **strutturali** [JL99].

Le modifiche lessicali sono quelle che, in linea di principio, possono essere eseguite da un *text editor* e non richiedono la conoscenza del linguaggio di programmazione con cui è stato sviluppato il codice. Alcuni casi esemplificativi sono:

- la riformulazione di commenti, la loro aggiunta o rimozione;
- la riformattazione del testo, come l'introduzione di spazi vuoti, di nuove linee o il cambio dell'ordine dei parametri nella definizione delle funzioni;
- cambiare il nome degli identificatori e delle funzioni o i tipi di dato: ad esempio da `int` a `Integer` o da `float` a `double`.

Le modifiche strutturali sono invece fortemente dipendenti dal linguaggio di programmazione e richiedono un maggior sforzo in termini di comprensione della logica del codice. Di seguito alcuni esempi di rifattorizzazioni che rientrano in questa classe:

- aggiungere istruzioni ridondanti, come dichiarazioni, inizializzazioni, istruzioni di stampa;
- sostituire i costrutti di loop con costrutti equivalenti: passare da `for` a `do/-while`, o da un approccio iterativo a uno funzionale, ad esempio tramite l'uso degli `Stream` in Java;
- sostituire istruzioni `if` nidificate con dichiarazioni equivalenti, ad esempio `switch-case` o `when` in Kotlin, e viceversa;
- cambiare l'ordine d'istruzioni indipendenti;
- cambiare l'ordine degli operandi: ad esempio `x < y` può essere cambiato in `y >= x`;
- sostituire la chiamata a funzione con il corpo della stessa.

In Figura 1.1 viene riportata la tassonomia dei livelli di plagio di Faidhi & Robinson definita in [FR87] che mappa le possibili rifattorizzazioni in sette livelli o categorie, sulla base della loro difficoltà: il più semplice è il livello zero che corrisponde a una copia letterale; il più impegnativo è il sesto livello che corrisponde a un cambiamento logico e può essere considerato plagio solo se si verifica in concomitanza con altri livelli.

Un qualunque sistema di rilevamento di plagi, perché possa essere considerato robusto ed efficace, dovrà tener conto di queste possibili rifattorizzazioni in fase di esecuzione.

1.2 Sistemi antiplagio automatici

A causa dell'elevato volume di dati e informazioni disponibili che oggigiorno si hanno a disposizione, l'approccio tradizionale al rilevamento di codice copiato attraverso un'ispezione manuale, oltre che essere tedioso, è nei fatti impraticabile e,

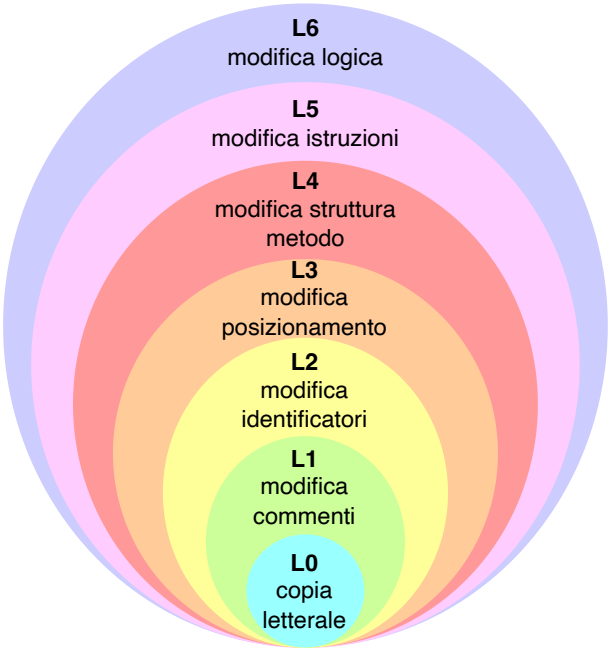

Figura 1.1: Tassonomia dei livelli di plagio di Faidhi & Robinson (1987).

nel corso degli anni, è stato necessario porsi il problema di sviluppare strumenti di rilevamento automatici di plagi che fossero efficienti ed efficaci. Ciononostante, un'ispezione manuale, a seguito di quella automatica, è tuttora ancora necessaria per stabilire con ragionevole certezza se le porzioni di codice individuate dal sistema sono effettivamente copiature o, più ingenuamente, blocchi simili di codice.

Più formalmente, detto $p = \{s_{plg}, d_{plg}, s_{src}, d_{src}\}$ un caso di plagio, s_{plg} un passaggio del documento d_{plg} che è copiato a partire da un passaggio s_{src} in un documento d_{src}, il compito di un sistema automatico di rilevamento di plagi consiste nell'individuare la tupla p [YCE22].

Già a partire dagli anni settanta del novecento, sono stati proposti algoritmi e tecniche per l'analisi del codice, nonché l'identificazione e localizzazione di sezioni simili di sorgenti.

In questo contesto è da porre in evidenza la differenza tra l'individuazione di cloni e quella di plagi. Quando ci si riferisce a un clone, infatti, lo si fa con riferimento a un frammento di codice che è stato copiato e marginalmente modificato. Quando invece ci si riferisce ad un plagio si intende una sezione che è stata copiata e la cui opera di copiatura si è cercato di dissimulare mediante opportune azioni di rifattorizzazione del codice [MAB13], descritte nel paragrafo precedente. Dunque, l'ambito di applicazione delle due ricerche è nettamente diverso: se nel primo l'obiettivo è quello di evidenziare il codice duplicato al fine di migliorare la

qualità del codice e la manutenibilità del sistema, nel secondo lo scopo primario è identificare possibili condotte illecite.

Questo aspetto è, insieme alle prestazioni, la sfida principale da affrontare durante la progettazione di un sistema antiplagio.

Gli strumenti d'identificazione di plagi devono infatti cercare di annullare gli effetti di queste rifattorizzazioni. Maggiore sarà il livello di rifattorizzazione a cui il sistema sarà insensibile, migliore sarà l'efficacia e la robustezza del sistema stesso. A questo scopo le tecniche di analisi si compongono di più fasi nelle quali trasformano i sorgenti in rappresentazioni intermedie che astraggono il più possibile dai dettagli implementativi, che possono essere facilmente cambiati, quindi applicano su di esse tecniche di confronto.

Si osservi, tuttavia, che esistono rifattorizzazioni il cui contributo è più facile da annullare di altre. Si pensi, ad esempio, all'aggiunta o alla modifica dei commenti: l'effetto di tale rifattorizzazione è facilmente annullabile semplicemente trascurando dall'analisi i commenti, in quanto non contribuiscono in alcun modo alla logica del programma. D'altro canto, creare tecniche che siano insensibili al riordino delle istruzioni è un compito più complesso, fonte di approfonditi studi e ambito di ricerca.

L'altro problema emergente nello sviluppo di un programma antiplagio che non si limiti a confrontare la similarità tra una coppia di progetti, bensì effettui un controllo uno a molti o molti a molti, in cui si testano tutte le possibili coppie, sono le prestazioni. Infatti, la maggioranza delle tecniche e degli algoritmi per effettuare i confronti sono inefficienti in termini di tempo costo. Questo è in larga parte dovuto al fatto che la misurazione della somiglianza tra una coppia di sorgenti, nella gran parte degli algoritmi noti in letteratura, ha una complessità almeno quadratica nel numero delle istanze delle sue rappresentazioni, e che, per ogni valutazione, il numero di confronti da effettuare è tipicamente elevato: detto N il numero di progetti, volendo confrontare tutte le coppie di progetti tra loro, dovrebbero essere eseguite $\frac{N(N-1)}{2}$ comparazioni. Questo problema è acuito inoltre dal fatto che i progetti *software* stanno diventando sempre più complessi e si hanno a disposizione una sempre maggior quantità di dati da dover processare.

Per questa ragione vengono sfruttate tecniche di parallelizzazione e devono essere adottate strategie di ottimizzazione in grado di ridurre il numero di confronti da effettuare e, quindi, diminuire il tempo di esecuzione. Per farlo, si utilizzano tecniche euristiche che permettono di determinare il grado di somiglianza dei sorgenti senza dover effettivamente eseguire il confronto.

Va da sé che l'utilizzo di tali tecniche impatta inevitabilmente la sensibilità del sistema: una stima errata, a monte, della somiglianza di due sorgenti può portare a non effettuare alcun confronto tra questi e, quindi, a non individuare possibili plagi. Questo si verifica soprattutto nei casi in cui l'ottimizzazione è molto marcata

e si riduce in modo eccessivo l'insieme dei sorgenti su cui effettuare il confronto, rendendo di fatto svantaggioso l'impiego di tali tecniche.

Il bilanciamento tra le prestazioni e l'efficacia del sistema è, dunque, di fondamentale importanza.

1.3 Stato dell'arte

Il codice sorgente non è nient'altro che un file di testo scritto da sviluppatori che deve essere compilato o interpretato e che, pertanto, si basa su regole sintattiche e grammaticali proprie del linguaggio di programmazione con cui è scritto che permettono a entrambi gli attori, il programmatore e il calcolatore, di "capirlo" ed elaborarlo. Per questo motivo, se processare la struttura di un sorgente non presenta grandi difficoltà, processare il significato, ovvero l'idea e la logica sottesa al codice, costituisce una sfida più grande, se non altro perché entra in gioco la competenza dello sviluppatore e la sua esperienza nella scrittura di codice "pulito".

Poiché il problema è complesso, gli attuali metodi analizzano il codice sorgente utilizzando un particolare "punto di vista", alcuni cercando di comprenderne il significato, altri la loro struttura. Con lo sviluppo della tecnologia, molti ricercatori hanno fatto progressi in questo campo e proposto diversi algoritmi, formando gradualmente tre popolari famiglie di tecniche: ***attribute-based***, ***structure-based*** e **tecniche ibride** [SK19].

1.3.1 Analisi *attribute-based*

Il primo sistema automatico di rilevazione di codice copiato che sia stato documentato in letteratura risale al 1976 [Ott76] ed era basato sulle seguenti quattro metriche di Healstead per determinare il livello di similarità tra coppie di sorgenti [Hal77]:

- η_1: il numero di operatori univoci;
- η_2: il numero di operandi univoci;
- N_1: il numero di occorrenze di un operatore;
- N_2: il numero di occorrenze di un operando.

Coppie di sorgenti con valori identici per ciascuna di queste metriche si presumeva fossero simili e meritevoli di un esame più approfondito.

Nel corso degli anni, sono state proposte altre metriche per ottenere stime sempre più accurate di somiglianza e per riflettere sempre più la struttura del flusso di controllo del programma, come il numero d'istruzioni di *loop* e di espressioni condizionali, il numero di procedure e d'istruzioni di *input*, il numero medio di caratteri per linea e altre ancora [MV05].

Collettivamente, questi sistemi sono stati definiti *attribute counting metric systems* in quanto fondano il confronto sull'analisi di metriche basate su attributi intrinseci del codice sorgente.

Tuttavia, considerando che le caratteristiche del codice sorgente non sono strettamente correlate alla semantica del programma e che la misurazione della somiglianza può risultare essere imprecisa, l'efficacia di queste tecniche è considerevolmente limitata in termini di accuratezza se confrontata con quella di tecniche *structure-based* [SK19].

1.3.2 Analisi *structure-based*

Le tecniche di analisi *structure-based*, a differenza di quelle *attribute-based*, si basano, come suggerisce il nome stesso, sulla struttura dei codici sorgenti per determinare il grado di similarità tra gli stessi.

Nella maggioranza dei casi, i sistemi che implementano questo tipo di tecnica lavorano in due fasi consecutive: prima il codice sorgente viene analizzato e viene generata una rappresentazione intermedia, poi si effettua il confronto dei sorgenti sulle rappresentazioni intermedie.

All'interno di questo tipo di analisi possiamo annoverare principalmente due diversi metodi: l'analisi lessicale e quella basata su un modello astratto e derivato dai sorgenti.

Analisi lessicale (*tokenizzazione*)

Tra tutte, la tecnica più usata è l'analisi lessicale del codice sorgente (***lexical analysis*** o ***tokenization*** in inglese), che consiste nel convertire la sequenza di caratteri di cui è composto il programma in una sequenza di *token*. Un *token* è, a sua volta, un simbolo astratto che rappresenta un tipo di unità lessicale nella grammatica del linguaggio. Esso è associato a un **lessema**, ovvero ad una sequenza di caratteri del programma che corrisponde al *pattern* di un *token* identificato dall'analizzatore lessicale come una specifica istanza di quel *token*, e da attributi opzionali derivati dal testo. Alcuni possibili esempi di *token* comuni con i relativi *pattern* e alcuni lessemi di esempio sono riportati in Tabella 1.1.

L'analisi lessicale rappresenta il primo stadio della struttura di un compilatore ed è eseguita da programmi denominati ***lexer***. Questi sono generati in maniera dichiarativa a partire da generatori di *lexer* (***lexer generator***) che, presi in *input* più automi a stati finiti, espressi per mezzo di espressioni regolari che definiscono in maniera formale la grammatica del linguaggio, generano il codice che implementa l'algoritmo di analisi lessicale. In Figura 1.2 è mostrato un esempio di automa a stati finiti che riconosce un identificatore. L'equivalente espressione regolare è la seguente: `[a-zA-Z].([a-zA-Z]|[0-9])*`.

Token	**Pattern**	**Lessemi di esempio**
identifier	Stringa di lettere e numeri che iniziano con una lettera	`x`, `name`, `color`, ...
keyword	Coincide con la sequenza di caratteri della *keyword* stessa	`if`, `for`, `return`, ...
literal	Ogni stringa racchiusa tra virgolette (`""`) tranne la stringa nulla "	`"Hello World!"`, ...
operator	`"<"`, `">"`, `"<="`, `">="`, `"=="` `"!="`	`<=`, `!=`, ...

Tabella 1.1: Esempi di possibili *token* comuni.

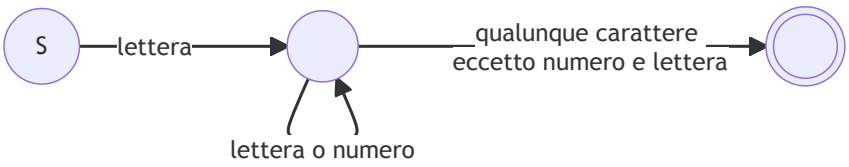

Figura 1.2: Esempio di automa a stati finiti che riconosce un identificatore.

Attraverso questo approccio, quindi, ogni programma viene trasformato in una sequenza di *token*, uno per ciascun elemento di base del linguaggio che si vuole valorizzare: le sezioni di codice non rilevanti ai fini della comparazione possono essere escluse, cioè non viene generato alcun *token* per questi elementi.

In Tabella 1.2 è mostrata una possibile sequenza di *token* generata a partire dalla classe presentata nel Listato 1.1. Il risultato che si ottiene è una versione condensata e semplificata del sorgente, con un vocabolario ristretto, in cui le dichiarazioni superflue, come quelle di `import`, `package` e i blocchi di documentazione, sono ignorati.

Questo approccio rende il sistema sufficientemente robusto contro semplici tecniche di *refactoring*, come la modifica degli identificatori, corrispondente al secondo livello della tassonomia di Faidhi & Robinson (Figura 1.1).

Tuttavia, bisogna evidenziare come l'efficacia della *tokenizzazione* dipenda dall'insieme di tipi di *token* usato. Consideriamo, a puro scopo esemplificativo, il caso delle *keyword* dedicate a rappresentare un intero in Java. Come sappiamo, un intero è rappresentabile con diversi tipi di dato; a seconda delle esigenze di memoria, infatti, si potrebbe usare, considerando solo i tipi di dato primitivo e tralasciando quelli *boxed*, un `byte`, piuttosto che uno `short`, un `int` o un `long`. Un insieme "stringente" di *token* potrebbe rappresentare ciascuno di questi tipi con un proprio *token*. D'altro canto, un set di *token* "lasco" potrebbe rappresentare tutti e quattro i tipi di dato sopra citati con un unico *token*, riconoscendo

Listato 1.1: Una semplice classe che stampa su *console* gli argomenti passati in *input*, se presenti, o un saluto, per mostrare il processo di *tokenizzazione*.

```java
package org.examples;

import java.util.Arrays;

/**
 * This is a sample class to demonstrate the tokenization process.
 */
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Program arguments: " + Arrays.
               toString(args));
        } else {
            System.out.println("Hello world from Java!");
        }
    }
}
```

che tutti e quattro possono servire, nella grande maggioranza dei casi, allo stesso scopo. Pertanto un insieme di *token* "lasco" aiuta il sistema a riconoscere e catalogare come equivalenti tipi di dato diversi e fornire quindi una maggiore resilienza alle tecniche di rifattorizzazione.

Spesso la fase di analisi lessicale è seguita da una fase di filtraggio che consente di ridurre il numero di rappresentazioni da dover confrontare e, quindi, il numero stesso dei confronti, portando a una riduzione dei tempi di esecuzione. Questo stadio è preceduto da una fase d'indicizzazione in cui, a partire dalla sequenza di *token*, vengono generati alcuni indici, ovvero strutture che raggruppano i dati e che permettono d'inferire una stima euristica della similarità senza dover effettuare il confronto *token* per *token*. Uno tra gli indici più comuni è costituito da coppie chiave−valore in cui, nelle chiavi, sono memorizzati i tipi di *token* e, nei valori, la loro frequenza, in termini di numero di occorrenze all'interno del sorgente. Un esempio di indice costruito a partire dalla sequenza di *token* elencata in Tabella 1.2 è mostrato in Tabella 1.3. Tali indici sono successivamente impiegati nella vera e propria fase di filtraggio in cui viene applicata una particolare metrica, ad esempio la similarità coseno, per escludere le coppie di sorgenti con similarità inferiore ad un valore di soglia, scelto coerentemente alla metrica utilizzata.

A seguire i sorgenti vengono confrontati con algoritmi di *string matching*. Tra questi, i due più conosciuti sono ***Winnowing*** [SWA03] e ***Running-Karp-Rabin Greedy String Tiling (RKR-GST)*** [Wis93].

Linee	**Costrutto**	***Token***
1	`package`	-
3	`import`	-
5-7	`/** This is ... process.*/`	-
8	`public`	-
8	`class`	`ClassOrInterfaceDeclaration`
8	`Main`	`Identifier`
8	`{`	-
9	`public`	`Modifier`
9	`static`	`Modifier`
9	`void`	`VoidType`
9	`main`	`Identifier`
9	`String[]`	`ArrayType`
9	`args`	`Identifier`
9	`{`	-
10	`if`	`IfStmt`
10	`args.length`	`FieldAccessExpr`
10	`>`	`Operator`
10	`0`	`IntegerLiteralExpr`
10	`{`	-
11	`System.out.println`	`MethodCallExpr`
11	`"Program arguments"`	`StringLiteralExpr`
11	`Arrays.toString`	`MethodCallExpr`
11	`args`	`NameExpr`
11	`}`	-
12	`else`	`elseStmt`
12	`{`	-
13	`System.out.println`	`MethodCallExpr`
13	`"Hello world ... Java!"`	`StringLiteralExpr`
14-16	`}`	-

Tabella 1.2: Associazioni costrutti-*token* del Listato 1.1.

