AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA *Informatyki Stosowanej i Modelowania*

Projekt dyplomowy

Metryki oprogramowania w zastosowaniu do określenia samodzielności pracy programistycznej

Software metrics applied to plagiarism detection of program code

Autor: *Karol Matoga*
Kierunek studiów: *Informatyka stosowana*
Opiekun pracy: *dr inż. Gabriel Rojek*

Kraków, 2022

Serdecznie dziękuję dr inż. Gabrielowi Rojkowi
za poświęcony czas i pomoc w
realizacji niniejszej pracy.

Spis treści

1. Wstęp

Od zarania dziejów, w cyklu nauczania danego człowieka, na każdym przedmiocie (ale i nie tylko), w większym bądź mniejszym stopniu kładziono nacisk na samodzielność wykonywanej, szeroko pojętej pracy. Ma to, jak najbardziej swoje uzasadnienie – praca wykonana samodzielnie uczy, jak najwięcej abstrahując od sytuacji gdzie plagiatując czyjąś twórczość, autor oryginału może nie dostać za nią wynagrodzenia w postaci niematerialnej czy też pieniężnej, ponieważ złodziej by go uprzedził – jest to okoliczność niedopuszczalna. Skupiając się na etapie edukacji, oczywiście można podczas tego całego procesu popełnić nieskończoną ilość błędów, ale w myśl starej maksymy: „*Człowiek uczy się na własnych błędach*" – jest to rzecz, nie na pierwszy rzut oka, bez wątpienia pożądana. Istotnie nie zawsze jest tak, że dana osoba musi pracować samodzielnie w celu kompletnego zrozumienia danego zagadnienia – prace grupowe, wzajemna pomoc też są kluczowe. Nie należy natomiast zatracić granicy między pracą grupową, a przepisywaniem rozwiązania 1:1.

W celu walki z plagiatem radzono sobie w różnoraki sposób. Miało na to wpływ bardzo dużo czynników np.: na jakim przedmiocie faktycznie z nim będziemy mieć styczność tudzież miejsce na linii czasu, z którym w danym momencie mamy do czynienia. Nie jest sekretem to, że w miarę postępu technologicznego co raz to trudniej było uprawiać plagiat przez wszelakie udogodnienia systemowe. Można tutaj dokonać ogólnego podziału wcześniej wspomnianych „udogodnień systemowych" na sprzętowe, systemowe oraz mieszane. Do tych sprzętowych można zaliczyć np.: kamery, gdzie prowadzący przy ich pomocy mógł sprawdzić czy ktoś nie łamie przepisów, przepisując prace od kolegi. Jeżeli chodzi o rozwiązanie systemowe, można do ich grona przyporządkować rozmaite programy komputerowe, które analizowałby prace uczących się. Niektóre z rozwiązań pasowałby bardziej w danym kontekście antyplagiatowym, z drugiej strony były metody, które w ogóle by się w konkretnej sytuacji nie sprawdziły. Warto mieć na uwadze to, że postęp technologiczny jest bronią obosieczną – metody plagiatowania również przeszły na inny, bardziej wyrafinowany poziom.

Zagadnienie minimalizacji plagiatu albo jego całkowitego wyeliminowania nie ominęło również relatywnie młodej dziedziny jaką jest informatyka, a dokładniej – programowanie. Jako, że jest ono stricte skorelowane z pisaniem, najbardziej adekwatną metodą do sprawdzenia czy doszło do plagiatu, będzie odpowiednio stworzony i skonfigurowany system. Niewątpliwie, można dokonywać sprawdzenia przy pomocy rozwiązań sprzętowych (np.: wcześniej wymienione kamery), aczkolwiek będzie to mniej efektywne niż program komputerowy, który analizowałby kod. Wraz z powstaniem pierwszego języka programowania, a co za tym idzie, programów w nim napisanych powstawały

strategie antyplagiatowe w postaci algorytmów. Bynajmniej stworzenie efektywnego algorytmu nie jest zadaniem trywialnym. My jako ludzie patrząc na kod, czytając go i porównując z innym niejako podświadomie jesteśmy w stanie powiedzieć czy padł on w większym bądź mniejszym stopniu ofiarą plagiatu, chociaż też nie zawsze – może być on bardzo długi i mieć wiele referencji, co skutecznie utrudni proces weryfikacji. Warto mieć na uwadze, że efektywny algorytm do wykrywania plagiatu powinien nie tylko zero-jedynkowo sprawdzać zbieżność konkretnych słów, ale też niejako pod kątem stylu, kontekstu (jest to tak zwany plagiat ukryty). Algorytmy z czasem ewoluowały, poczynając od porównywania metryk Maurice Howarda Halstead'a [1], kreując zaawansowane rozwiązania takie, jak JPlag [2] i są dalej rozwijane tworząc nowe przyrządy do walki z plagiatem, które to dają co raz to lepsze rezultaty. Mogą być one wykorzystywane w różnych miejscach: w szkołach, uczelniach wyższych, prywatnie lub komercyjne w firmach, skutecznie wykrywając próbę oszustwa, dodatkowo poprawiając umiejętności programowania danego programisty, jeżeli wykorzystuje on algorytm do własnego, pozytywnego celu.