Winnowing costituisce un miglioramento della tecnica nota come *k-gram fingerprinting* e consiste nei seguenti passi:

- il documento viene suddiviso in una lista di sottostrighe di lunghezza k, denominati *k-grams*;
- grazie all'utilizzo di un'opportuna funzione di *hashing* ciascun *k-gram* è trasformato in un valore numerico;

Tipo di *token*	**Frequenza**
`ClassInterfaceDeclaration`	1
`Identifier`	3
`Modifier`	2
`VoidType`	1
`ArrayType`	1
`IfStmt`	1
`FieldAccessExpr`	1
`Operator`	1
`IntegerLiteralExpr`	1
`MethodCallExpr`	3
`NameExpr`	1
`elseStmt`	1
`StringLiteralExpr`	2

Tabella 1.3: Indice con coppie (tipo di *token*−frequenza) generate a partire dalla Tabella 1.2.

- la sequenza numerica è trasformata in una sequenza di "finestre" di valori di *hash*, in cui la grandezza di ciascuna, detta w, è data da: $w = t - n + 1$ dove t rappresenta la lunghezza minima delle stringhe che è garantito siano trovate e n rappresenta la lunghezza massima delle stringhe che è garantito *non* siano trovate;
- sono selezionati un sottoinsieme dei valori di *hash*, da usare come impronta digitale del documento, considerando il valore minimo di ogni "finestra".

La comparazione avviene successivamente sulla base di queste "impronte digitali", che sono dimostrate essere rappresentative della struttura del documento [SWA03]. In Figura 1.3 è presentato un esempio tratto da [SWA03] in cui vengono mostrati i passi dell'algoritmo applicati a una semplice stringa di caratteri.

Running-Karp-Rabin Greedy String Tiling (RKR-GST) costituisce l'altra popolare alternativa a *Winnowing*, che tenta di aumentare la sua precisione a discapito dell'efficienza. Esso incorpora, di fatto, due algoritmi. *Greedy String Tiling (GST)* è l'algoritmo che trova la massima sottosequenza comune in due stringhe. *Running-Karp-Rabin (RKR)* è l'algoritmo per la ricerca rapida di sottosequenze comuni e trova tutte le occorrenze di una stringa P all'interno di una stringa più lunga T effettuando l'*hashing* di tutte le sottostringhe di lunghezza $|P|$ e confrontando tutti i valori *hash* di T con quelli di P. I due algoritmi cooperano in questo modo: *RKR* è eseguito su due programmi. Per ogni sottosequenza comune identificata da *RKR*, *GST* è eseguito per estendere il *match* rintracciato mediante l'uso

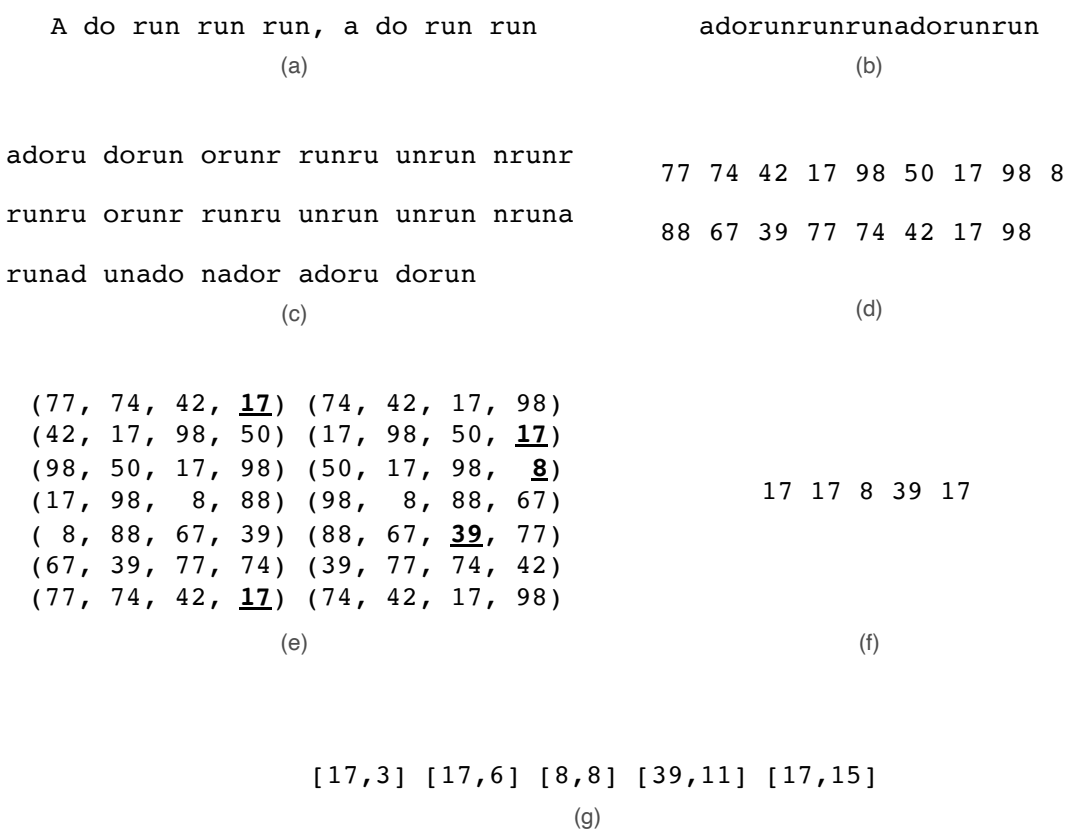

Figura 1.3: Passi esemplificativi dell'algoritmo *Winnowing* applicati a una semplice sequenza di caratteri. In (*b*) le caratteristiche irrilevanti sono state rimosse (virgole, spazi) e in (*c*) la sequenza di caratteri è trasformata in una sequenza di *5-gram*. In (*d*) è mostrata una possibile sequenza risultante dell'applicazione di una funzione di *hashing*. A partire da tale sequenza viene creata una sequenza di "finestre" di lunghezza 4, da ciascuna delle quali è selezionato il valore di *hash* minore (in grassetto sottolineato) avendo cura di non selezionare più volte lo stesso valore a causa della sovrapposizione delle finestre (*f*). Infine, in *g*, ciascun valore di *hash* selezionato viene accoppiato con l'indice *0-based* della sua posizione al passo (*d*).

della funzione di *hashing*. I dettagli e lo pseudocodice dell'algoritmo sono descritti nella Sezione 3.3.

Analisi basata su un modello

Anziché utilizzare la sequenza di *token* generata dal *lexer* possono essere generate delle strutture e dei modelli più complessi, che permettano di astrarre ulteriormente

dalla sintassi del linguaggio.

Una tra le rappresentazioni più note e naturali in questo ambito, derivata dalla stessa analisi lessicale, è l'**albero sintattico** (o *Abstract Syntax Tree* in inglese, abbreviato *AST*). Questo è generato a partire dalla sequenza di *token* descritta nel paragrafo precedente e rappresenta il risultato ottenuto a seguito dell'applicazione del secondo stadio di un compilatore: il ***Parser*** (Figura 1.4). Esso, infatti, presa in *input* la sequenza di *token*, la converte in una struttura dati ad albero che rappresenta la struttura logica del codice sorgente.

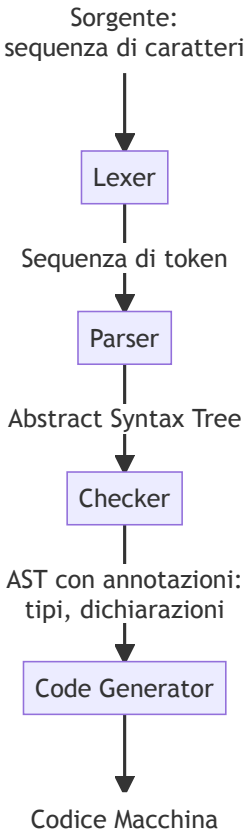

Figura 1.4: Struttura sommaria di un compilatore.

L'albero risultante dall'applicazione del *parser* è definito "astratto" in quanto non rappresenta ogni dettaglio tipico della sintassi del linguaggio, bensì solo i dettagli strutturali e relativi al contenuto. Ad esempio, le parentesi di raggruppamento sono rappresentate nativamente nella struttura ad albero, quindi non devono essere rappresentate come nodi separati. Allo stesso modo, un costrutto

sintattico come un'istruzione `if-then-else` può essere facilmente denotata per mezzo di un singolo nodo con tre rami.

In Figura 1.5 viene mostrata la struttura parziale dell'*AST* del metodo `main()` del Listato 1.1.

Avere a disposizione una struttura ad albero per ciascun sorgente è di fondamentale importanza e agevola la fase di *preprocessing* in cui si va "ripulire" il sorgente di tutte le sezioni non rilevanti. Inoltre, il confronto dei sorgenti per mezzo di alberi sintattici fornisce risultati migliori di quelli ottenuti mediante una pura analisi lessicale del codice. Il principale e non trascurabile svantaggio di utilizzare questo approccio, tuttavia, risiede nelle prestazioni. Siccome la maggior parte degli AST sono strutture complesse e voluminose, la comparazione diretta degli alberi sintattici per una grande quantità di sorgenti è proibitiva. Per far fronte a questo problema, per ogni albero viene determinato, mediante un'opportuna funzione di *hashing* che tiene conto del tipo di ciascun nodo e del suo sotto-albero, un valore di *hash* e la comparazione è effettuata confrontando tali valori.

Altri approcci sono presenti in letteratura, tra cui l'utilizzo di grafi che descrivono le dipendenze del programma, detti *Program Dependency Graph*, abbreviato *PDG* [Liu+06]. *PDG* è una rappresentazione di una procedura del sorgente a grafo in cui le istruzioni rappresentano l'insieme dei vertici e le dipendenze di dati e di controllo tra le istruzioni rappresentano gli archi. In Figura 1.6 è mostrato un esempio, tratto da [Liu+06], di *PDG* per le due funzioni presentate nel Listato 1.2. La comparazione tra questi avviene sfruttando l'isomorfismo di grafi[1].

Listato 1.2: Due semplici funzioni, in C, che effettuano la somma di un *array*, utilizzate per mostrare il corrispondente *PDG*.

```
1  int sum(int array[], int count) {
2      int i, sum;
3      sum = 0;
4      for(i = 0; i < count; i++) {
5          sum = add(sum, array[i]);
6      }
7      return sum;
8  }
9
10 int add(int a, int b) {
11     return a + b;
12 }
```

[1]Nella teoria dei grafi l'isomorfismo di due grafi, G e H, è una biiezione tra i loro vertici che preserva i bordi, ovvero tale che, data una coppia qualsiasi (u, v) adiacente in G, la coppia $(f(u), f(v))$ sia ancora adiacente in H.

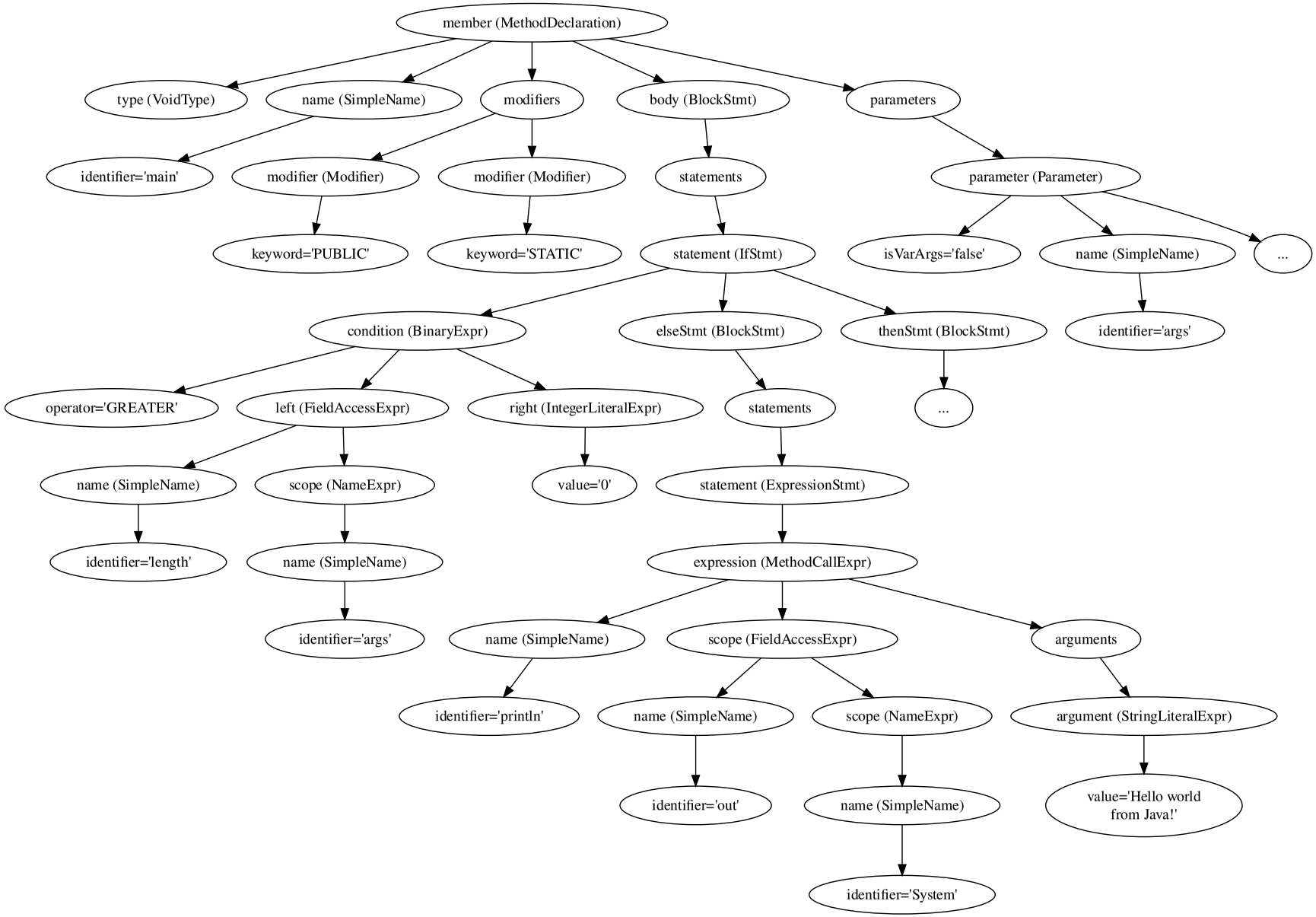

Figura 1.5: Rappresentazione di una porzione dell'*AST* del metodo `main` del Listato 1.1.

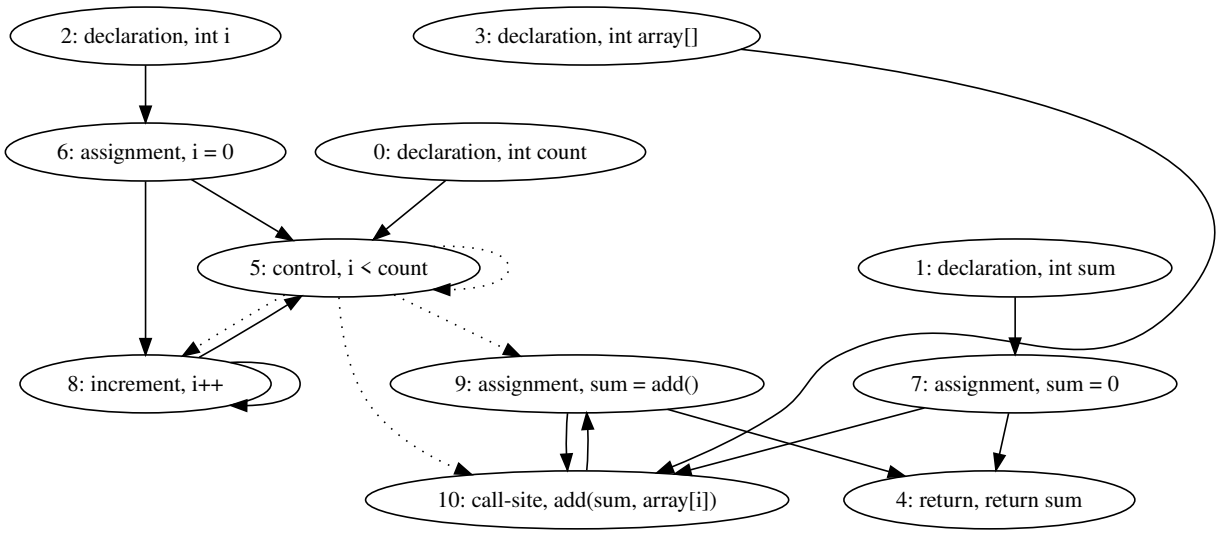

Figura 1.6: Esempio di *PDG* della funzione `sum` del Listato 1.2.

1.3.3 Tecniche ibride

Entrambi i tipi di tecniche descritti sopra hanno i loro vantaggi e svantaggi. Alcuni lavori di ricerca hanno pertanto cercato di combinare entrambi gli approcci creando, di fatto, tecniche di analisi ibride, alcune delle quali adattate da altri domini, regolando l'influenza di un approccio sull'altro sulla base delle proprie esigenze. Tra queste, spiccano per importanza la *Latent Semantic Analysis* [UJM21] e le tecniche di *Clustering* [MV05].

La prima è un metodo di elaborazione del linguaggio naturale che analizza le relazioni tra un insieme di documenti e i termini in esso contenuti: dapprima sono applicati modelli statistici che riflettono l'importanza che ciascun termine (nel caso dei sorgenti, i *token*) assume in un insieme di documenti e, successivamente, attraverso l'utilizzo di una tecnica algebrica di fattorizzazione di una matrice basata sull'uso di autovalori e autovettori detta *Singular Value Decomposition*, vengono trovate relazioni nascoste (latenti, da qui il nome *Latent Semantic Analysis*) tra i termini e concetti.

La seconda utilizza una rappresentazione a grafo in cui i vertici rappresentano gli identificatori e gli archi (pesati) la misura di similarità tra i programmi, trasformando il problema di trovare sezioni di codice duplicati nel problema di *clustering* di un grafo pesato.

Altri metodi, qui non approfonditi, sono in via di sviluppo e impiegano algoritmi di *machine learning* come *Random Forest* e *Gradient Boosting* applicati agli alberi di regressione.

1.3.4 Quale tecnica scegliere?

In conclusione, bisogna rendersi conto che, a prescindere dal grado di sofisticatezza della tecnica che si utilizza, è sempre possibile che si verifichi un plagio non rilevabile [JL99]. Il bilanciamento tra le risorse investite nell'individuazione dei plagi e i rendimenti decrescenti di trovare i pochi, se non nessuno, casi difficili da rilevare è un fattore critico che necessita un'analisi approfondita, tenuto conto delle specifiche esigenze del dominio applicativo.

Stabilire quale tecnica scegliere è complesso, perché difficili da confrontare tra loro: la maggior parte di queste viene valutata utilizzando il proprio *set* di dati, che raramente sono resi pubblicamente accessibili e che potrebbero non rappresentare veri casi di plagio [KAR+19].

Ciononostante, le tecniche più utilizzate in questo ambito e più efficaci sulla base delle considerazioni discusse, sono quelle *structure-based*, che verranno impiegate anche nello strumento sviluppato nell'ambito di questa tesi.

Capitolo 2

Analisi e Progettazione

In questo secondo capitolo viene presentata l'analisi dei requisiti e il *design* del sistema. Nei primi due paragrafi vengono elencati i requisiti ed è descritto il dominio applicativo. Il terzo paragrafo è dedicato alla progettazione dello strumento: si parte da una visione architetturale e a seguire si dettagliano le parti di *design* più rilevanti al fine di chiarificare la logica con cui è stato implementato il sistema.

2.1 Requisiti

Di seguito vengono descritti, per punti, i requisiti del sistema, suddivisi tra requisiti *funzionali* e *non funzionali*.