1.1 Cel pracy

Celem pracy jest wykrywanie plagiatu między kodem źródłowym (w przypadku tej pracy – Java) w kontekście metryk oprogramowania, zaimplementowanie rozwiązania oraz jego przetestowanie. Zadanie to wymagało głębszego rozeznania w zakresie algorytmów antyplagiatowych oraz w jaki sposób one działają. Istnieje, bowiem wiele sposobów wykrywania kradzieży kodu, np.: algorytmy bazujące na tzw. metodzie „odcisku palca" [3] czy też metodzie grafów zależności [4]. Samo zagadnienie metryk oprogramowania i użycie ich do pomocy wykrywania splagiatowanych prac również daje nam szerokie pole do popisu. Możemy np.: porównać wyliczone metryki dla danych kodów źródłowych przy pomocy wzorów statystycznych, takich, jak współczynnik równoważności. Oprócz kwestii samego porównywania metryk możemy operować właśnie na ich wariacji – nie ograniczają nas tylko wyżej wspomniane metryki Maurice Halstead'a, możemy też użyć np.: szerszej gamy wyznaczników liczbowych kodu [6]. Konsolidując, cel pracy można wyszczególnić w następujących punktach:

- objaśnienie zagadnienia znajdywania podobieństw w programach
- przedstawienie oraz opisanie metod, które pomagają w znajdywaniu splagiatowanych prac
- opisanie szeroko pojętych metryk oprogramowania: co to jest, skąd się wzięły, do czego można ich użyć, jak są liczone i jak można je wykorzystać do walki z plagiatem

- implementacja programu wykrywającego plagiat bazującego na metrykach oprogramowania
- przeprowadzenie testów na zaimplementowanym programie, porównanie metod, które używają metryk do analizy kodu
- wyciągnięcie wniosków z pracy, napisanie ewentualnych sugestii odnośnie poprawy oprogramowania

2. Metryki kodu

Nie od dzisiaj wiadomo, że ilość w każdej dziedzinie życia nie gra głównej roli, ważna też jest jakość. Pisząc kod źródłowy, możemy mierzyć jakość oprogramowania przy użyciu nie tylko zadowolenia klienta z danego programu (chociaż to nie jest wymierny wskaźnik), lecz też przy użyciu wszelakiej maści metryk oprogramowania. Oczywiście jakość oprogramowania jest niejednoznacznym terminem, jeżeli chodzi o metryki – możemy policzyć metryki w celu użycia ich potem do analizy czy doszło do plagiatu (nie każde metryki będą się do tego celu nadawały, a też wpasowuje się to w ramy jakości oprogramowania). Z drugiej strony, wybieramy inne metryki do oceny czy oprogramowanie jest dobrej, czy też złej jakości, znowu – jest to, jak najbardziej zagadnienie z dziedziny jakości oprogramowania. Systemy pisane przez programistów mają naturalną skłonność do bycia co raz to bardziej zawiłymi ze względu na goniący postęp technologiczny. Pielęgnowalność oraz utrzymanie ładu i składu jest z perspektywy czasu zadaniem kluczowym. I w tym wypadku atrybutami, które mogą nam pomóc wykonać to zadanie będą metryki oprogramowania.

Metryki oprogramowania to termin, który nie ma konkretnie sprecyzowanej definicji, może de facto oznaczać losową wartość liczbową znajdującą odzwierciedlenie w kodzie. Doszukując się bardziej usystematyzowanej definicji, można trafić na dokument IEEE 1061-1992, który opisuje metryki oprogramowania jako funkcje odwzorowujące jednostki oprogramowania na wartości liczbowe [7]. Inżynieria jakości oprogramowania dzieli metryki na trzy główne kategorie [25]:

- metryki jakości produktu
- metryki wewnątrzprocesowe
- metryki serwisowe oprogramowania

Warto mieć w pamięci to, że powyższe trzy kategorie metryk analizy dynamicznej programu – ISO/IEC 9126 kategoryzują je jako metryki zewnętrzne. W opozycji do nich mamy do czynienia z metrykami wewnętrznymi (ISO/IEC 9126) odpowiadającymi za analizę statyczną kodu. Optimum analizy jest osiągane wtedy, kiedy złączymy w analizie metryki zewnętrzne z wewnętrznymi.