Requisiti funzionali

- Il sistema riceve in *input* un insieme di progetti di cui si vuole verificare l'autenticità, detto ***Submission***, e un insieme di progetti con cui confrontarli, detto ***Corpus***;
- Il confronto viene effettuato tra progetti sviluppati nello stesso linguaggio di programmazione: Java.
- I progetti sono mantenuti in *repository* pubbliche su *GitHub* e *Bitbucket*[1]. Si assume che i progetti passati siano tempo-invarianti: dal momento in cui vengono corretti, le rispettive *repository* sono archiviate e mai più modificate;
- Il sistema deve fornire in *output* le sezioni di codice che, con un determinato livello di accuratezza, ha stabilito essere simili.

[1] *GitHub* e *Bitbucket* sono due tra i più conosciuti servizi di *hosting* per progetti *software* che utilizzano sistemi di controllo di versione decentralizzati, come Git.

Requisiti non funzionali

- L'algoritmo per determinare la similarità, così come le metriche utilizzate, devono essere interscambiabili, facilmente estendibili e configurabili da parte dell'utente;
- Le informazioni estrapolate dai sorgenti o i sorgenti stessi sono salvati in modo tale da essere riutilizzati nelle analisi successive di altri progetti. Si noti che questo è possibile data la natura, invariante nel tempo, degli stessi. Questo presumibilmente abbatterà i tempi di esecuzione derivanti dallo scaricamento dei sorgenti, che rappresenta spesso un notevole "collo di bottiglia".
- È necessario che il sistema impieghi un tempo "ragionevole" per effettuare la computazione.

2.2 Analisi e modello del dominio

Il sistema deve essere in grado, a partire da un insieme di `Repository`, corrispondenti a progetti coerenti per linguaggio di programmazione, di estrarne una rappresentazione confrontabile (`SourceRepresentation`) mediante opportuni algoritmi di analisi (`Analyzer`). Ciascuna coppia di rappresentazioni intermedie deve essere successivamente filtrata per mezzo di un apposito `Filter` e confrontata tramite algoritmi di rilevamento di somiglianze (`PlagiarismDetector`) al fine di poter determinare eventuali parti di codice duplicato e/o somiglianze (`ComparisonResult`), generando infine dei `Report`.

Gli elementi costitutivi il problema sono sintetizzati in Figura 2.1.

La principale difficoltà sarà individuare tecniche di analisi e di rilevamento delle somiglianze che siano robuste, ovvero permettano d'identificare casi di copiature anche se lo sviluppatore ha effettuato modifiche per offuscarle. Particolare attenzione dovrà essere posta sulla progettazione dei componenti per l'analisi e per il confronto, in quanto, data la natura del sistema, possono dover cambiare frequentemente ed essere fortemente configurabili. Inoltre, il requisito non funzionale sulle *performance* richiederà un'analisi dei tempi di esecuzione non appena il sistema sarà completato.

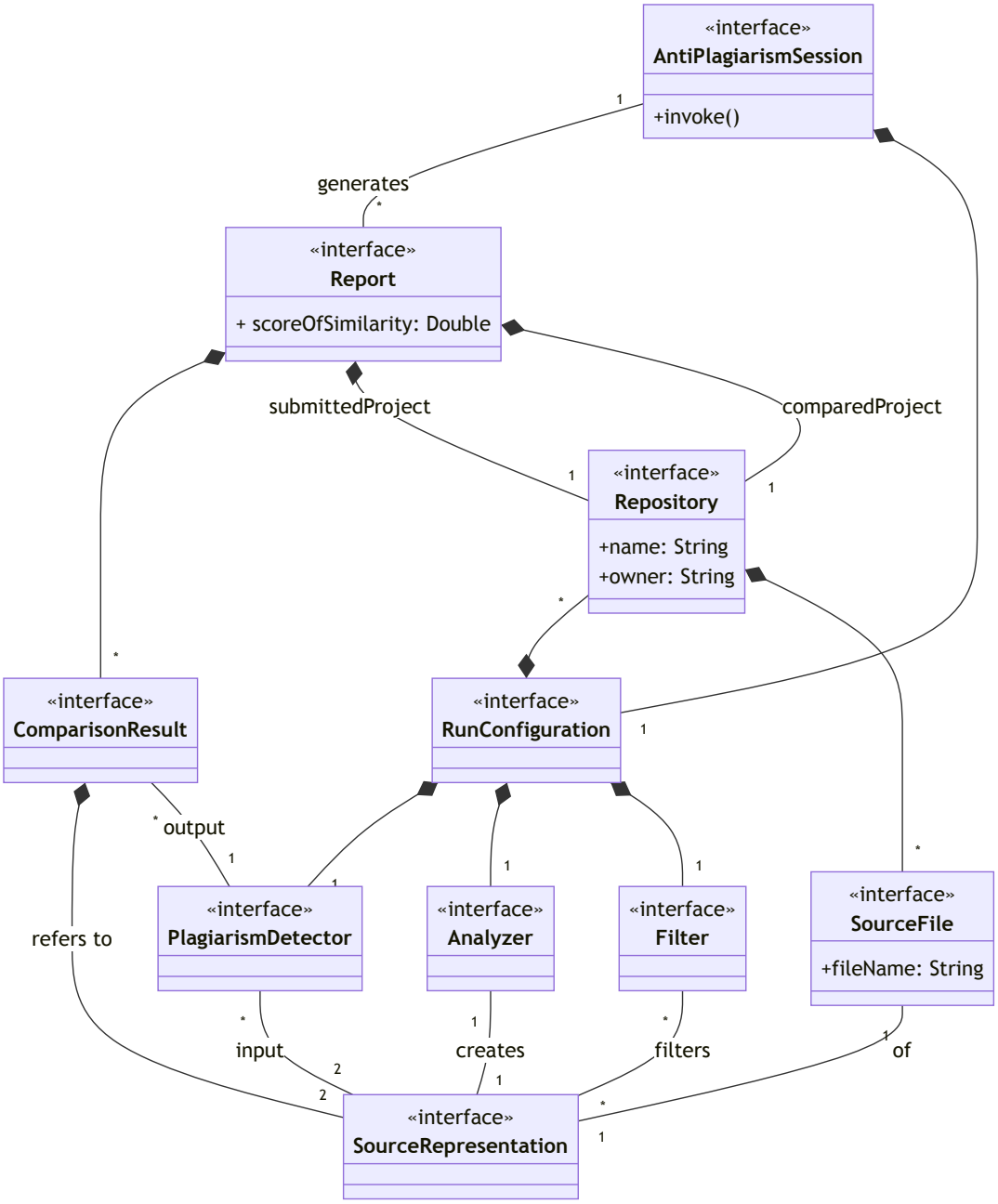

Figura 2.1: Schema UML delle classi dell'analisi del problema, con rappresentate le entità principali e i rapporti fra loro.

2.3 *Design*

2.3.1 Architettura

L'architettura del sistema è così organizzata: `AntiPlagiarismSession` è l'interfaccia responsabile della logica dell'applicazione e rappresenta una specifica sessione, dove per sessione si intende l'entità che, una volta opportunamente configurata, esegue la logica dell'applicazione. La configurazione su cui essa opera è reificata nell'interfaccia `RunConfiguration` e rappresenta un "contenitore" d'informazioni necessarie per l'esecuzione del sistema, tra cui la triade composta da `Analyzer`, `Detector` e `Filter`, nonché le `Repository` su cui deve essere effettuata l'analisi, a loro volta recuperate da un `RepositoryProvider` che si occuperà di scaricare i progetti dai servizi di *hosting* che verranno supportati dal sistema. Tale configurazione è creata da un `RunConfigurator`, a cui spetta la corretta istanziazione in accordo con le opzioni definite dall'utente.

Gli `Output` rappresentano le risorse su cui *loggare* le informazioni durante l'esecuzione. Un particolare tipo di `Output` è rappresentato dal `ReportsExporter`, il componente che si occuperà di esportare opportunamente i risultati dell'analisi.

Infine, `KnoledgeBaseRepository` rappresenta l'interfaccia che permette il salvataggio e il recupero delle rappresentazioni dei sorgenti già precedentemente analizzate e, perciò, salvate. La tecnica con cui questo verrà realizzato, salvataggio su *file* piuttosto che l'uso di un *database*, è un aspetto tecnologico implementativo che non deve impattare sull'architettura e di cui non si tiene qui conto.

Questa architettura permette di configurare opportunamente ogni sessione in base alle proprie esigenze utilizzando le tecniche e le metriche che si ritengono più consone in relazione al contesto in cui lo strumento viene calato. Inoltre, si lascia aperta la possibilità di estendere le strategie per l'esportazione dei risultati (più in generale dell'`Output`) e per la creazione della configurazione, nel caso in futuro si voglia aggiungere un'interfaccia grafica al sistema.

In Figura 2.2 è esemplificato il diagramma UML architetturale.

2.3.2 *Design* dettagliato

Provider dei progetti

Per quanto concerne i *provider* di *repository*, ovvero i componenti che devono recuperare i sorgenti dei progetti da *repository* pubbliche da *GitHub* e/o da *Bitbucket*, si è scelto un *design* che permettesse il massimo riuso degli elementi comuni, aderendo a uno schema interfaccia − classe astratta − classe concreta (Figura 2.3).

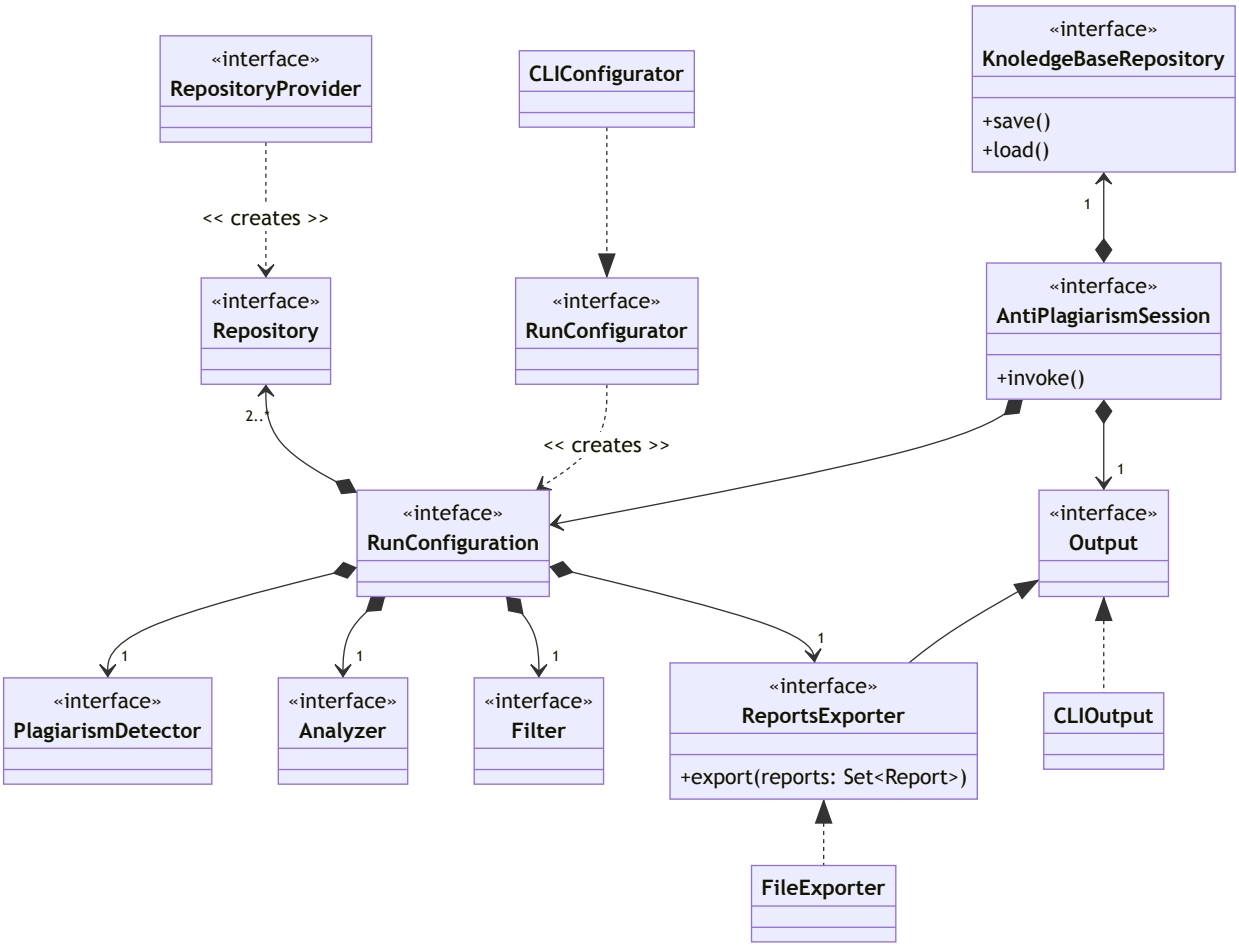

Figura 2.2: Schema UML architetturale del sistema.

Viste le limitazioni in termini di numero di richieste che i due servizi impongono[2] e la conseguente necessità di autenticarsi mediante degli opportuni *token* per effettuare le richieste *REST*, si è optato di demandare la logica di recupero dei suddetti *token* di autenticazione all'interfaccia `TokenSupplierStrategy` che reifica il *pattern Strategy* [EV44]. L'implementazione di *default* ricerca tra variabili d'ambiente, ma altri approcci possono essere introdotti in modo del tutto trasparente ai *provider*.

La creazione dei *provider* avviene per mezzo di due *static factory* [Blo18], incap-

[2]Sia *GitHub* che *Bitbuckt* hanno un limite massimo di richieste che possono essere effettuate in un certo intervallo di tempo che varia in base la richiesta sia autenticata o meno. Questi vincoli sono imposti per garantire la protezione da attacchi di tipo *Denial of Service*, consentire la scalabilità e garantire buone prestazioni in termini di tempo di risposta.

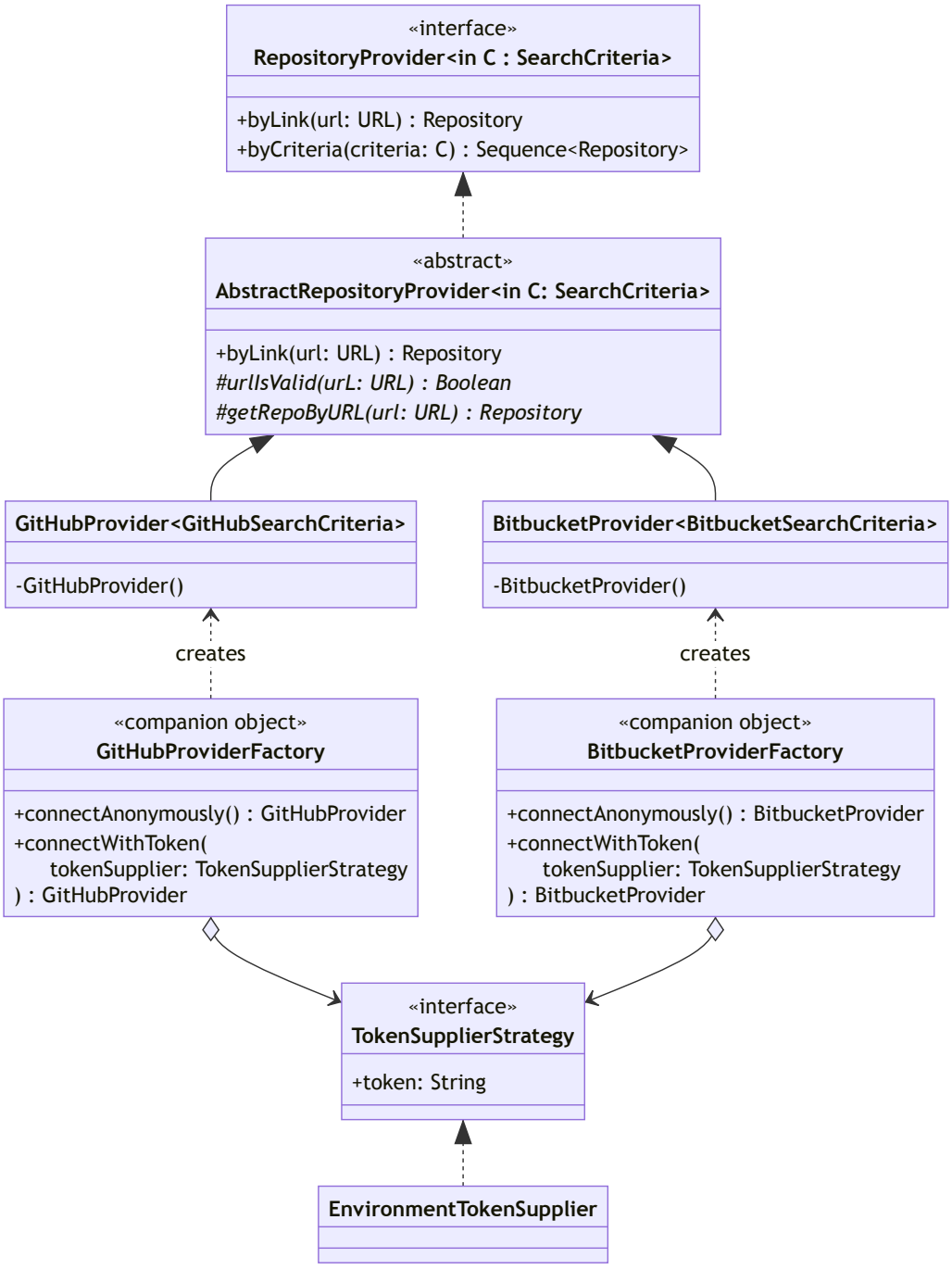

Figura 2.3: Schema UML dei provider.

sulate all'interno dei due *provider*, in modo da permettere di ottenere un oggetto *provider* anche senza autenticazione. Nonostante questo sia in genere sconsigliato per via delle limitazioni sopra citate, può essere utile in fase di *testing* per permettere di testare il sistema anche in contesti in cui non sia possibile usare *token* di autenticazione validi, ad esempio nel contesto delle *GitHub Actions* nei *workflow* scatenati da eventi di tipo *Pull Request* (si veda Sezione 3.5)

Dopo una prima analisi delle *API* dei due servizi di *hosting* si è convenuto di permettere di recuperare i progetti mediante due metodi: un *link* diretto alla *repository*, oppure mediante un criterio di ricerca, rappresentato dall'interfaccia `SearchCriteria`. Per permettere che questi siano componibili è stato qui utilizzato il pattern *Decorator* [EV44]. La classe astratta che funge da decoratore è `(GitHub|BitBucket)CompoundCriteria` e le sue concrete implementazioni permettono di specificare il nome della *repository* o il linguaggio. In questo modo possono essere creati dinamicamente criteri compositi in base alle esigenze del *client* e in futuro potrebbero essere aggiunti nuovi criteri di ricerca, come, ad esempio, il numero di stelle ricevute o la data di creazione, aderendo all'*Open Closed Principle* secondo cui le entità di un sistema devono essere aperte all'estensione, ma chiuse alla modifica. Il *design* è presentato in Figura 2.4.

Le *repository* sono modellate attraverso l'omonima interfaccia, le cui concrete implementazioni rappresentano degli *Adapter* [EV44] alle interfacce di libreria utilizzate (si veda Figura 2.5). Questo permette di essere indipendenti dalla specifica libreria utilizzata, che rimane un dettaglio implementativo, e avere un livello di astrazione adeguato al contesto applicativo.