2.1 Historia

Badania nad metrykami rozpoczęły się w latach 70 poprzedniego wieku [8], kiedy głównym paradygmatem programowania był paradygmat proceduralny. W 1974 roku, Wolverton pierwszy raz użył wartości, która mierzyła ilość linii kodu w programie [8]. Kolejnym atrybutem mierzącym jakość kodu był zaproponowany w 1976 roku przez Thomasa J. McCabe'a [8] termin „cykl" jako metryka, bazująca na grafie definicji [8] – było to nic innego, jak złożoność cyklomatyczna, służąca do pomiaru skomplikowania kodu, gdzie podstawą do jej wyliczenia była ilość rozgałęzień w schemacie blokowym programu [9]. Wcześniej wspomniany Maurice Halstead, zasugerował coś co nazwał „naukę o programowaniu" [8] – pod tym terminem ukrywało się rozwiązanie, które estymowało obciążenie pracy tudzież czas pracy programistów używając do tego operatorów, operandów i procesu wywołującego [10] dane oprogramowanie. W 1979 roku, Albrecht [8] zademonstrował metodę, która została nazwana „punkty funkcyjne" – znalazła zastosowanie w fazie analizy wymagań [8]. Oprócz wcześniej wymienionego paradygmatu proceduralnego mamy do czynienia również z innymi sposobami pisania kodu – jest nim np.: szeroko używany dzisiaj paradygmat zorientowany obiektowo, za którego ojcem uważa się Alana Kay'a [11]. Kod napisany w innym paradygmacie może wymagać wykorzystania odmiennych metryk oprogramowania, czasami specyficznych dokładnie dla tego konkretnego stylu. W 1993 roku naukowcy: J-Y Chen oraz J-F Liu uzewnętrznili metodę „Chen&Liu" [8], która używała różnorodne właściwości do obliczenia programu napisanego w paradygmacie obiektowo zorientowanym, takie, jak np.: złożoność, atrybuty klasy i możliwości ponownego wykorzystania (z ang. reusability). Kolejno w 1994, Shyama R. Chidamber i Chris F. Kemerer opublikowali zestaw metryk obiektowo zorientowanych bazując na drzewie dziedziczenia [8]. Idąc z duchem czasu, w 1995 roku Brito opowiadał się za zestawem metryk czerpiących cechy z paradygmatu obiektowo zorientowanego – nazwał ten zbiór „MOOD" [8]. Wynalezione metody (metryki) były pokłosiem intensywnego rozwoju metryk oprogramowania w latach 80 i 90. Wraz z nastaniem nowego wieku wymyślono taką metodę, jak „SPECTRE", wymyślona przez Victora i Daily'ego, która kalkulowała czas ewaluacji kodu i skalę modułów (z ang. scale of modules) [12], miara wielkości wektora (z ang. vector size measure (VSM)), opracowana przez Hastings'a i Sajeev'a, używana do oceny skali oprogramowania (z ang. software scale), a ponadto klasyfikacji oprogramowania (z ang. software classification) [8], czy też metoda prognozowania, stworzona przez Arlene F., szacująca obciążenie pracy nad kodem (z ang. work load) [8].

2.2 Taksonomia

Metryki związane z interpretacją programów, tak, jak wcześniej wspomniano, możemy podzielić na metryki statyczne (badamy zachowanie kodu nieuruchomionego) oraz dynamiczne (określane w oddzieleniu od kodu i analizują proces uruchomionego programu) [23]. Możemy też wyszczególnić metryki niepowiązane bezpośrednio z implementacją oprogramowania, lecz z [23]:

- specyfikacja
- wymogami klienta
- testami

Kolejna segregacja metryk wiąże się z ich podziałem na obiekty i atrybuty, które są przez nie liczone [24]:

- metryki produktów
- metryki rozmiaru
- metryki jakości
- metryki procesów
- metryki projektu

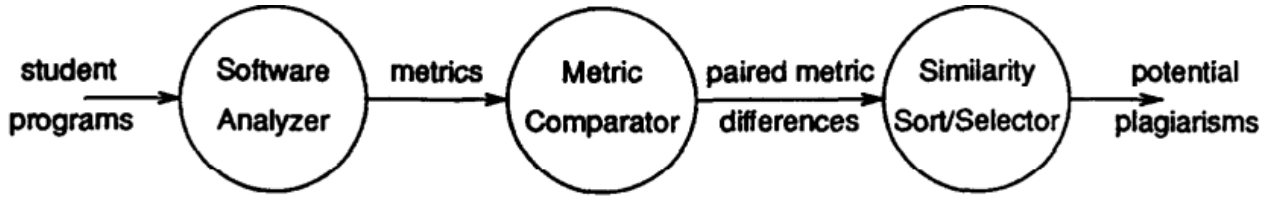
Warto podkreślić, że metryki mogą być obliczane bezpośrednio z kodu (np.: linie kodu), czy też pośrednio (np.: występowanie błędów).

2.3 Zastosowanie metryk

Metryki stosuje się w celu otrzymania liczbowych wartości, które odnoszą się do konkretnego oprogramowania. Dają nam one możliwość „surowej" oceny kodu na podstawie liczb – wtedy jest to czysto obiektywna ocena, a nie jak w przypadku innych wartości – subiektywna. Tom DeMarco uważał, że kontrolowanie tego, co jest nieuchwytne do zmierzenia jest niemożliwe [13].

Obliczone metryki można wykorzystać w różnych sytuacjach, np.: użyć ich w procesie wytwarzania oprogramowania, aby oszacować nakład pracy, nierozłączony, aby zrealizować pomysł (faza projektowania – najistotniejsza z punktu widzenia klientów oraz projektantów), pomagać w utrzymaniu jakości kodu źródłowego (faza produkcji – istotna z punktu widzenia programistów) pod kątem wyszukiwania miejsc w kodzie, które są podatne na błędy i w fazie testów badając przebieg wykonania programu, gromadząc dane dotyczące wydajności, jak również niezawodności aplikacji.

Kolejną obszerną dziedziną, w jakiej możemy wykorzystać metryki oprogramowania jest problematyka znajdywania plagiatu pomiędzy różnymi kodami źródłowymi. W telegraficznym skrócie, system wykrywający plagiat bazujący właśnie na metrykach oprogramowania jako metodzie porównawczej można zaprezentować w postaci poniższego rysunku:



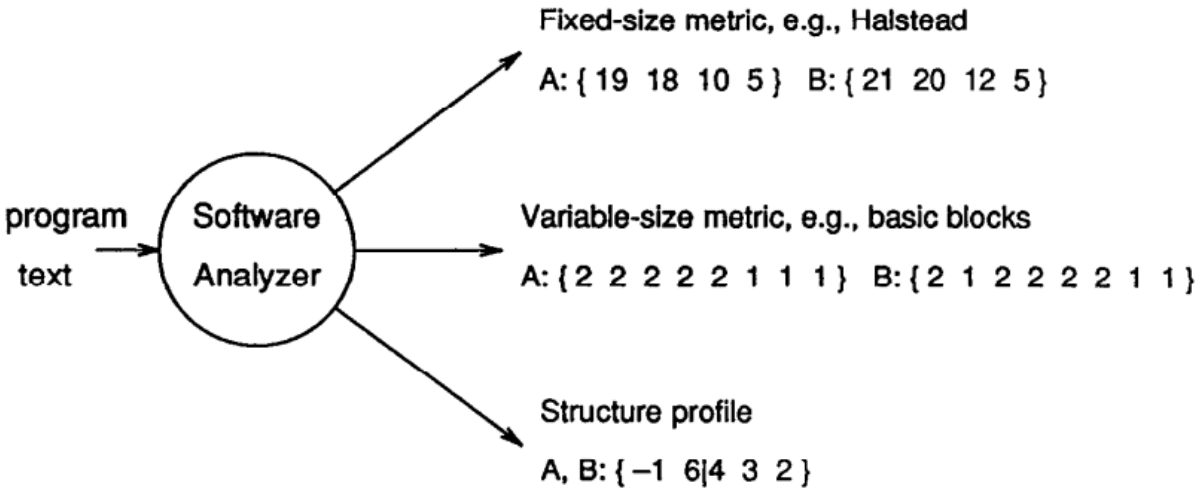
Rysunek 1 – system wykrywający plagiat
źródło [14]

Mechanizm można rozdzielić na trzy główne fazy:

- metryki generowane przez swojego rodzaju analizator oprogramowania
- algorytm porównujący metryki oprogramowania
- algorytm przetwarzający wyniki wcześniejszego porównania

Metryki generowane przez analizator mogą być różne, tak samo, jak użyte algorytmy.

2.4 Metryki w zastosowaniu do badań plagiatu



Rysunek 2 – metryki używane przy wykrywaniu podobieństw w kodzie
źródło [14]

Pierwszą próbą zautomatyzowanego wykrywania podobieństwa było rozwiązanie Ottenstein'a [1], które bazowało na metrykach Halstead'a [15] – całkowitej liczbie operatorów i operandów, jak i odrębnych operatorach i operandach. Całość formowała tzw. „profil Halstead'a" [14], gdzie podobne programy pod względem syntaktycznym miały mieć takowy zbliżony. To rozwiązanie było często używane jako metoda kontrolna dla innych systemów szukających podobieństw w kodach źródłowych.

Kolejno, starano się szukać „lepszych" metryk, ale podobnego typu co w profilu Halstead'a, aby otrzymać możliwie najlepszy rezultat. Grier zaimplementował te metryki do programów napisanych w języku Pascal, aczkolwiek poszerzył je o trzy własności:

- liczba linii kodu
- liczba zmiennych
- liczba zmiennych decyzyjnych

ulepszając przy tym algorytm, który potrafił znaleźć drobne detale pomiędzy profilami, a co z tego wynika, wyszukać potencjalny plagiat **[**6**]**. Donaldson używał w swoim narzędziu wykrywającym plagiat siedmioelementowy wektor, gdzie każdy z poszczególnych elementów charakteryzował jakąś wartość oprogramowania - była to m.in. liczba pętli, liczba podprogramów czy też ilość przypisań **[**26**]**. Oprócz tych rozwiązań Berghel i Sallach wprowadzili składający się z czterech zmiennych zestaw, w którego skład wchodziły takie wartości, jak:

- liczba przypisań
- liczba zmiennych
- liczba linii kodu
- liczba kluczowych słów

porównanie tych zestawień różnych programów dawało wymierniejsze efekty pod kątem znajdywania oszustw [17]. Wspólnym mianownikiem powyższych metod jest to, że może wystąpić taka okoliczność, iż będą słabo i niedokładnie oceniać stopień powiązania pomiędzy programami [14].