Per quanto attiene il salvataggio dei sorgenti e il loro recupero, in modo tale che possano essere riusati in sessioni successive senza dover essere ri-scaricati, è affidato ad un gestore rappresentato dall'interfaccia `KnoledgeBaseManager`. Per il momento, la tecnica utilizzata consiste nel salvare i sorgenti in locale. Per tale motivo, la concreta implementazione `FileKnowledgeBaseManager` si occupa di scaricare i progetti, "ripulirli" da artefatti non importanti ai fini dell'analisi, come i file di configurazione di Gradle, e infine salvarli in un'apposita cartella di sistema. Nel momento in cui è necessario recuperare i sorgenti di un progetto ci si interfaccia con il gestore chiedendogli di fornire tutti i sorgenti disponibili per quel determinato progetto, già precedentemente salvati.

Un indomani, se si volesse sostituire questo metodo di *caching* con uno più avanzato, di cui oggi non se ne ravvisa l'esigenza, si potrebbe implementare un nuovo `KnoledgeBaseManager`.

Analizzatore e *detector*

L'analizzatore, il *filter* e il *detector* sono il cuore dell'intero sistema. Il loro *design* è stato pensato in modo tale da permettere il massimo grado di configurabilità e

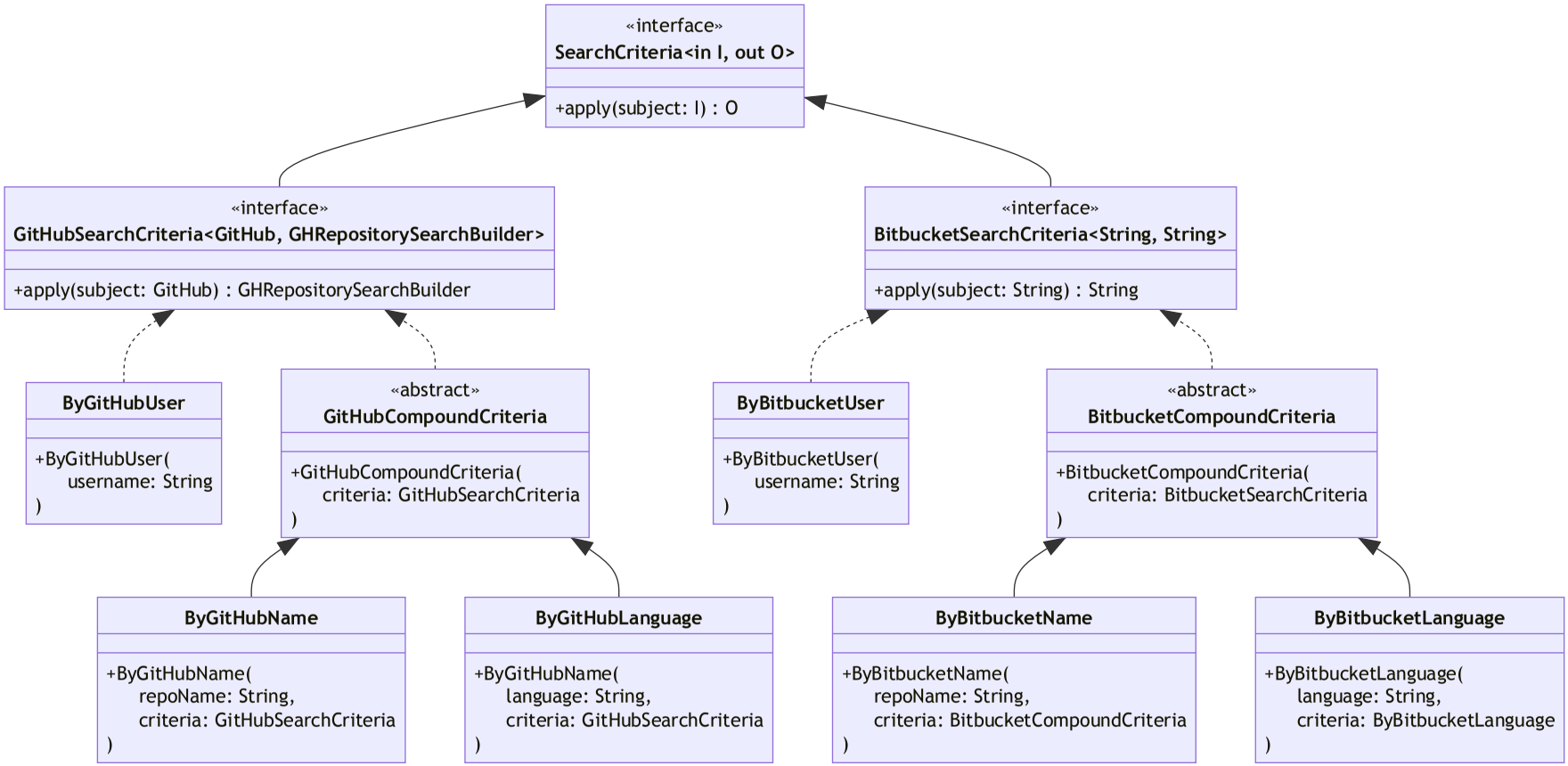

Figura 2.4: Schema UML dei criteri di ricerca delle *repository*.

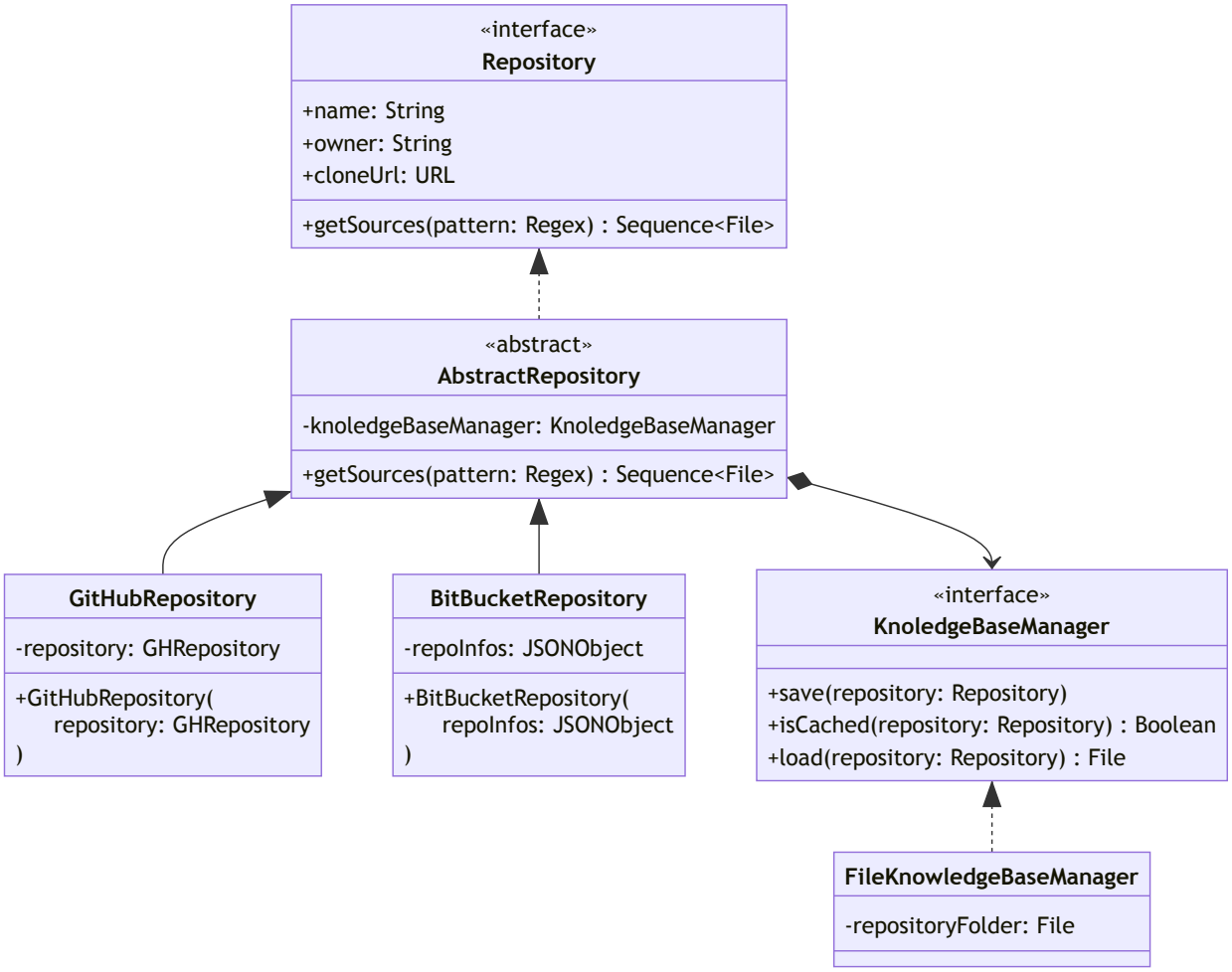

Figura 2.5: Schema UML delle *repository* e del gestore dei loro sorgenti.

di estendibilità. Presumibilmente, infatti, questi sono i componenti che varieranno più spesso nell'arco del ciclo di vita di questo *software* e il cui grado di configurabilità influirà certamentente in modo preponderante sia sulle prestazioni, sia sull'efficacia del sistema.

Tutti e tre i componenti sono stati progettati, in stile funzionale, come delle funzioni che, preso in *input* uno più argomenti, ritornano in *output* l'*input* del componente successivo. L'esecuzione di una tecnica è, infatti, una catena di funzioni applicate in sequenza.

L'analizzatore è il componente che si occupa di trasformare i file sorgenti in rappresentazioni confrontabili. Indipendentemente dalla tipologia di analisi che si voglia applicare, la trasformazione non avviene tramite un semplice passaggio,

bensì una sequenza di operazioni che sono eseguite sequenzialmente in cascata. Nel caso in esame, queste sono: il *parsing* del file, una fase di *preprocessing* ed infine la *tokenizzazione* del sorgente. Questa sequenza di azioni è naturalmente modellata attraverso il pattern *Pipeline* [Pat]: ogni singolo stadio è modellato attraverso l'interfaccia `StepHandler` e la *pipeline* viene costruita all'interno del concreto `Analyzer`, nel nostro caso `JavaTokenizationAnalyzer`. Il *design* dell'analizzatore è esemplificato in Figura 2.6.

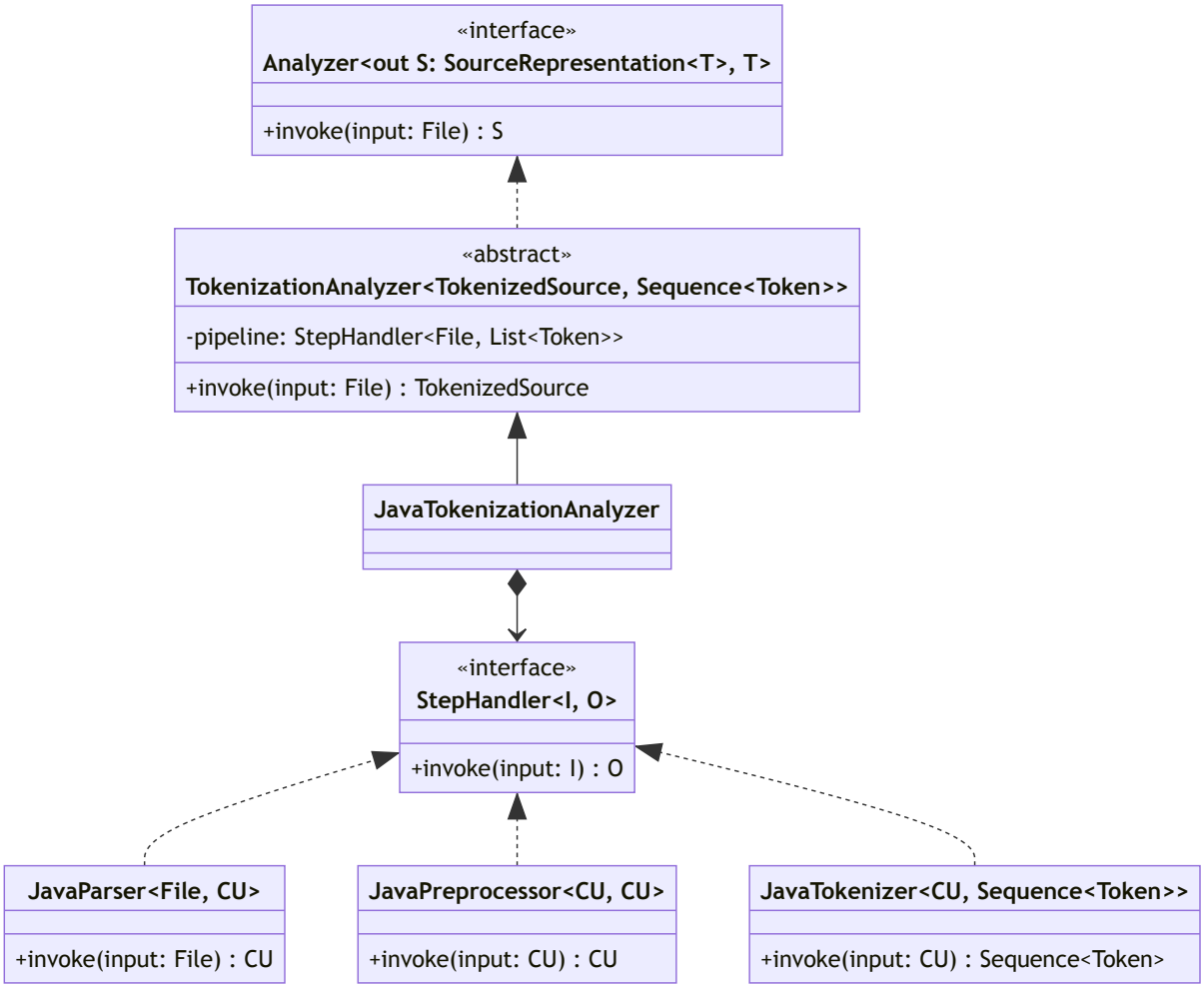

Figura 2.6: Schema UML dell'analizzatore.

La rappresentazione intermedia prodotta dall'analizzatore è modellata dall'interfaccia `SourceRepresentation` (Figura 2.7). Tra tutte, quella implementata nell'attuale sistema è composta da una sequenza di *token* e perciò denominata `TokenizedSource`. L'elenco dei possibili tipi di *token* che la compongono dipende

chiaramente dal linguaggio utilizzato. La strategia di recupero di questi ultimi è affidata, al solito tramite *Strategy*, all'interfaccia `TokenTypesSupplier`, la cui implementazione di *default* effettua la de-serializzazione di un file di configurazione in cui sono salvate tutte le tipologie di *token*.

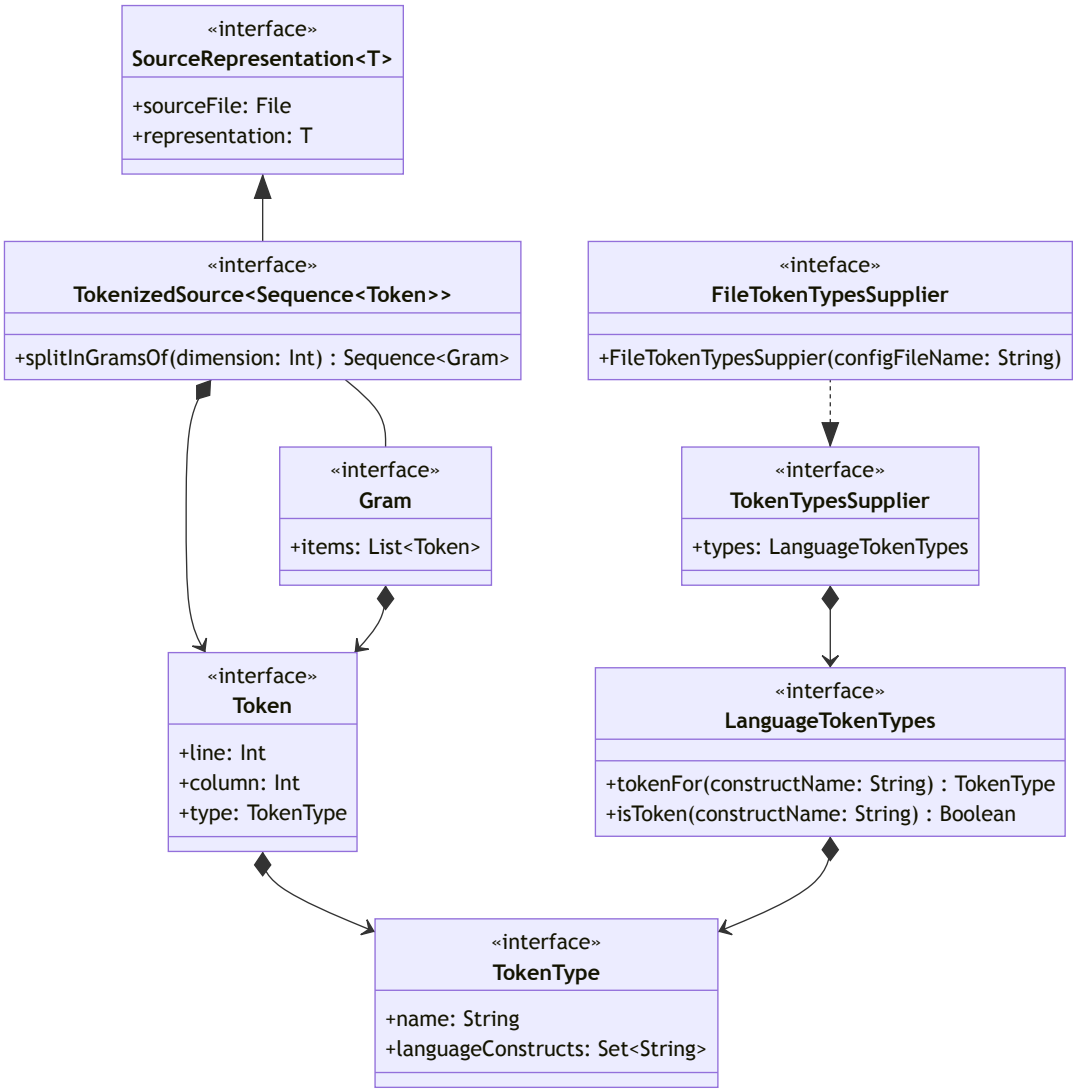

Figura 2.7: Schema UML delle rappresentazioni intermedie.