Metryki używane do rozpoznawania plagiatu nie są ograniczone do tzw. „stałych wektorów" [14] – czyli tak naprawdę liczbowych wartości określających konkretną rzecz w kodzie źródłowym, gdzie ich zmiany kolejności w wektorze nie wpływają na końcowy wynik oceny stopnia plagiatu. W

swojej pracy Donaldson, opisał w drugiej metodzie wykrywającej plagiat sposób reprezentacji programu, który polegał na jego przedstawieniu przy użyciu ciągu symboli (z ang. string) [26]. Każdy symbol oznaczał występowanie jakiegoś wyrażenia, a podobne programy powinny mieć ich sekwencje porównywalną. Z kolei Jankowitz opisał system, który wyszukiwał potencjalnie podobne kody źródłowe programów bazując na grafie wywołań proceduralnych [17]. Graf jest zapisany w postaci sekwencji binarnych symboli. Taki dobór metryk spowodował, że algorytm był zbyt bardzo precyzyjnie dostosowany do problemów o określonej wielkości, co zrodziło pewne niedociągnięcia w kontekście wyszukiwania plagiatu – małe programy wykazywały niewystarczającą różnorodność w strukturze wykonań procedur, gdzie z drugiej strony algorytm pracujący nad dużymi programami jest banalnie rozpraszany przez małe reorganizacje programu. Następne, nieco odmienne było rozwiązanie zaproponowane przez Robinsona oraz Soffę [18]. Program zawierał wektor liczb, w którego skład wchodziły wartości takie, jak ilość wyrażeń zawierająca się w tzw. „bloku podstawowym". Blok podstawowy jest konstruktem w programie, który zawiera jakąś część kodu wydzieloną ale stanowiącą całość. Sam w sobie jest metryką rozmiaru, ponieważ wraz z rozrostem programu zmienia się ich liczba. Ogół formuje profil gotowy do porównania. Wartości w blokach podstawowych są znaczącym ulepszeniem w stosunku do metryk o stałej długości, ponieważ są wrażliwe na sekwencje struktur sterujących, które tworzą cały program.

2.5 Metryki użyte w projekcie

W projekcie zdecydowano się użyć głównie trzech rodzajów metryk – Maurice Halstead'a oraz Chidamber & Kemerer (metryki CK) oraz metryk rozmiaru.

2.5.1 Metryki złożoności Halstead'a

Metryki Halstead'a zostały opracowane w 1997 roku przez Maurice Howarda Halstead'a w związku z jego pracą o ustanowieniu empirycznej nauki o rozwoju oprogramowania [15]. Celem przy ich opracowywaniu było wytworzenie mierzalnych wartości kodu, aby później móc je procesować w konkretnym kierunku np.: identyfikując plagiat. Mając takie narzędzie w postaci liczb opisujących oprogramowanie, czysto obiektywnie pochodzimy do jego oceny.

Dla zadanego problemu możemy wyszczególnić [10]:

- liczbę odrębnych operatorów – oznaczmy jako n1
- liczbę odrębnych operandów – oznaczmy jako n2

- całkowitą liczbę operatorów – oznaczmy jako N1
- całkowitą liczbę operandów – oznaczmy jako N2

Operator charakteryzuje znak, który definiuje konkretną czynność, która ma być wykonana. Możemy rozróżnić trzy rodzaje operatorów: operatory arytmetyczne, relacyjne oraz logiczne. Operand to ta część instrukcji komputerowej, która charakteryzuje dane będące częścią operacji wykonywanej przez komputer. Posiadając ten zestaw liczb, który już sam w sobie opisuje pewne charakterystyki kodu i na jego podstawie istnieje możliwość wyciągnięcia wniosków np.: dotyczących plagiatu, możemy policzyć następujące wartości [10]:

- słownictwo programu (z ang. program vocabulary) – n1 + n2 – definiuje liczbę unikatowych wystąpień operatorów i operandów
- długość programu (z ang. program length) – N1 + N2 - reprezentuje całkowitą liczbę wystąpień operatorów i całkowitą liczbę wystąpień operandów
- obliczona szacunkowa długość programu (z ang. calculated estimated program length) – n1*$\log_2 n1$+n2*$\log_2 n2$ – proporcjonalnie do wielkości kodu źródłowego, definiuje w bitach, przestrzeń potrzebną do przechowywania programu
- wolumen (z ang. volume) – (N1 + N2) × $\log_2(n1 + n2)$ – potencjalna, minimalna objętość wolumenu jest obliczana jako objętość najbardziej zwięzłego programu, w którym może być zakodowany
- trudność (z ang. difficulty) – n1/2 × N2/n2 – ten parametr ukazuje, jak trudny do obsługi jest program
- wysiłek (z ang. effort) – wolumen × trudność – mierzy wielkość aktywności umysłowej niezbędnej do przełożenia algorytmu na implementacje w konkretnym języku programowania
- czas potrzebny do zaprogramowania (z ang. time required to program) – wysiłek/18 sekundy – pokazuje czas w minutach (można dowolnie przełożyć na każdą inną jednostkę) potrzebny do przełożenia algorytmu na implementacje w określonym programie
- liczba dostarczonych błędów (z ang. number of delivered bugs) – $E^{\frac{2}{3}}$/3000 – wartość oblicza ilość szacunkowych błędów i jest zależna od użytego języka programowania

2.5.2 Metryki Chidamber & Kemerer (metryki CK)

Metryki Chidamber & Kemerer (metryki CK) zostały opracowane finalnie w 1994 roku przez Shyama R. Chidamber'a i Chrisa F. Kemerer'a szczególnie pod kątem programowania zorientowanego obiektowo – ukazują powiązania pomiędzy klasami oraz poziom ich złożoności. Bazują one na analizie drzewa dziedziczenia [19]. Metryki CK składają się z [20]:

- metody ważone na klasę (z ang. weighted methods per class) – reprezentuje sumę zagregowanych metod
- głębokość drzewa dziedziczenia (z ang. depth of inheritance tree) – reprezentuje stopień dziedziczenia obliczając maksymalną liczbę poziomów klas z wyższego stopnia dziedziczenia, po których badana klasa dziedziczy
- liczba dzieci (z ang. number of children) – reprezentuje bezpośrednio podklasy danej klasy
- sprzężenie między obiektami (z ang. coupling between objects) – reprezentuje zależność klasy z innymi klasami, które nie są jej przodkami ani potomkami, w relacjach odmiennych niż dziedziczenie
- odpowiedź dla klasy (z ang. response for a class) – reprezentuje możliwość komunikacji między klasą, a projektem – ilość metod, które mogą zostać wyegzekwowane, jak przyjmą informację przez obiekt badanej klasy
- brak spójności metod (z ang. lack of cohesion of methods) – reprezentuje brak spójności (klasa jest oddelegowana do egzekucji więcej niż jednej funkcji)
    a) LCOM1 (Chidamber & Kemerer)
    b) LCOM2 i LCOM3 (Henderson-Sellers, Constantine & Graham)
    c) LCOM4 (Hitz & Montazeri)

2.5.3 Metryki rozmiaru

Metryki rozmiaru pozwalają określić ilość atrybutów w kodzie oraz ich powiązań w różnych kontekstach. Są one niezbędne do określania rozmiaru i poziomu skomplikowania programu na danym poziomie. Znajdują one zastosowanie głównie w programowaniu proceduralnym.

- kompletna ilość metod (z ang. number of methods) – reprezentuje ogólną liczbę metod (metody statyczne, publiczne, abstrakcyjne, prywatne, chronione, domyślne i synchronizowane) i dzieli je ze względu na modyfikator dostępu
- liczba statycznych wywołań (z ang. number of static invocations (NOSI)) – reprezentuje liczbę wywołań metod statycznych
- liczba linii kodu (z ang. lines of code (LOC)) – reprezentuje liczbę linii kodu, ignoruje przy tym puste linie oraz komentarze
- liczba instrukcji powrotu (z ang. return quantity) – reprezentuje liczbę instrukcji powrotu
- liczba pętli (z ang. loop quantity) – reprezentuje liczbę pętli
- liczba konstrukcji *try/catch* (z ang. try/catch quantity) – reprezentuje liczbę konstrukcji *try/catch*
- liczba wyrażeń w nawiasach (z ang. parenthesized expressions qunatity) – reprezentuje liczbę wyrażeń w nawiasach
- liczba zmiennych typu *String* (z ang. string literals quantity) – reprezentuje liczbę zmiennych typu *String*, jeżeli jakaś zmienna z daną zawartością się powtórzy, również zwiększy licznik
- liczba zmiennych typu liczbowego (z ang. numbers quantity) – reprezentuje ilość zmiennych typu liczbowego (*int, float, double*)
- liczba przypisań (z ang. assignments quantity) – reprezentuje ilość przypisań do zmiennych
- liczba operacji matematycznych (z ang. math operations quantity) – reprezentuje liczbę operacji matematycznych (mnożenie, dzielenie, dodawanie, odejmowanie itd.)
- liczba zmiennych (z ang. variables qunatity) – reprezentuje liczbę zadeklarowanych zmiennych
- maksymalna liczba skonsolidowanych bloków kodu (z ang. max nested blocks quantity) - reprezentuje maksymalną liczbę bloków kodu, które są w sobie zagnieżdżone
- liczba unikalnych słów (z ang. unique words quantity) – reprezentuje liczbę unikalnych słów w kodzie źródłowym, po usunięciu słów charakterystycznych dla języka Java

3. Projekt

W rozdziale tym opisano detale projektu badającego zbieżność kodów źródłowych napisanych w języku programowania Java. Skupiono się na części dotyczącej projektowania, struktury, interakcji w systemie oraz doborze narzędzi, na których podstawie była budowana analiza, pomijając na tą chwilę część stricte implementacyjną.