Le rappresentazioni generate dal processo di analisi, vengono poi opportunamente filtrate per mezzo di un `RepresentationFilter` che, preso in *input* una istanza dell'insieme delle *submission* e l'insieme del *corpus* con cui confrontarla, ne effettua un filtraggio secondo opportune metriche. Per farlo, fa uso di un `Inde-`

`xer` che crea, a partire dalle rappresentazioni, indici (strutture dati) su cui poter calcolare stime di similarità. Uno schema di massima è presentato in Figura 2.8

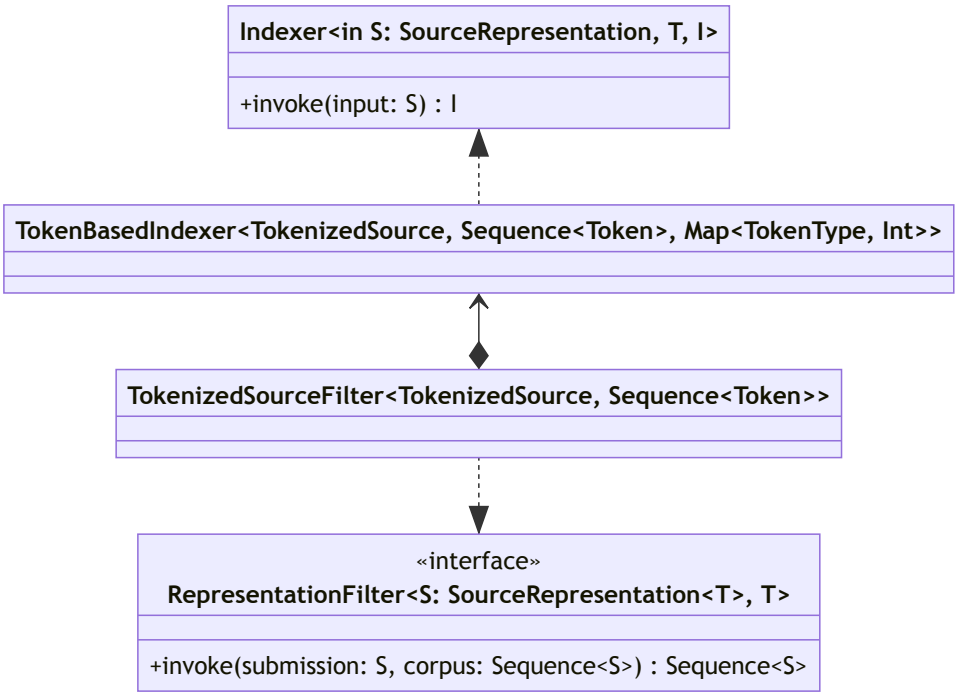

Figura 2.8: Schema UML del filtro.

`PlagiarismDetector` (Figura 2.9) rappresenta invece il componente che individua le similarità tra una coppia di `SourceRepresentation` e si occupa di calcolare la loro similarità. Poiché vi sono una molteplicità di algoritmi presenti in letteratura che assolvono a questo compito, si è deciso d'implementarli sotto forma di gerarchia di classi di algoritmi il cui contratto comune è definito da `ComparisonStrategy`. In particolare, ogni classe implementante `ComparisonStrategy` genera un `Match`, dove per `Match` si intende due sezioni simili di `SourceRepresentation`. L'insieme dei `Match` che compongono la comparazione di due `SourceRepresentation` sono raggruppate in un `ComparisonResult`, il cui grado di similarità è stimato dalla `SimilarityEstimationStrategy`.

Nel contesto della *tokenizzazione*, i due concreti algoritmi che sono implementati come strategia di confronto sono rappresentati dalle sottoclassi `GreedyStringTiling` e `RKRGreedyStringTiling`. Entrambi generano dei `TokenMatch`, costituiti da sequenze contigue di *token* dello stesso tipo. A partire da queste la classe concreta `TokenBasedPlagiarismDetector` demanda alle concrete implementazioni di `TokenBasedSimilarityStrategy` il calcolo della similarità.

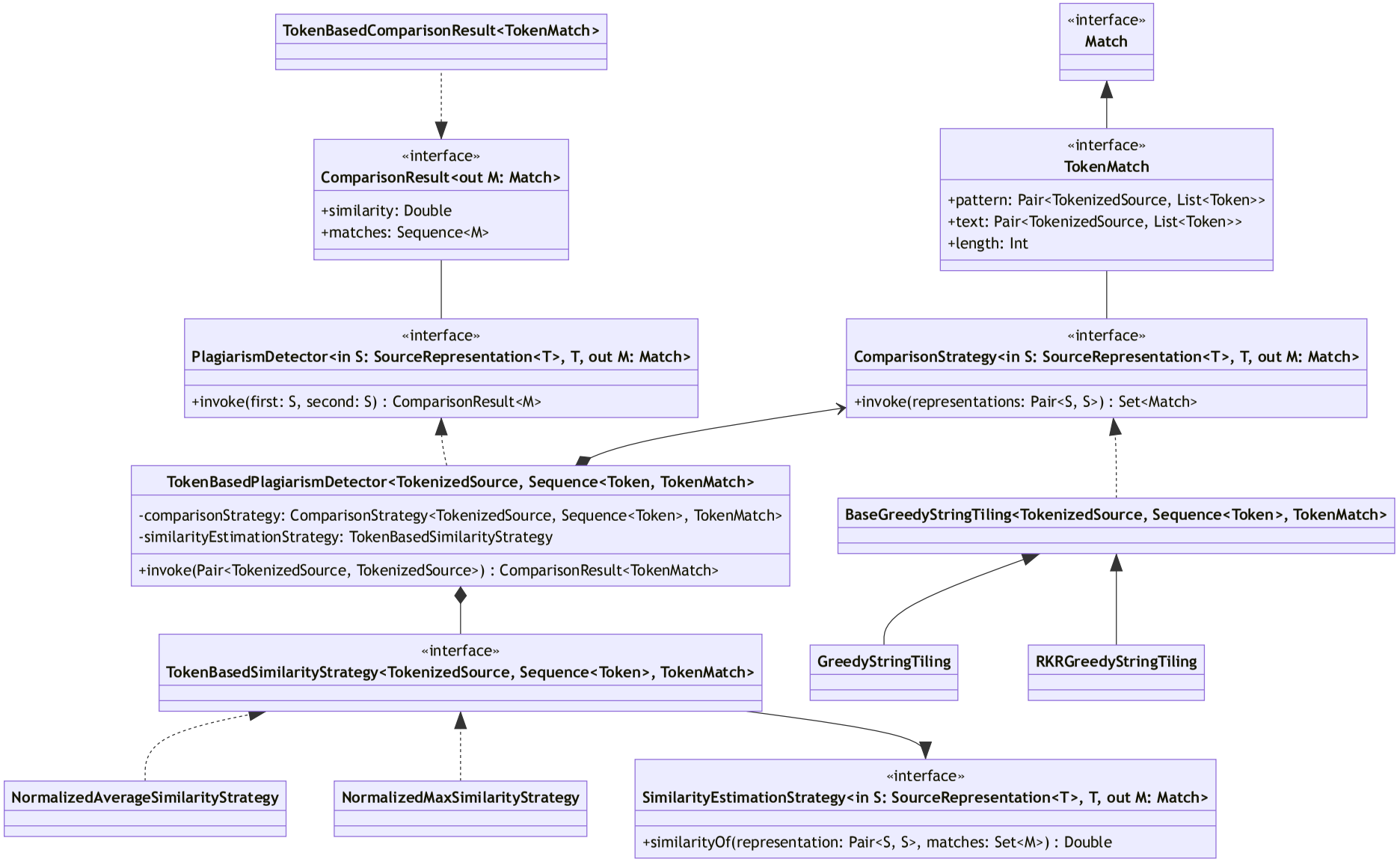

Figura 2.9: Schema UML del *detector*.

Tutti e tre i componenti, `Analyzer`, `Filter` e `PlagiarismDetector`, sono incapsulati all'interno di un concreto `TechniqueFacade` [EV44]. Si è deciso di adottare questo approccio principalmente per due motivi:

- la specifiche strategie di analisi e di confronto devono essere istanziate e opportunamente configurate a *runtime* a seconda del tipo di tecnica e delle opzioni scelte dall'utente. Inoltre, devono essere coerenti tra loro: non è possibile, ad esempio, poter istanziare un analizzatore per effettuare la *tokenizzazione* e un rilevatore che non operi sulla sequenza di *token*. Il *facade* pertanto si occupa d'istanziare i corretti componenti necessari all'esecuzione della tecnica in accordo con i parametri a lui fornitogli, sollevando il *client* da questa responsabilità.
- Il processo di analisi e confronto può essere complesso. Pertanto si fornisce ai *client* un'interfaccia semplificata che assolve al compito di eseguire una generica tecnica, rendendo di fatto trasparente al chiamante la complessità del sottosistema che, dal suo punto di vista, è di fatto una *black box*.

Configurabilità, Input e Output

Come già detto, l'aspetto della configurabilità è un aspetto molto importante in questo contesto, in quanto la scelta dei parametri influisce sull'efficacia del rilevamento. Come già presentato nell'architettura, una specifica sessione è legata ad una particolare configurazione (`RunConfiguration`) che contiene tutte le informazioni necessarie per l'esecuzione della stessa. A differenza di quanto presentato nella Sezione 2.3.1, giacché per i motivi suddetti `Analyzer`, `Filter` e `Detector` sono stati incapsulati all'interno della `TechniqueFacade`, la configurazione non ha un riferimento diretto a ciascuno dei tre componenti, bensì all'oggetto *facade*.

L'attuale `ReportsExporter` implementato esporta i dati in un semplice *file* di testo e i `RunConfigurator` attualmente supportati sono il `CLIConfigurator`, che crea la configurazione a partire dagli argomenti da riga di comando, oppure il `FileConfigurator` che sfrutta un file di configurazione per impostare tutti i parametri. In futuro, il design potrebbe essere esteso per supportare un'interfaccia grafica per la configurazione o per generare *report* più ricchi in termini di presentazione (ad esempio un file *HTML*).

In Figura 2.10 è mostrato lo schema UML delle classi che mostra complessivamente i rapporti tra le varie componenti.

Comportamento

Per concludere, in Figura 2.11 viene mostrato il diagramma UML di sequenza che mostra macroscopicamente il flusso delle azioni del sistema e come le varie componenti cooperano tra loro.

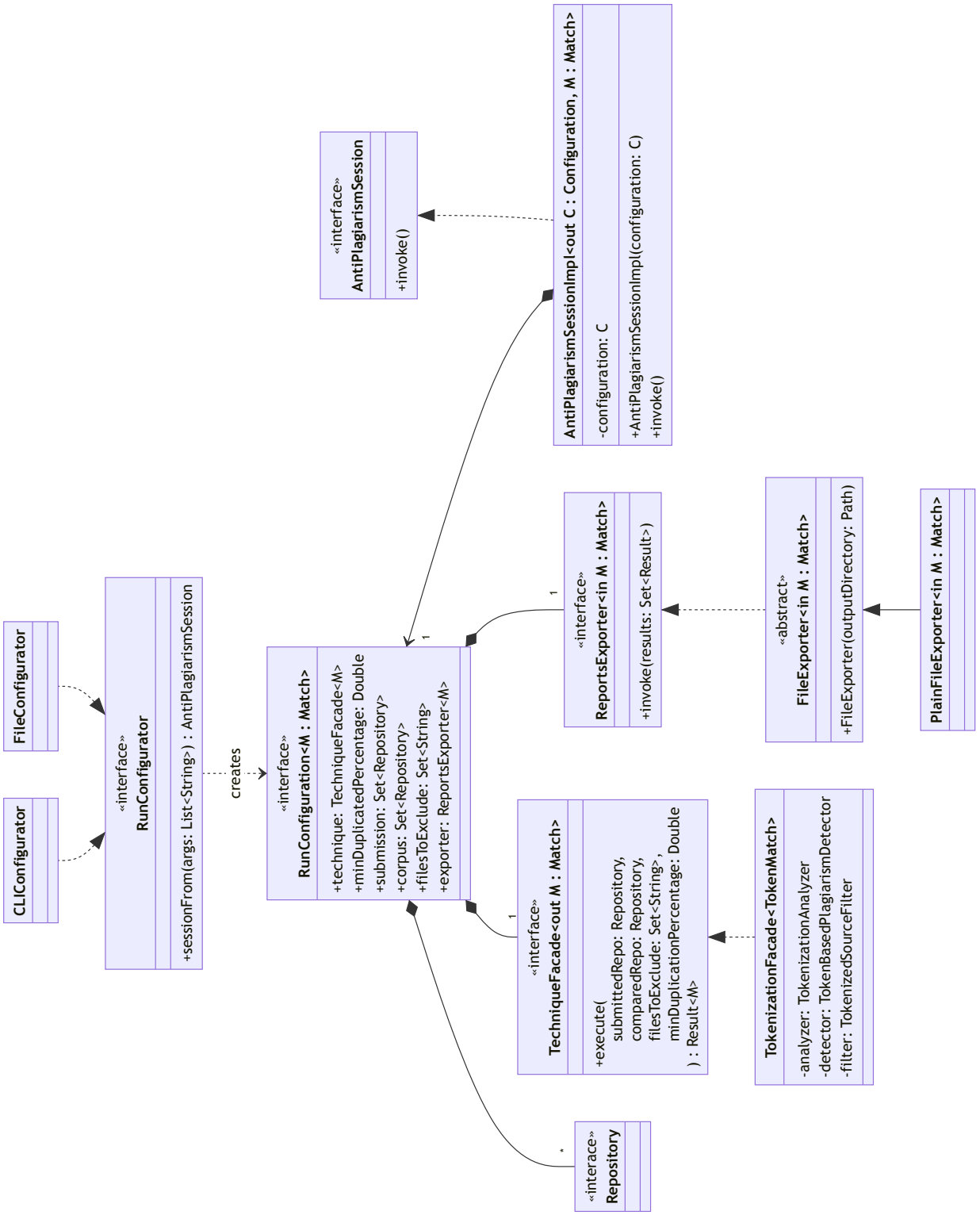

Figura 2.10: Schema UML della sessione.

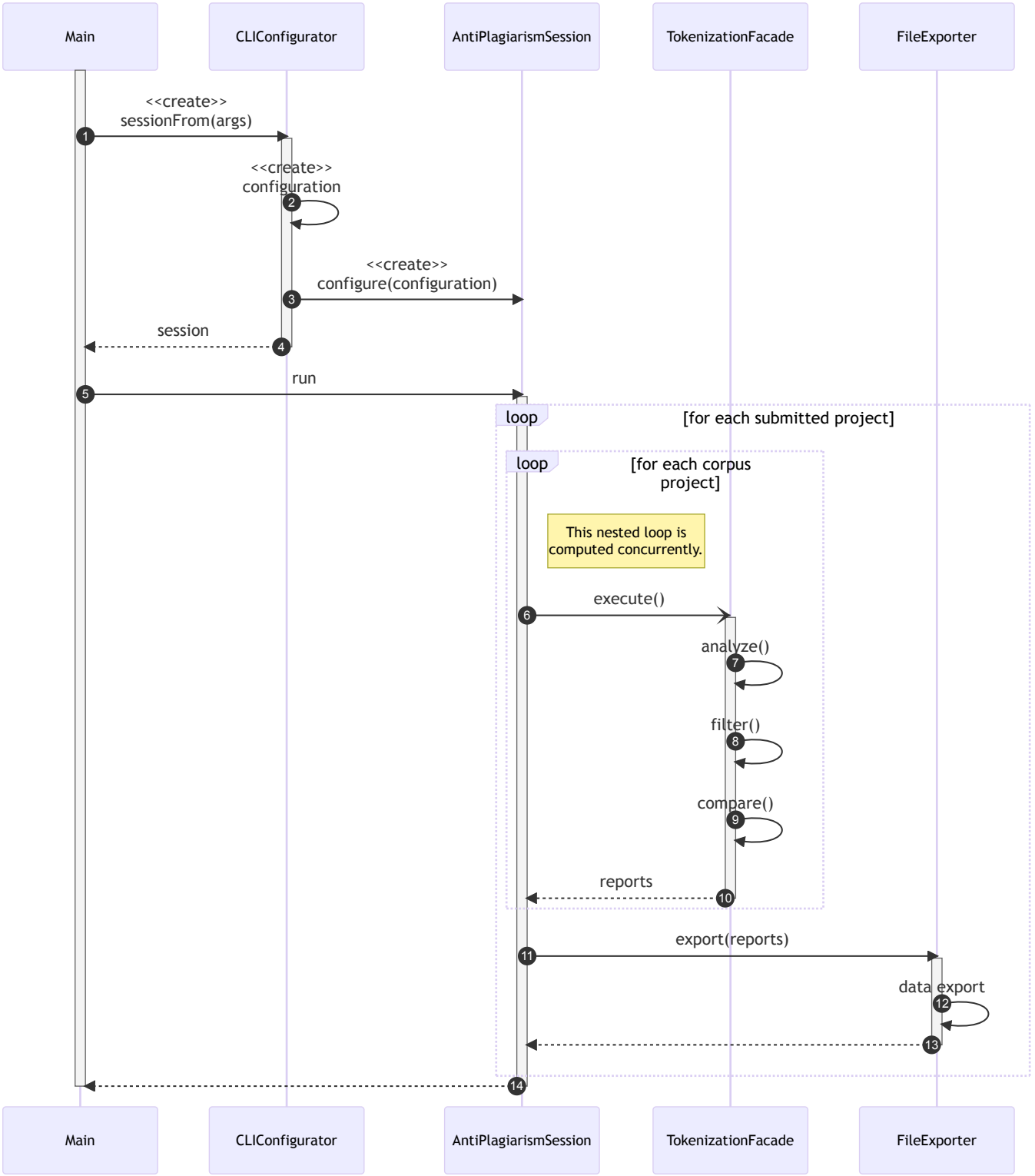

Figura 2.11: Diagramma UML di sequenza rappresentante l'interazione tra le componenti del sistema.

Capitolo 3

Implementazione

In questo capitolo vengono affrontati gli aspetti implementativi del sistema, descrivendo le tecnologie e i paradigmi utilizzati.

La tecnica impiegata nel sistema appartiene alla famiglia di tecniche *structure-based* e si compone macroscopicamente di tre fasi, schematizzate in Figura 3.1:

1. **Analisi**: i sorgenti sono *parsati*, preprocessati e *tokenizzati*;
2. Una fase opzionale di **filtraggio**;
3. **Rilevamento** delle somiglianze.

Di seguito ciascuna delle tre viene approfondita.

3.1 Tecnica di analisi

Il primo passo consiste nell'effettuare il *parsing* dei sorgenti, affidato alla libreria JavaParser[1]. JavaParser è una libreria *open source* che permette di effettuare il *parsing* di sorgenti scritti in linguaggio Java, fornendo comodi meccanismi per l'analisi e la manipolazione degli stessi.

Il risultato del *parsing* è un albero sintattico equivalente a quello mostrato in Figura 1.5 che rappresenta la struttura dei sorgenti.

Tuttavia, nonostante la generazione dell'albero sintattico sollevi dalla responsabilità di eliminare dal sorgente *token* superflui, come gli spazi e i punti e virgola, in quanto naturalmente modellati dalla struttura stessa dell'albero, tale AST contiene ancora informazioni sovrabbondanti, come le dichiarazioni di `import` e dei `package`. Per questo motivo è necessario una fase intermedia di *preprocessing* in cui l'albero viene "sfoltito". In particolare, in questa fase l'albero viene visitato e, oltre a rimuove le dichiarazioni suddette, sono rimosse anche le funzioni

[1] `https://javaparser.org/`

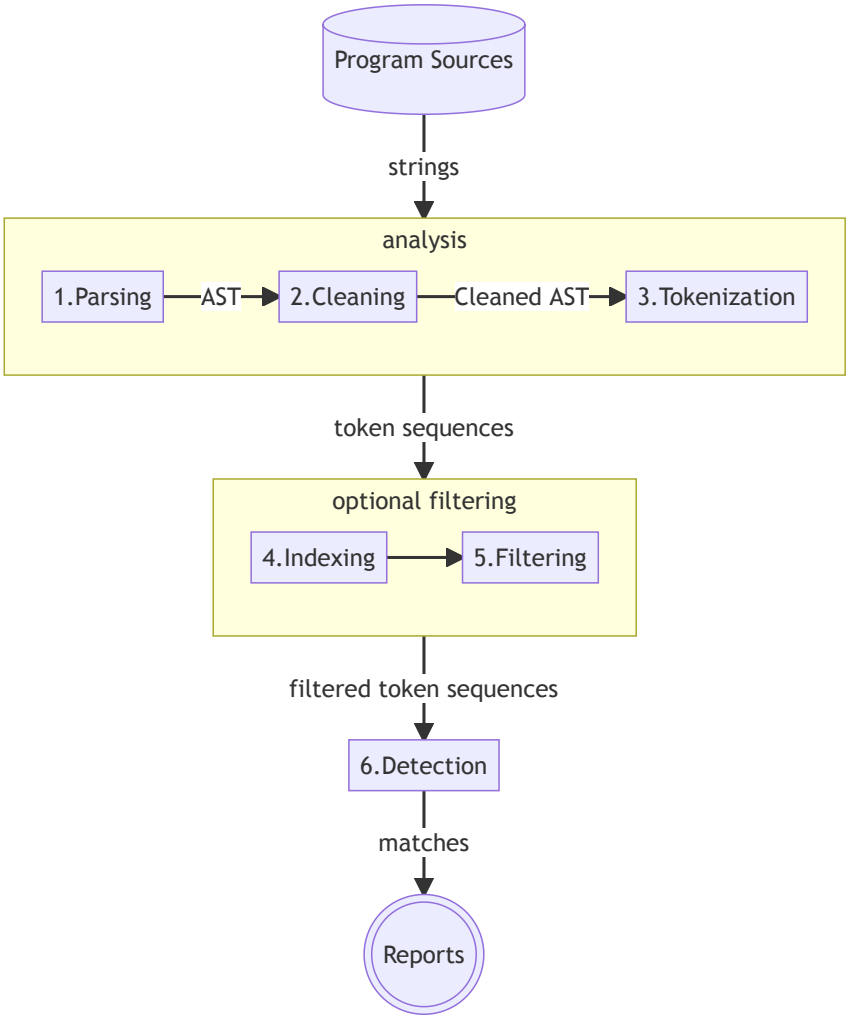

Figura 3.1: Visione d'insieme delle fasi della tecnica impiegata.

`hashCode()`, `equals()` e `toString()`. Queste, infatti, nella maggioranza dei casi, vengono automaticamente generate dall'*IDE* e non sono rappresentative della logica dei sorgenti.

A seguire avviene la conversione del codice sorgente in una sequenza di *token*. Si noti che la fase di *tokenizzazione* vera e propria descritta nella Sezione 1.3.2 è già stata eseguita internamente dal *parser* di JavaParser i cui *token* sono quelli presenti nell'albero. Tali *token*, tuttavia, rappresentano in modo molto specifico la struttura del programma: ad ogni tipo del linguaggio è associato un tipo differente di *token*. Come descritto nella Sezione 1.3.2 l'obiettivo è creare invece un insieme di *token* "lasco" che raggruppi tipi di *token* equivalenti a livello semantico.

A questo scopo, l'albero sintattico preprocessato viene visitato e, per ogni nodo, si emette un *token*, con le relative informazioni circa la sua posizione nel sorgente, a seconda sia una dichiarazione semanticamente rilevante o meno.

In particolare, è stato costruito un file di configurazione *Yaml*[2] (Listato 3.1) in cui sono elencati i tipi di dichiarazioni rilevanti, eventualmente tralasciando tipi troppo dettagliati, come `PrimitiveType` o `VarType`, ed aggregando tra loro dichiarazione semanticamente simili e che potrebbero essere facilmente cambiate per offuscare la copiatura. Ad esempio, sono state aggregate sotto un unico tipo `loop-stmt` tutti i costrutti che effettuano un ciclo. In questo modo un cambio sintattico di tipo non sarà sufficiente ad aggirare il sistema perché verrà generato lo stesso tipo di *token*. Si noti che questo approccio è molto flessibile e può essere "aggiustato" a piacimento nel caso sia necessario effettuare una *tokenizzazione* con un insieme di *token* più stringente.

Listato 3.1: Porzione del file di configurazione con la definizione dei tipi di *token*.

```
# DECLARATIONS
- name: class-interface-decl
  languageConstructs:
    - ClassOrInterfaceDeclaration
- name: constructor-decl
  languageConstructs:
    - ConstructorDeclaration
    - CompactConstructorDeclaration
...
# EXPRESSIONS
- name: literal-expr
  languageConstructs:
    - DoubleLiteralExpr
    - IntegerLiteralExpr
    - LongLiteralExpr
    - NullLiteralExpr
    - CharLiteralExpr
    - StringLiteralExpr
    - TextBlockLiteralExpr
- name: class-expr
  languageConstructs:
    - ClassExpr
- name: lambda-expr
  languageConstructs:
    - LambdaExpr
- name: annotation-expr
  languageConstructs:
    - MarkerAnnotationExpr
```

[2]Linguaggio, *superset* di Json, per la serializzazione di dati che viene impiegato per la scrittura di file di configurazione: `https://yaml.org/`.

```
    - NormalAnnotationExpr
    - SingleMemberAnnotationExpr
- name: variable-decl-expr
  languageConstructs:
    - VariableDeclarationExpr
...
# STATEMENTS
- name: loop-stmt
  languageConstructs:
    - ForEachStmt
    - ForStmt
    - WhileStmt
    - DoStmt
- name: return-stmt
  languageConstructs:
    - ReturnStmt
...
```

Il risultato dell'applicazione del processo sopra descritto al Listato 1.1 viene presentato nel Listato 3.2.

Listato 3.2: Risultato dell'analisi del Listato 1.1

```
[class-interface-decl (line=8, column=1),
method-decl (line=9, column=5),
parameter (line=9, column=29),
block-stmt (line=9, column=44),
if-stmt (line=10, column=9),
binary-expr (line=10, column=13),
field-access-expr (line=10, column=13),
name-expr (line=10, column=13),
literal-expr (line=10, column=27),
block-stmt (line=10, column=30),
expression-stmt (line=11, column=13),
method-call-expr (line=11, column=13),
field-access-expr (line=11, column=13),
name-expr (line=11, column=13),
binary-expr (line=11, column=32),
literal-expr (line=11, column=32),
method-call-expr (line=11, column=56),
name-expr (line=11, column=56),
name-expr (line=11, column=72),
block-stmt (line=12, column=16),
expression-stmt (line=13, column=13),
method-call-expr (line=13, column=13),
field-access-expr (line=13, column=13),
name-expr (line=13, column=13),
literal-expr (line=13, column=32)]
```

3.2 Filtraggio

A seguito dell'analisi del codice sorgente e della generazione della sequenza di *token* i sorgenti del *corpus* possono essere filtrati, al fine di limitare il numero di progetti da dover confrontare. Questo *step* rappresenta uno snodo critico nella *pipeline* di azioni da eseguire in quanto scambia una frazione dell'efficacia del sistema in favore dell'efficienza: una stima fuorviante della similarità di una coppia di progetti può comportare l'esclusione dalla comparazione degli stessi, con la conseguente perdita di sensibilità nel caso in cui rappresentassero dei veri casi di copiatura. Per questo motivo la fase di filtraggio è *opzionalmente* eseguita a discrezione dell'utente, che si presume possa valutare se necessita di un'analisi più veloce ma meno esaustiva o viceversa.

Come già presentato nella Sezione 1.3.2, il processo di filtraggio è preceduto da una fase d'indicizzazione in cui vengono aggregati i dati sotto forma di una struttura dati per mezzo della quale è possibile estrarre informazioni statistiche significative per la stima della similarità.

Nel sistema è attualmente implementato un indice equivalente a quello presentato in Tabella 1.3 a partire dal quale la similarità tra progetti è stata calcolata mediante la **similarità coseno**. Essa è una metrica euristica ampiamente utilizzata nel contesto dell'analisi testuale che misura la similitudine di due vettori calcolando il coseno del loro angolo:

$$cosine\_similarity(A, B) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{k=1}^{n} A(k)B(k)}{\sqrt{\sum_{k=1}^{n} A(k)^2 \cdot \sum_{k=1}^{n} B(k)^2}} \tag{3.1}$$

dove A e B sono due vettori di attributi numerici a n dimensioni.

In generale, il risultato della similarità è un valore compreso tra -1 e 1 dove -1 indica che i due vettori sono *anti*-correlati (hanno verso opposto), 1 indica massima correlazione e 0 un'assenza di correlazione (i due vettori sono ortogonali tra loro).

Nel nostro caso, il contenuto dei due vettori è la frequenza dei tipi di *token* in cui il k-esimo elemento contiene il numero di volte in cui il k-esimo tipo di *token* ricorre nel sorgente, oppure 0 se non presente. Poiché le frequenze sono valori sempre positivi, nel caso in esame, si otterranno valori compresi tra 0 e 1, dove 1 indica che i tipi di *token* contenuti nelle due rappresentazioni sono gli stessi, presenti in egual numero ma non necessariamente disposti nello stesso ordine, e 0 indica che non c'è alcun tipo di *token* comune.

A seguito della stima della similarità coseno, tutte le coppie di rappresentazioni la cui similarità è inferiore a un valore di soglia sono esclusi. Tale valore è calcolato, seguendo quanto riportato in [SK19], utilizzando la formula riportata nell'Equazione (3.2).

$$threshold = sim_{min} + init\_threshold \cdot (sim_{max} - sim_{min}) \tag{3.2}$$

dove $init\_threshold$ è un valore compreso tra 0 e 1 (corrispondente al range $0 - 100\%$) e sim_{max} e sim_{min} sono rispettivamente la similarità coseno massima e minima risultante dalla comparazione di tutte le coppie.

Secondo le valutazioni presentate in [SK19] questo tipo di filtraggio aumenta l'efficienza del sistema con un compromesso in termini di efficacia che rimane accettabile fintanto che la soglia $init\_threshold$ è mantenuta al di sotto del 60%.

3.3 Rilevamento delle somiglianze

Gli algoritmi impiegati per confrontare due sequenze di *token* sono il *Greedy String Tiling* e il *Running Karp-Rabin Greedy String Tiling*, che ne rappresenta una sua evoluzione, introdotti da M. Wise nel 1993 in [Wis93] a seguito della necessità di sviluppare proprio un sistema automatico antiplagio, anche se negli anni successivi gli stessi algoritmi sono stati impiegati in altri campi, tra cui il confronto di sequenze di proteine/DNA.

Sebbene questi siano stati concepiti per lavorare su sequenze di stringhe, possono essere facilmente riadattati per operare su sequenze di *token*

Dette A e B due sequenze di *token*, un algoritmo per misurare la similarità in questo dominio deve determinare una sottosequenza comune che abbia le seguenti proprietà:

- ogni *token* di A può essere abbinato solo con esattamente un solo *token* di B;
- le sottosequenze comuni devono essere trovate indipendentemente dalla loro posizione nel sorgente;
- le sottosequenze più lunghe sono preferite a quelle più piccole, in quanto le sottosequenze brevi è molto probabile rappresentino casi spuri.

L'algoritmo si compone principalmente di due fasi:

- **Fase 1**: in questa fase, viene ricercata la corrispondenza più lunga. Questo è fatto grazie un triplo ciclo innestato: il primo itera sui token della sequenza più corta, denominata *pattern*, il secondo compara ciascuno di questi con ogni *token* della sequenza più lunga, denominata *text*. Se i due *token* corrispondono il ciclo più interno cerca di estendere il *match* il più possibile (fermandosi non appena trova un *token* che nelle due sequenze differisce).
- **Fase 2**: questa fase marca ciascun *match* di lunghezza massima (*maximal match*) trovato nella fase precedente. Questo garantisce che ciascun *token* venga usato per un solo *match* e divenga indisponibile per i *match* successivi

(che hanno una minor lunghezza). Nella terminologia di [Wis93] i *match* i cui *token* sono stati marcati sono denominati *tile*. Si noti che, anche se un *match* è parzialmente occluso e perciò non viene marcato, una versione più corta dello stesso *match* può essere marcato durante le successive iterazioni.

Queste due fasi vengono ripetute fino a quando non vengono trovate più corrispondenze di lunghezza almeno `minimum_match_length`. Tale valore è imposto per non generare troppe sequenze spurie di lunghezza irrisoria e garantire risultati migliori. Giacché la lunghezza dei *match* diminuisce ad ogni iterazione è garantito che l'algoritmo termini.

Lo pseudocodice dell'algoritmo è presentato nel Listato 3.3.

Listato 3.3: Pseudocodice dell'algoritmo *Greedy String Tiling (GST)*. In accordo con la terminologia del *paper*, P rappresenta il *pattern*, ovvero la sequenza da confrontare più corta, mentre T il *text*, ovvero la sequenza tra le due più lunga.

```
1  length_of_tokens_tiled := 0
2  Repeat
3    max_match := minimum_match_length
4    starting at the first unmarked token of P, foreach P_p do
5      starting at the first unmarked tokne of T, foreach T_t do
6        j := 0
7        while P_{p+j} = T_{t+j} ∧ unmarked(P_{p+j}) ∧ unmarked(T_{t+j}) do
8          j := j + 1
9          if j = max_match then add match(p,t,j) to list of matches of
               length j
10         else if j > max_match then start new list with match(p,t,j)
               and max_match = j
11   for each match(p,t, max_match) in list
12     if not occluded then
13       for j := 0 to max_match - 1 do
14         mark_token(P_{p+j})
15         mark_token(T_{t+j})
16       length_of_tokens_tiled := length_of_tokens_tiled + max_match
17 Until max_match = minimum_match_length
```

Questo algoritmo è dimostrato [Wis93] essere ottimo in termini di massimizzazione della copertura delle stringhe. Nonostante ciò, nel caso peggiore, ha una complessità $O(n^3)$ e pertanto è molto inefficiente.

Per far fronte a questo problema è stato ideato un nuovo algoritmo chiamato *Running Karp Rabin Greedy String Tiling*. L'idea su cui si fonda per velocizzare il confronto è impiegare una funzione di *hashing*. In particolare, il valore di *hash* di ogni sottosequenza di lunghezza s del *pattern* viene confrontato con il valore di *hash* di quelle del *text*. Laddove i due valori corrispondano, e solo in questo caso, le sottosequenze vengono confrontate *token* per *token* per assicurarsi che effettivamente siano uguali e non frutto di una collisione della funzione di *hash*.

Nel Listato 3.4 viene mostrato lo pseudocodice della *routine* che si occupa di effettuare questo confronto. Come si può osservare, nelle righe 1-3 vengono calcolati i valori di *hash* di tutte le sottosequenze (non marcate) di lunghezza s del *text* e vengono salvati in una *hashtable*, una mappa con le corrispondenze sottosequenza di *token*-valore di *hash*. Lo stesso viene fatto per il *pattern* con la differenza che, non appena calcolato il valore *hash* per la sottosequenza considerata, si verifica se lo stesso sia già presente nella *hashtable*. In questo caso, infatti, potrebbe effettivamente esserci una corrispondenza e si procede verificando *token* per *token*. Si noti che nei casi in cui la corrispondenza sia effettiva e non sia causata da un artefatto della funzione di *hash*, il *match* è esteso fino al primo *token* non corrispondente, proprio come nel precedente algoritmo.

Listato 3.4: Pseudocodice della *routine* `scanpattern` dell'algoritmo *RKR-GST*.

```
1  Starting at the first unmarked token of T, foreach unmrked T_t do
2      if distance to next tile ≤ s then advance t to first unmarked
           token after next tile
3      else create the KR hash-value for substring T_t to T_{t+s-1} and add
           to hashtable
4  Starting at the first unmarked token of P, for each unmarked P_p do
5      if distance to next tile ≤ s then advance p to first unmarked
           token after next tile
6      else
7          create the KR hash-value for substrng P_p to P_{p+s-1}
8          check hashtable for hash of KR hash-value
9          for each hash-table entry with equal hashed KR hash-value
               do
10             if for all j from 0 to s-1, P_{p+j} = T_{t+j} then
11                 /* i.e. match is not hash artifact */
12                 k := s
13                 while P_{p+k} = T_{t+k} ∧ unmarked(P_{p+k}) ∧ unmarked(T_{t+k})
14                     k := k + 1
15                 if k > 2 · s then
16                     /* restart scanpattern with s = k */
17                     return(k)
18                 else record new maximal-match
19 return(length of longest maximal-match)
```

Nella versione originale dell'algoritmo la struttura dati utilizzata per memorizzare i *match* è una lista di code in cui ogni coda memorizza i *match* della stessa lunghezza, mantenuta ordinata per valori decrescenti.

Il Listato 3.5 presenta lo pseudocodice della *routine* che effettua la marcatura dei *match* individuati nella *routine* `scanpattern`, corrispondente alla seconda fase dell'algoritmo precedente. La principale differenza con la precedente versione risiede nel fatto che la marcatura viene effettuata per ogni *match* e non solo per quelli di lunghezza massima. Inoltre, se il *match* risulta parzialmente occluso, ma

la rimanente parte non occlusa è di lunghezza $m \geq s$, questa viene salvata nella lista di code, in modo tale da essere valutata nella successiva iterazione in cui vengono valutati tutti i *match* di lunghezza m (righe 11-13).

Listato 3.5: Pseudocodice *routine* `markarrays` dell'algoritmo *RKR-GST*.

```
1  Starting with the top queue, while there is a non-empty queue do
2      if the current queue is empty then drop to next queue /*
           corresponding to smaller maximal-matches */
3      else
4          /* Assume the length of maximal-matches in the current
               queue is L */
5          remove match(p, t, L)
6          if match is not occluded then
7              for j := 0 to L − 1 do
8                  mark_token(P_{p+j})
9                  mark_token(T_{t+j})
10             length_of_tokens_tiles := length_of_tokens_tiled + L
11         else if L − L_occluded ≥ s then
12             /* i.e. the unmarked part remaining of the maximal-
                   match */
13             record unmarked portion on list of queues
```

Infine, nel Listato 3.6 è mostrato l'*entry-point* dell'algoritmo. Come si può osservare, corrispondenze molto lunghe vengono trovate solo durante la prima iterazione: dopo di questa, infatti, non è possibile siano trovate corrispondenze maggiori del doppio della lunghezza di ricerca corrente. Questo garantisce che, nelle successive iterazioni, l'algoritmo termini.

Listato 3.6: *Entry-point* dell'algoritmo *RKR-GST*.

```
1  Search-length s := initial-search-length
2  stop := false
3  Repeat
4      /* Lmax is the size of the largest maximal-matches found in
           this iteration */
5      L_max := scanpattern(s)
6      if L_max > 2 · s then
7          /* Very long string: don't mark tiles but try again with
               larger s */
8          s := L_max
9      else
10         /* Create tiles from matches takes from list of queues */
11         markarrays(s)
12         if s > 2· min_match_length then s := / 2
13         else if s > min_match_length the s := min_match_length
14         else stop := true
15 until stop
```

Il valore di s (l'equivalente di `minimum_match_length` nell'algoritmo *GST*) è scelto in modo tale da garantire che non vengano cercate corrispondenze troppo piccole. Un valore plausibile di s consigliato in [Wis93] è 20.

Sebbene la complessità di questo algoritmo sia dimostrato essere, nel caso peggiore, $O(n^2)$, nel caso medio la complessità è pressoché **lineare**. Ciò migliora notevolmente le prestazioni rispetto all'algoritmo *Greedy String Tiling*.

3.4 Stima della similarità

Similarità tra coppie di sorgenti

La misura della similarità di due sorgenti dovrebbe intuitivamente riflettere la frazione di *token* dei programmi originali che sono "coperti" da *match*. Seguendo questo principio ci sono due possibili scelte che possono essere intraprese e che determinano la formula di stima di similarità: se si vuole che la similarità del 100% significhi che le due sequenze sono identiche dobbiamo considerare entrambe le stringhe nel computo, come segue:

$$sim_S(A, B) = \frac{2 \cdot \sum_{match \in tiles} length}{|A| + |B|} \tag{3.3}$$

Se, invece, preferiamo che ogni programma che è stato copiato completamente (e poi magari esteso) produca una similarità del 100%, dobbiamo considerare solo la sottosequenza più corta:

$$sim_S(A, B) = \frac{\sum_{match \in tiles} length}{min(|A|, |B|)} \tag{3.4}$$

Per mitigare il problema dell'insensibilità al riordino delle sottosequenze di cui è intrinsecamente affetta la tecnica di analisi impiegata, in [SK19] viene proposto di modificare la formula di stima della similarità introducendo un meccanismo di penalità ispirato dal principio che, solitamente, un numero elevato di sottosequenze comuni si verifica proprio a causa del riordino delle stesse. Applicando pertanto una penalità proporzionata al numero di sottosequenze comuni, maggiore sarà il numero di sottosequenze individuate, minore sarà il grado di similarità determinato. Le formule aggiornate sono di seguito presentate.

$$sim_S(A, B) = \frac{2 \cdot \left(\sum_{match \in tiles} length - (|tiles| - 1)\right)}{|A| + |B|} \tag{3.5}$$

$$sim_S(A, B) = \frac{\sum_{match \in tiles} length - (|tiles| - 1)}{min(|A|, |B|)} \tag{3.6}$$

Similarità tra coppie di progetti

Vista la grande quantità di progetti di cui si dispone, sebbene i risultati ottenuti siano filtrati per non riportare singole coppie di sorgenti con una similarità inferiore a una soglia (scelta dall'utente), la quantità di risultati rimane corposa. Pertanto, non è sufficiente stimare la similarità sorgente per sorgente, ma è necessario fornire una stima aggregata delle similarità progetto per progetto, così da individuare più facilmente, a un livello di granularità maggiore, i progetti simili.

In generale, questa stima di similarità è inferita a partire da quelle tra sorgenti come segue:

$$sim_P(A,B) = \underbrace{max\left\{\frac{|sorgenti\ segnalati\ di\ A|}{|sorgenti\ di\ A|}, \frac{|sorgenti\ segnalati\ di\ B|}{|sorgenti\ di\ B|}\right\}}_{\text{I}} \cdot \underbrace{P_{75}(similarità\ sorgenti\ segnalati)}_{\text{II}} \tag{3.7}$$

dove A e B sono due progetti e P_{75} il settantacinquesimo percentile (o terzo quartile) delle similarità dei sorgenti segnalati.

In questo modo, la seconda parte dell'equazione (II) fornisce una stima aggregata delle similarità dei sorgenti, considerando il valore di similarità che "lascia alla sua sinistra" il 75% della distribuzione. L'utilizzo di questa statistica è migliore della semplice media o mediana, in quanto queste darebbero stessa importanza a tutte le stime di similarità, fornendo risultati troppo bassi anche nei casi effettivi di copiatura, visto che, generalmente, anche in questi, vengono rintracciate sezioni con una bassa stima di similarità.

Tuttavia, applicare solo una statistica di aggregazione dei valori di similarità dei sorgenti segnalati può fornire misure fuorvianti nel caso in cui questi siano pochi (nell'ordine di alcune unità) ma con valori di similarità elevati. Per questo motivo, la misura di similarità aggregata sopra descritta viene "pesata" sulla base del massimo rapporto tra A e B del numero di sorgenti segnalati e quelli del progetto in esame. Così facendo, si andrà a diminuire il peso complessivo della similarità per quei progetti i cui sorgenti segnalati sono esegui rispetto alla loro totalità. Questo determina tuttavia che copiature di circa il 50% del progetto possano avere al più similarità 50% nel caso limite in cui la II parte dell'Equazione (3.7) sia uguale a 1. Questo però è raro che accada perché significherebbe che le copiature sono frutto di un copia e incolla letterale; nella realtà tale valore sarà strettamente minore di uno e ciò comporterebbe una diminuzione significativa del valore di similarità per la coppia di progetti che in definitiva verrebbe classificata come non copiata nonostante la copiatura di metà progetto. Per questo motivo, un possibile miglioramento all'Equazione (3.7) consiste nell'applicare alla I parte dell'equazione

una funzione di peso, come quella descritta in Equazione (3.8) il cui grafico è mostrato in Figura 3.2.

$$w(x) = \begin{cases} \frac{1}{2}sin\big(4,5 \cdot x - \frac{\pi}{2}\big) + \frac{1}{2} & \text{se } 0 \leq x < 0,7 \\ 1 & \text{se } x >= 0,7 \end{cases} \tag{3.8}$$

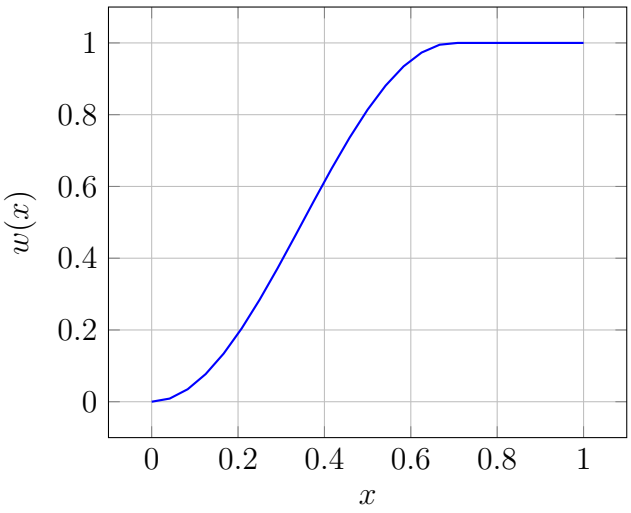

Figura 3.2: Grafico della funzione di pesatura della I parte dell'Equazione (3.8).

Come si può osservare, applicare questa semplice funziona alla parte I dell'Equazione (3.7) garantisce di attenuare il contributo dove il rapporto tra sorgenti riportati e sorgenti totali è esiguo ed enfatizzare i rapporti più elevati.

3.5 Strumenti di sviluppo

Kotlin

Il progetto è stato sviluppato interamente in *Kotlin*, un linguaggio di programmazione **orientato agli oggetti** e **funzionale** *open source*[3] progettato e gestito da *JetBrains*[4] a partire dal 2010. Si tratta di un linguaggio multipiattaforma compilato, tipato staticamente e completamente interoperabile con *Java* e la *JVM* e *Android*, applicabile in una vasta gamma di ambiti, dallo sviluppo *server side* a quello di applicazioni mobili (attualmente è il linguaggio di riferimento per *Android*, avendo surclassato Java). Inoltre, oltre a Java e Android, *Kotlin* può essere compilato in *JavaScript*, consentendo di eseguire codice *Kotlin* nel *browser*.

[3] `https://github.com/JetBrains/kotlin`
[4] `https://www.jetbrains.com/`

L'obiettivo principale del linguaggio è fornire un'alternativa a Java più concisa, produttiva e sicura, adatta in tutti i contesti in cui è tutt'oggi utilizzato. Infatti, essendo completamente compatibile con Java, sfrutta tutte le sue numerose librerie, estendendole e garantendo le stesse *performance*.

I tratti distintivi di *Kotlin* sono certamente la sua espressività e concisione, che permettono di sviluppare codice che rivela l'intento senza oscurarlo con codice verboso per specificare come viene realizzato.

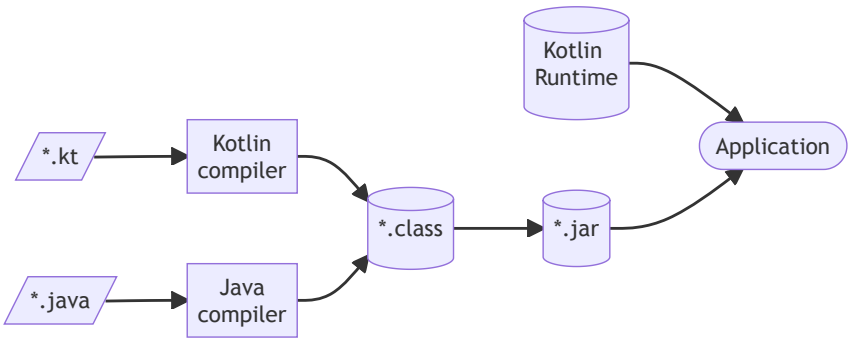

Figura 3.3: Processo di compilazione di *Kotlin*.

Git e *CI/CD*

Per lo sviluppo del progetto si è utilizzato Git, uno *standard de facto* nell'ambito dei sistemi di controllo di versione decentralizzato, che permettere di tenere traccia dello storico delle modifiche e lo sviluppo simultaneo da parte di più sviluppatori, senza che questi debbano sobbarcarsi la gestione dell'intero progetto a mano, con tutti i conseguenti problemi.

In particolare, il *workflow* adottato è il seguente: il progetto è mantenuto in una *repository* pubblica centrale[5] e ciascun sviluppatore lavora su una *working copy* (*fork*). Non appena una *feature* è completata o si arriva a un buon grado di sviluppo si aprono delle *pull request*, nelle quali i contributi sono revisionati per accertarsi passino i controlli di qualità. Questi vengono eseguiti automaticamente grazie all'utilizzo delle *GitHub Actions*, una piattaforma di *Continuous Integration and Delivery* (*CI/CD*) che permette di automatizzare il processo di *testing*, *building* e *deployment*. Tramite un file di configurazione *Yaml* è possibile definire uno o più *workflow* che effettuino i controlli desiderati al verificarsi di alcuni eventi come, ad esempio, una *pull request* o il *push* di un nuovo *commit*. In particolare, il progetto è configurato per eseguire i *test* ad ogni *commit* che viene *pushato* su

[5] `https://github.com/DanySK/code-plagiarism-detector`

tutti e tre i sistemi operativi più diffusi, *Linux*, *MacOS*, *Windows*, e per effettuare, ad ogni *pull request*, ulteriori controlli come la copertura del codice in termini di test[6] e la qualità del codice[7] al fine d'identificare problemi di sicurezza, prestazioni, affidabilità e stile. Questo metodo di operare è alla base della filosofia *DevOps* e prevede di verificare, ad ogni cambiamento, che la *build* rimanga intatta, al fine d'individuare eventuali problemi appena questi si verificano e non dopo settimane, al termine dell'integrazione.

Note di sviluppo

- La parallelizzazione dell'analisi dei sorgenti è stata sviluppata per mezzo di *ParallelStream* il quale, a loro volta, fa uso di un *framework* di tipo *fork-join*, che ha l'onere di creare e dividere la computazione tra i vari *thread* e gestire le relative *join* in maniera trasparente allo sviluppatore. Per far ciò si è dovuto, tuttavia, prestare attenzione affinché gli algoritmi di *detection* fossero *thread-safe*.
- Per lo sviluppo complessivo del progetto sono state impiegate le seguenti librerie:
  - **JavaParser**[8] per effettuare l'analisi dei sorgenti sviluppati in linguaggio Java;
  - **Github API for Java**[9] per interagire con l'API di GitHub e recuperare le *repository*. Per recuperare i progetti caricati su *Bitbucket*, non essendoci una libreria pronta all'uso, si è utilizzato **Unirest** per effettuare le richieste HTTP;
  - **JGit**[10] per lo scaricamento (*clone*) dei progetti da *GitHub* e *Bitbucket*.
  - **Clikt**[11] per la scrittura dell'interfaccia da riga di comando;
  - **Kaml**[12] per la (de)serializzazione di file di configurazione *Yaml*.
  - **Kotest**[13] e **Mockk**[14] per effettuare e agevolare la scrittura di *test*.

[6] `https://about.codecov.io/`
[7] `Sonatype Lift`
[8] `https://javaparser.org`
[9] `https://github-api.kohsuke.org`
[10] `https://www.eclipse.org/jgit/`
[11] `https://ajalt.github.io/clikt/`
[12] `https://github.com/charleskorn/kaml`
[13] `https://kotest.io`
[14] `https://mockk.io`

Capitolo 4

Validazione, conclusioni e lavori futuri

In questo capitolo vengono riportati e analizzati i risultati ottenuti nell'ambito di applicazione dello strumento sviluppato. Per ultimo, si traggono le conclusioni del progetto, elencando brevemente i possibili sviluppi futuri.

4.1 Valutazione qualitativa

Le prestazioni del sistema sono misurate sulla base di due aspetti fondamentali e indipendenti tra loro: le *performance*, in termini di tempo d'esecuzione, ma soprattutto l'accuratezza. Infatti, lo scopo primario è rilevare con precisione casi probabili di plagio e presentarli all'utente il più in alto possibile nei *report* generati dallo strumento. È solo in seguito ad aver raggiunto un grado di precisione soddisfacente che possono essere adottate ottimizzazioni che permettano di ridurre il tempo di calcolo.

Dunque, per testarne l'accuratezza, sono stati considerati 130 diversi progetti universitari sviluppati nel triennio 2019-2021 e per ciascuno di questi si è effettuato il confronto con tutti i progetti a disposizione dal 2015 in poi che, complessivamente, ammontano a 354 progetti. Questi sono sviluppati in Java, generalmente in *team* di quattro componenti, e rappresentano artefatti di medio-alta complessità, costituiti in media da più di un centinaio di sorgenti ciascuno.

4.1.1 Analisi di sensitività

Dapprima si è testato il sistema eseguendo una scansione senza applicare alcun filtraggio intermedio delle rappresentazioni come descritto nella Sezione 3.2, scegliendo come parametri di riferimento i seguenti:

- s, minima lunghezza della sequenza di *token* rilevabile dall'algoritmo di *detection* (*RKR-GST*), pari a 15;
- *min_duplication*, valore di soglia percentuale sotto la quale la similarità di una coppia di sorgenti non viene segnalata, pari al 30%;
- metrica di similarià media sorgente-sorgente espressa dall'Equazione (3.3).

A seguito dell'analisi si è effettuata un'ispezione manuale dei sorgenti in modo tale da verificare che le stime effettuate dallo strumento non fossero fuorvianti o prive di fondamento.

I risultati complessivi sono presentati in Tabella 4.1.

Progetto originale	**Progetto copiato**	**Similarità**	**Ispezione manuale**
`9b7266`	`80fd2e`	100%	corrispondenza totale
`f0caf3`	`c1e451`	90%	corrispondenza totale
`ac8a48`	`7d79ff`	81%	corrispondenza elevata
`7bc0ee`	`2308d9`	70%	corrispondenza medio-alta
`a5c39e`	`2ed153`	62%	corrispondenza medio-alta
`005bc2`	`f67c20`	55%	corrispondenza parziale
`501b0f`	`c01302`	49%	corrispondenza parziale
`8f5d5a`	`afcd72`	48%	falso positivo

Tabella 4.1: Risultati ottenuti confrontando i progetti consegnati negli ultimi tre anni accademici. Sono riportati, per semplicità, solo le corrispondenze con indice di similarità superiore al 40%; i nomi dei progetti sono offuscati per rispetto della *privacy*. Nell'ultima colonna è riportato l'esito dell'ispezione manuale effettuata a seguito di quella automatica; le corrispondenze sono catalogate in: corrispondenza *totale*, *elevata*, *parziale* o *falso positivo*.

Come si può osservare, lo strumento è in grado di rilevare molto bene copiature evidenti, ovvero casi in cui sono state copiate intere classi di sorgenti: è il caso dei primi due risultati in cui i progetti sono pressoché identici e la percentuale di similarità stimata non è inferiore al 90%.

Come naturale che sia, invece, laddove la copiatura è effettuata in modo più scaltro, attingendo in modo parziale da altri progetti, la similarità cala, mantenendosi tuttavia al di sopra del 60%.

Sotto il 60% si è in presenza di codice piuttosto simile, ma in maniera molto limitata rispetto alla totalità dei sorgenti. È il caso, ad esempio, dei progetti `005bc2` e `f67c20` in cui è intuibile, guardando i risultati generati dal rilevatore, che gli sviluppatori del secondo abbiano preso spunto dal primo. Tuttavia non è ravvisabile una copiatura evidente come nei precedenti casi.

Un solo caso, tra tutti i progetti analizzati, riporta similarità maggiore del 45% senza essere realmente copiato. Ciononostante, la similarità riportata non è superiore al 50% e il fatto che i progetti non siano copiati né simili è facilmente verificabile in pochi minuti guardando il *report* stesso, che comprende solo costanti e *getter/setter*.

Successivamente, sono state rieffettuate le scansioni dei soli progetti somiglianti presentati in Tabella 4.1, variando in modo opportuno i parametri scelti al fine d'identificare la loro combinazione ottimale da utilizzare.

Come si può notare dalla Tabella 4.2, le misurazioni di similarità più accurate in relazione all'ispezione manuale riportata in Tabella 4.1 sono in corrispondenza di una lunghezza s variabile tra 15 e 20 e un valore di soglia di duplicazione minima compreso tra il 30 e il 40%. Infatti, scegliere una lunghezza s e un valore di duplicazione al di sopra di questi limiti, seppur mantenga inalterate, se non migliori, le prestazioni per i casi di copiature evidenti (è questo il caso della coppia di progetti `f0caf3` e `c1e451`), diminuisce l'accuratezza nei casi in cui la copiatura è meno evidente.

Inoltre, si noti che la misurazione della similarità mediante la normalizzazione massima, così come descritta nell'Equazione (3.4), in concomitanza con quella dei progetti (Equazione (3.7)), genera risultati fuorvianti in corrispondenza di sorgenti e/o progetti di piccola dimensione. Questo fenomeno è facilmente osservabile dalle stime di similarità tra i progetti `005bc2` e `f67c20` e ci spinge a preferire l'utilizzo della metrica di normalizzazione media descritta nell'Equazione (3.4) che fornisce risultati più stabili, derivanti dal fatto che considera la media della lunghezza dei sorgenti e non il sorgente più corto per il calcolo della similarità.

Dunque, la combinazioni di parametri che forniscono risultati migliori secondo i *test* effettuati sono le seguenti:

- $s = 15$;
- $min\_duplication = 0.3$
- metrica descritta nell'Equazione (3.3)

- $s = 15$;
- $min\_duplication = 0.4$
- metrica descritta nell'Equazione (3.3)

Entrambi sono due scelte ragionevoli: la seconda offre il vantaggio di annullare il falso positivo, portando la sua stima di similarità ad essere inferore del 25%, ma, d'altro canto, mette in maggior risalto (in maniera ingiustificata) le corrispondenze tra la coppia `f67c20` − `005bc2` e `501b0f` − `c01302` che, presumibilmente, ci si aspetterebbe sotto il 60%, mentre la prima fornisce una stima più accurata della similarità di questi ultimi casi, ma presenta lo svantaggio di riportare un falso positivo all'attenzione dell'utente.

				Coppie di progetti simili							
				9b7266 – 80fd2e	f0caf3 – c1e451	ac8a48 – 7d79ff	7bc0ee – 2308d9	a5c39e – 2ed153	005bc2 – f67c20	501b0f – c01302	8f5d5a – afcd72
parametri	$min\_dup = 0.3$	$s = 15$	Max	100%	95%	72%	72%	75%	80%	56%	74%
			Avg	100%	90%	81%	70%	62%	55%	49%	48%
		$s = 20$	Max	100%	99%	72%	75%	63%	83%	50%	57%
			Avg	100%	91%	58%	71%	48%	61%	51%	31%
		$s = 25$	Max	100%	98%	54%	66%	49%	79%	46%	26%
			Avg	100%	92%	42%	58%	41%	51%	40%	12%
	$min\_dup = 0.4$	$s = 15$	Max	100%	96%	81%	76%	70%	79%	61%	75%
			Avg	100%	92%	73%	79%	57%	62%	55%	22%
		$s = 20$	Max	100%	99%	65%	85%	63%	80%	57%	47%
			Avg	100%	97%	49%	69%	44%	54%	39%	10%
		$s = 25$	Max	100%	99%	49%	64%	42%	80%	46%	64%
			Avg	100%	99%	36%	62%	34%	52%	20%	6%
	$min\_dup = 0.5$	$s = 15$	Max	100%	99%	80%	84%	64%	76%	67%	27%
			Avg	100%	99%	51%	73%	33%	52%	33%	14%
		$s = 20$	Max	100%	99%	56%	82%	59%	56%	45%	35%
			Avg	100%	99%	30%	67%	30%	47%	23%	4%
		$s = 25$	Max	100%	99%	44%	63%	38%	56%	27%	15%
			Avg	100%	98%	21%	55%	27%	47%	11%	3%

Tabella 4.2: Analisi della similarità dei progetti rilevati come "sospetti" al variare dei tre parametri principali: s, minima lunghezza della sequenza di *token* rilevabile dagli algoritmi; min_dup, valore di soglia percentuale al di sotto della quale la similarità di una coppia di sorgenti non viene segnalata; metrica di normalizzazione media (Avg) e massima (Max) per la stima della similarità.

Si è infine testato il risultato dell'analisi eseguendo il filtraggio delle rappresentazioni come descritto nella Sezione 3.2.

Soglia filtro / Progetti	assente	0,3	0,4	0,5
`9b7266 – 80fd2e`	100%	100%	100%	100%
`f0caf3 – c1e451`	90%	90%	90%	90%
`ac8a48 – 7d79ff`	81%	81%	81%	80%
`7bc0ee – 2308d9`	70%	70%	70%	78%
`a5c39e – 2ed153`	62%	62%	62%	59%
`005bc2 – f67c20`	55%	55%	55%	55%
`501b0f – c01302`	49%	49%	49%	49%
`8f5d5a – afcd72`	48%	48%	48%	46%

Tabella 4.3: Analisi della similarità tra progetti al variare del valore di taglio con cui le rappresentazioni sono filtrate.

Osservando i risultati, si conviene che l'utilizzo di una fase intermedia di filtraggio di rappresentazioni non inficia le stime similarità dei casi presentati se il valore di taglio è mantenuto al di sotto del 50%.

In definitiva, dai risultati sopra riportati, una **buona soglia di "sospetto"** sopra la quale l'utente dovrebbe concentrare la sua attenzione è stimabile sul **45%**, sapendo che valori più alti di similarità implicano maggiore evidenza di copiatura.

4.1.2 Analisi di sensibilità

Dedichiamo ora qualche osservazione sulla sensibilità dello strumento.

Come sappiamo, il processo di *tokenizzazione* permette di convertire il codice sorgente in una sequenza di *token* che, da un lato, astragga il più possibile dalla sintassi del linguaggio, e dall'altra lo descriva a livello semantico, cosicché possano essere riconosciute porzioni semanticamente simili di codice.

Quando, tuttavia, si opera un'azione di astrazione di questo tipo si va incontro al rischio che strutture simili di sorgenti, e tuttavia diverse semanticamente, vengano convertite in sequenze di *token* che generano stime di similarità fuorvianti. È questo il caso di interfacce o classi molto semplici dove sono presenti solo *getter*/*setter* e/o una sequenza di costanti. A riprova di questo, vengono riportati, nel Listato 4.1 e Listato 4.2, alcune sezioni che il rilevatore ha determinato essere simili nella coppia di progetti riconosciuta come un caso di falso positivo del paragrafo precedente.

Listato 4.1: Esempio di falso positivo dovuto alla presenza di molteplici costanti.

```
>> With similarity: 0.676923076923077
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
[afcd72/src/main/java/*.java]
    public static final String IMAGE_PATH = "/images";

    /**
     * The title of the main stage that will be shown.
     */
    public static final String TITLE = "TrafficSimulator";

    /**
     *
     */
    public static final String MAIN_SCENE_PATH = "layouts/base.
       fxml";

    /**
     *
     */
    public static final long MS_BETWEEN_FRAMES = 200;

    /**
     *
     */
    public static final double MAX_UPDATE_BOUND = 0.06;

    /**
     * Used to convert from milliseconds to seconds.
     */
    public static final double MSEC_TO_SEC = 0.001;
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
[8f5d5a/src/main/java/*.java]
    private static final double HEALTH = 20;
    private static final double DAMAGE_ON_CONTACT = 2;
    private static final double R = 0;
    private static final double DX = 2;
    private static final double DY = 0;
    private static final double DR = 0;
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Listato 4.2: Esempio di falso positivo dovuto alla presenza di una sequenza molto simile di *getter/setter*.

```
>> With similarity: 0.6119402985074627
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
[8f5d5a/src/main/java/*.java]
        this.deathAnimationCounter =
            STANDARD_DEATH_ANIMATION_COUNTER_VALUE;
        this.animationTimerStopped =
            STANDARD_ANIMATION_TIMER_STOPPED_VALUE;
    }

    //Getters&Setterss:
    public final DebugInformations getDebugInformations() {
        return debugInformations;
    }

    public final void setDebugInformations(final DebugInformations
        debugInformations) {
        this.debugInformations = debugInformations;
    }

    public final int getDeathAnimationCounter() {
        return deathAnimationCounter;
    }

    public final void setDeathAnimationCounter(final int
       deathAnimationCounter) {
        this.deathAnimationCounter = deathAnimationCounter;
    }

    public final boolean isAnimationTimerStopped() {
        return animationTimerStopped;
    }

    public final void setAnimationTimerStopped(final boolean
       animationTimerStopped) {
        this.animationTimerStopped = animationTimerStopped;
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
[afcd72/src/main/java/*.java]
        this.number = number;
        this.maxVel = maxVel;
    }

    public DirOfMovement getSense() {
        return sense;
    }

    public void setSense(final DirOfMovement sense) {
```

```
42          this.sense = sense;
43      }
44
45      public int getNumber() {
46          return number;
47      }
48
49      public void setNumber(final int number) {
50          this.number = number;
51      }
52
53      /**
54       * @return the maxVel
55       */
56      public int getMaxVel() {
57          return maxVel;
58      }
59
60      /**
61       * @param maxVel the maxVel to set
62       */
63      public void setMaxVel(final int maxVel) {
64          this.maxVel = maxVel;
65  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Come si può osservare entrambi i sorgenti sono erroneamente classificati come somiglianti con un'elevata percentuale, superiore al 60%. Quando questo fenomeno è limitato a qualche istanza, ciò non comporta un particolare problema in quanto viene assorbito e bilanciato dalle diverse metriche che computano una stima media delle similarità. Laddove, invece, il numero di queste errate classificazioni cresce in modo corposo, si possono verificare casi di falsi negativi come quello presentato nel paragrafo precedente.

Si evidenzia, tuttavia, che questi casi di errata classificazione *non* si verificano in presenza di sorgenti con un grado di logica rilevante, dove la sequenza di *token* corrispondono se effettivamente la logica è molto simile. Perdipiù, questi casi sono facilmente individuabili mediante un'ispezione manuale e non rappresentano un elevato rischio in termini di riduzione di accuratezza.

4.2 Tempi d'esecuzione

Per quanto concerne i tempi d'esecuzione, attualmente lo strumento impiega, per fare il confronto di un progetto con i 354 progetti disponibili, grazie alla parallelizzazione e al salvataggio in locale dei sorgenti da analizzare, mediamente un terzo del tempo che avrebbe impiegato senza effettuare il *caching*, stimabile all'incirca in

al più 20 minuti. Questo dipende anche, in maniera incontrollabile, dal calcolatore su cui viene eseguito.

Sebbene possa sembrare un tempo elevato, nell'ambito di applicazione dello strumento questo è considerato un tempo ragionevole.

Infine, un'ultima considerazione sulla fase di filtraggio: nonostante il suo obiettivo fosse quello di diminuire il numero di comparazioni e, quindi, il tempo d'esecuzione dell'algoritmo, non si riescono ad osservare miglioramenti tangibili sotto questo aspetto. Questo induce a non consigliare l'utilizzo di questa fase intermedia che, nel peggiore dei casi, influenza la stima di similarità senza apportare alcun vantaggio sotto il profilo computazionale.

4.3 Conclusioni

Lo scopo della tesi, realizzare un sistema automatico per la scansione di progetti *software* alla ricerca di possibili segni di plagio, è stato raggiunto.

Dopo aver passato in rassegna le tecniche più usate e documentate in letteratura, nonché i relativi algoritmi da utilizzare per il confronto delle rappresentazioni, e aver soppesato *pro* e *contro* di ciascuna, si è deciso d'implementare una tecnica *structure-based* basata sulla visita dell'albero sintattico dei sorgenti e la *tokenizzazione* degli stessi.

Si è proceduto, quindi, con la progettazione architetturale e di dettaglio dello strumento, cercando sempre di prediligere e favorire la configurabilità e la possibilità di estensione dello stesso, in pieno accordo con uno dei principi cardine dell'ingegneria del *software*, l'anticipazione dei cambiamenti.

Ampio spazio è stato dedicato alla scelta di metriche congrue per la stima della similarità tra sorgenti ed è stato dato il proprio contributo nel definire una stima aggregata tra progetti.

Al termine dell'implementazione di una prima versione dello strumento, consci dell'elevato tempo d'esecuzione dello stesso, è stata introdotta la parallelizzazione e il salvataggio locale dei sorgenti.

Infine, si è validato lo strumento confrontando 130 progetti universitari consegnati negli ultimi tre anni, analizzando gli aspetti positivi e le criticità. Dall'analisi è emerso che lo strumento realizzato è in grado di rilevare, con un alto livello di accuratezza e in tempi accettabili, evidenti copiature e parti molti simili tra loro. Riesce altresì a fornire una stima di similarità che degrada mano a mano le copiature sono effettuate in modo parziale e/o con oculate tecniche di rifattorizzazione. Inoltre, l'elevato grado di configurabilità con cui è stato concepito permette di tararlo opportunamente in base alle proprie esigenze.

Sebbene, quindi, il sistema sia migliorabile, soprattutto sotto il punto di vista del tempo di calcolo, ora si ha uno strumento con un'architettura flessibile ed

estendibile che permette di poter implementare nuove tecniche e/o metriche più efficienti e performanti partendo da una solida base.

4.4 Sviluppi futuri

Per concludere, vengono di seguito elencati, per punti, alcune estensioni che si potrebbero implementare per rendere lo strumento più completo:

- Sviluppare tecniche di filtraggio delle rappresentazioni più evolute della semplice similarità coseno, qui implementata, al fine di determinare un incremento di prestazioni senza penalizzare l'accuratezza;
- Aggiungere tecniche di *clustering* a seguito della comparazione dei sorgenti, come descritto in [MV05], al fine d'individuare con maggior precisione addensamenti di sorgenti duplicati e, quindi, incrementare l'accuratezza del sistema;
- Aggiungere un'interfaccia grafica per rendere lo strumento più *user-friendly*;
- Aggiungere metodi di esportazione dei risultati come, ad esempio, pagine HTML in cui visualizzare comparativamente le sezioni di codice simili con una formattazione migliorata tipica di un ambiente di sviluppo.

Bibliografia

[Blo18] Joshua Bloch. *Effective Java*. USA: Pearson Education, 2018. ISBN: 978-0134685991.

[Bri22] Encyclopedia Britannica. *plagiarism*. September 23, 2022. URL: https://www.britannica.com/topic/plagiarism..

[ĎKH17] Michal Ďuračík, Emil Kršák e Patrik Hrkút. «Current Trends in Source Code Analysis, Plagiarism Detection and Issues of Analysis Big Datasets». In: *Procedia Engineering* 192 (2017). 12th international scientific conference of young scientists on sustainable, modern and safe transport, pp. 136–141. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2017.06.024. URL: https://www.sciencedirect.com/science/article/pii/S1877705817325705.

[EV44] R. Johnson E. Gamma R. Helm e J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1944. ISBN: 0201633612.

[FR87] J.A.W. Faidhi e S.K. Robinson. «An empirical approach for detecting program similarity and plagiarism within a university programming environment». In: *Computers & Education* 11.1 (1987), pp. 11–19. ISSN: 0360-1315. DOI: https://doi.org/10.1016/0360-1315(87)90042-X. URL: https://www.sciencedirect.com/science/article/pii/036013158790042X.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[JL99] M. Joy e M. Luck. «Plagiarism in programming assignments». In: *IEEE Transactions on Education* 42.2 (1999), pp. 129–133. DOI: 10.1109/13.762946.

[KAR+19] Oscar KARNALIM et al. «Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation». In: *Informatics in Education* 18.2 (2019), pp. 321–344. ISSN: 1648-5831. DOI: 10.15388/infedu.2019.15.

[Liu+06] Chao Liu et al. «GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis». In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 872–881. ISBN: 1595933395. DOI: 10.1145/1150402.1150522. URL: https://doi.org/10.1145/1150402.1150522.

[MAB13] Basavaraju Muddu, Allahbaksh Asadullah e Vasudev Bhat. «CPDP: A robust technique for plagiarism detection in source code». In: *2013 7th International Workshop on Software Clones (IWSC)*. 2013, pp. 39–45. DOI: 10.1109/IWSC.2013.6613041.

[MV05] Lefteris Moussiades e Athena Vakali. «PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets». In: *The Computer Journal* 48.6 (giu. 2005), pp. 651–661. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxh119. eprint: https://academic.oup.com/comjnl/article-pdf/48/6/651/1025682/bxh119.pdf. URL: https://doi.org/10.1093/comjnl/bxh119.

[Ott76] Karl J Ottenstein. «An algorithmic approach to the detection and prevention of plagiarism». In: *ACM Sigcse Bulletin* 8.4 (1976), pp. 30–41.

[Pat] Java Design Patterns. *Pipeline pattern*. URL: https://java-design-patterns.com/patterns/pipeline/.

[SK19] Lisan Sulistiani e Oscar Karnalim. «ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment». In: *Computer Applications in Engineering Education* 27.1 (2019), pp. 166–182. DOI: https://doi.org/10.1002/cae.22066. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22066. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.22066.

[SWA03] Saul Schleimer, Daniel S. Wilkerson e Alex Aiken. «Winnowing: Local Algorithms for Document Fingerprinting». In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: Association for Computing Machinery, 2003, pp. 76–85. ISBN: 158113634X. DOI: 10.1145/872757.872770. URL: https://doi.org/10.1145/872757.872770.

[Uff] Gazzetta Ufficiale. *Protezione del diritto d'autore e di altri diritti connessi al suo esercizio. (041U0633)*. URL: http://www.normattiva.it/uri-res/N2Ls?urn:nir:stato:legge:1941-04-22;633.

[UJM21] Farhan Ullah, Sohail Jabbar e Leonardo Mostarda. «An intelligent decision support system for software plagiarism detection in academia». In: *International Journal of Intelligent Systems* 36.6 (2021), pp. 2730–2752. DOI: https://doi.org/10.1002/int.22399. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/int.22399. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/int.22399.

[Wis93] Michael J. Wise. *Running Karp-Rabin matching and greedy string tiling*. eng. OCLC: 38352625. Sydney: Basser Dept. of Computer Science, University of Sydney, 1993. ISBN: 9780867586695.

[YCE22] Kadir Yalcin, Ilyas Cicekli e Gonenc Ercan. «An external plagiarism detection system based on part-of-speech (POS) tag n-grams and word embedding». In: *Expert Systems with Applications* 197 (2022), p. 116677. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2022.116677. URL: https://www.sciencedirect.com/science/article/pii/S0957417422001610.

Ringraziamenti

Dopo tre anni eccoci arrivati al tanto atteso finale di stagione. Sono stati anni impegnativi, ricchi di emozioni, soddisfazioni e momenti difficili. Fortunatamente non ero solo, ma avevo affianco persone speciali, a cui non potrei non rivolgere un caro ringraziamento.

In primis, desidero ringraziare il professor Pianini, il relatore di questa tesi, per la sua disponibilità, cordialità e i suoi preziosi stimoli e consigli.

Un Grazie alla mia famiglia che mi ha sostenuto e permesso di trovarmi dove sono ora.

Grazie a tutti gli amici con cui ho avuto il piacere di condividere questo percorso e, in particolare, a Luana, compagna di progetti e di studio, preziosa consigliera, nonché sopraffina intenditrice di *palette* colori e gradienti.

Per ultimo il ringraziamento più sentito a una persona speciale, Matteo, e alla sua famiglia, che, soprattutto in quest'ultimo periodo, mi ha saputo consigliare, supportare e sopportare e che, sin da quando ci siamo conosciuti, mi ha reso una persona migliore.

Appendice A

Esempio di *report* generato dallo strumento

Di seguito viene mostrato una porzione di un *report* generato dallo strumento sviluppato.

Listato A.1: Esempio di *report* generato dal *tool*.

```
**********************************************************************
Report from plagiarism detector tool generated at 2022-11-13T17...
**********************************************************************

Submitted Project: 7d79ff

++++++++++++++++++++++++++++ SUMMARY ++++++++++++++++++++++++++++
Compared with 354 repositories.
----------------------------------------------------------------
|compared with                                          |similarity|
----------------------------------------------------------------
|ac8a48                                                 |      0.81|
|929c18                                                 |      0.10|
|2b58aa                                                 |      0.08|
|                            ...                                 |
----------------------------------------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

> Matches found with ac8a48
                            ...
>> With similarity: 0.8655737704918033
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
[7d79ff/src/main/java/*/ShotManagerImpl.java]
        this.zombieModel.getZombies().stream()
            .filter(z -> Collision.isColliding(s.getBoundingBox(),
                z.getBoundingBox()))
            .forEach(z -> {
```

```
27                  this.zombieModel.hitZombie(z, s.getDamage());
28                  delete.add(s);
29              });
30          if(!delete.contains(s) && this.walls.stream().anyMatch(w
                -> Collision.isColliding(w, s.getBoundingBox()))) {
31              delete.add(s);
32          }
33      });
34      this.ended.addAll(delete);
35      delete.forEach(s -> this.shotsActive.remove(s));
36  }
37
38  /**
39      * {@inheritDoc}
40      */
41  @Override
42  public final Set<Shot> getShotsEnded() {
43      final Set<Shot> tmp = new HashSet<>(this.ended);
44  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
45  [ac8a48/src/main/java/*/AttackManagerImpl.java]
46              this.zombieModel.getZombies().stream()
47                  .filter(z -> CollisionsUtils.isColliding(z.getBBox
                        (), a.getBBox()))
48                  .forEach(z -> {
49                      this.zombieModel.hitZombie(z, a.getDamage());
50                      toDelete.add(a);
51                  });
52              if (!toDelete.contains(a) && this.obstacles.stream().
                    anyMatch(e -> CollisionsUtils.isColliding(e, a.
                    getBBox()))) {
53                  toDelete.add(a);
54              }
55          }
56      });
57
58      this.deleted.addAll(toDelete);
59      toDelete.forEach(a -> this.attacksActive.remove(a));
60  }
61
62  /**
63   * {@inheritDoc}
64   */
65  @Override
66  public final Set<Attack> getAttacksEnded() {
67      final Set<Attack> tmp = new HashSet<>(deleted);
68  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```