3.1 Wymogi systemu

Projektując system wykrywający plagiat, starano się, żeby był on napisany „schludnie", biorąc pod uwagę przyszłe potencjalne zmiany, takie, jak np.:

- dodanie nowych języków programowania do analizy
- wprowadzenie nowych wzorów korelacji
- implementacja odmiennych algorytmów znajdujących plagiat
- konwersja rozwiązania na inny język programowania poprzez brak użytych zaawansowanych i specyficznych konstruktów języka Java (w tym języku programowania było pisane rozwiązanie)

Dodatkowo, zaprojektowana funkcjonalność powinna spełniać następujące kryteria:

- danie względnie wolnej ręki użytkownikowi, poprzez możność określenia miejsca gdzie znajdują się kody źródłowe do przetestowania
- możliwość interpretacji wyników w różnoraki sposób
    a) porównanie wzrokowe wyników (np.: metryki Halstead'a)
    b) procentowy wynik
- program powinien porównywać tylko dwa kody ze sobą (dla to możliwie najlepszy rezultat)
- nie ma ograniczenia, jeżeli chodzi o miejsce pochodzenia z komputera programów do porównania – programy mogą należeć do tego samego projektu, jak i znajdować się w dowolnym innym miejscu na komputerze
- program powinien analizować kody źródłowe niezależnie od stopnia ich złożoności zarówno rozmiarowej (ilość kodu), jak i syntaktycznej
- wzór korelacji użyty w projekcie powinien obliczać stopień podobieństwa niezależnie od ilości danych, pod warunkiem, że dane są poprawne

3.2 Wybór metryk oprogramowania do analizy

Motywem przewodnim przy wyborze wartości z punktu 2.5 do oceny oprogramowania i następnie dalszej obróbki pod kątem znajdywania plagiatu w kodzie były trzy rzeczy:

- kontekst historyczny
- paradygmat programowania
- różnorodność metryk w kontekście odporności na zaawansowany kod

Zapoznając się z literaturą odnoszącą się do metryk oprogramowania w kontekście znajdywania plagiatu między kodami źródłowymi, znaleziono pionierskie rozwiązanie Karl'a J. Ottenstein'a, które wykorzystywało właśnie metryki Halstead'a [1]. Rozwiązanie to porównywało wyliczone metryki dla danego kodu z innymi metrykami dla innego programu i na tej podstawie określało czy doszło do oszustwa. Badanie zostało wykonane przy użyciu języka programowania Fortran, czyli działano w tym przypadku, jeżeli chodzi o kontekst paradygmatu stricte w obszarze proceduralnym (chociaż język Fortran jest językiem wieloparadygmatowym [21]). Z uwagi na ten kontekst historyczny zdecydowano się na użycie metryk Halstead'a.

W pracy pracowano ściśle z językiem programowania Java – użyto go zarówno do napisania rozwiązania, jak i „testowych kodów" (kodów na których testowano rozwiązanie wykrywające plagiat). Język Java jest językiem wieloparadygmatowym [22], natomiast w pracy skupiono się na jego obiektowym charakterze. Metryki CK ściśle nawiązują do paradygmatu obiektowego [19].

Selekcja spośród obydwu zestawów metryk brała pod uwagę ograniczenia płynące z kalkulacji współczynników korelacji, które były użyte do obliczania powiązań pomiędzy badanymi kodami źródłowymi.

Biorąc pod uwagę powyższe ograniczenia, starano się spośród metryk zaproponowanych przez Halstead'a i Chidamber & Kemerer wybrać te, które spełniają wylistowane warunki.

Oprócz tych metryk użyto też np.:

- ilość procedur powrotu (z ang. quantity of returns)
- ilość pętli (z ang. quantity of loops)
- ilość operacji porównawczych (z ang. quantity of comparisons)
- ilość sformułowań try/catch (z ang. quantity of try/catches)

Dodatkową rzeczą na którą należało zwrócić uwagę był fakt, że nie w każdym kodzie występował jakiś konstrukt programistyczny (np.: *try/catch*), a brano pod uwagę również optymalizację kodu – wtedy starano się nie uwzględniać danej miary przy określaniu wyniku końcowego.

Chciano, aby praca zawierała duży zestaw metryk, aby kody, które są przez nią analizowane, nie kompromitowały jej pod kątem użycia np.: odrębnego paradygmatu programowania. Odrębny paradygmat programowania tudzież kod bardzo zaawansowany mógłby spowodować, że nie obliczylibyśmy adekwatnych metryk pod ten program i finalnie analiza plagiatu nie byłaby efektywna.

3.3 Wybór algorytmu porównującego obliczone metryki

W projekcie zdecydowano się użyć podejścia procesowania metryk oprogramowania dla kodów źródłowych przez współczynnik równoważności (z ang. equivalence ratio)

Współczynnik równoważności dla każdej pary analizowanych programów jest obliczany poprzez podział liczby ekwiwalentnych metryk policzonych dla każdego z tych programów przez średnią liczbę metryk, zakładając, że metryki mają tą samą wagę. Wynikiem jest współczynnik równoważności (podobieństwa) tudzież poziom plagiatu dla pary kodów źródłowych. Współczynnik ten skaluję się między <0, 1> gdzie 1 to najwyższy współczynnik podobieństwa, a 0 to najniższy współczynnik podobieństwa [5]. Wzór pozwalający obliczyć tą wartość prezentuje się następująco:

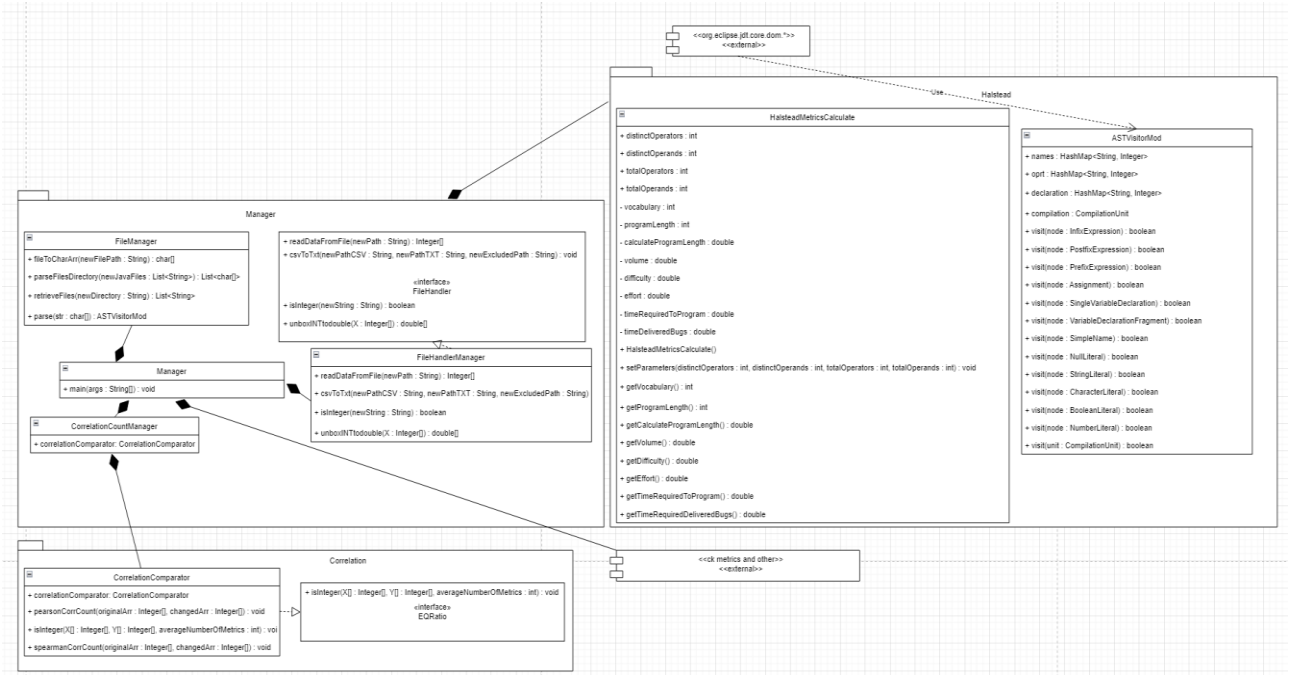
$$R_{ij} = EQ_{ij} / N_{ij},\ 0 \leq R_{ij} \leq 1$$

Wzór 1 – wzór pozwalający obliczyć współczynnik równoważności dla pary programów źródło [5]

gdzie:

- i, j – programy w parze
- EQ_{ij} – ilość ekwiwalencji
- N_{ij} – średnia liczba metryk użytych w obydwu programach
- R_{ij} – współczynnik równoważności

3.4 Struktura narzędzia



Rysunek 3 – diagram klas dla zbudowanego systemu
źródło [opracowanie własne]

Trzon programu stanowi klasa *Manager*, w której wykonuje się główny algorytm, a dodatkowo skupia w sobie wszystkie zachodzące procesy z nim związane, czy to bezpośrednio czy też przy pomocy referencji obiektów odpowiednich klas. W klasie *Manager* operują m.in: obiekty klas *FileManager*, *FileHandlerManager*, *CorrelationCountManager*, *HalsteadMetricsCalculate*, *ASTVisitorMod*. W niej też obsługiwane są metryki obliczone przez zewnętrzny program (narzędzie CK [27]).

Klasa *FileManager* jest stworzona w celu pomocy obliczenia metryk Halstead'a od strony „zajmowania" się plikami podjętymi analizie. Znając ścieżkę do katalogu, posiada ona metodę, która zwraca znajdujące się w nim pliki .java (algorytm będzie działał najlepiej, jeżeli będziemy porównywać ze sobą dwa pliki). Na ich podstawie możemy potem przystąpić do finalnych przygotowań, jeżeli chodzi o obliczenie metryk pod kątem plików (wykonanie metody *parse*, która zwraca obiekt, który będzie procesowany stricte pod obliczenie metryk), mianowicie obróbki przez metody *fileToCharArr* i *parseFilesDirectory*.

Klasa *FileHandlerManager* służy do obsługi plików, ale pod kątem „wyłuskiwania" z nich danych (*readDataFromFile*), przygotowania do sczytania przez program (*csvToTxt*, tutaj w kontekście metryk CK) tudzież sprawdzania możliwości potencjalnej konwersji danych z pliku na

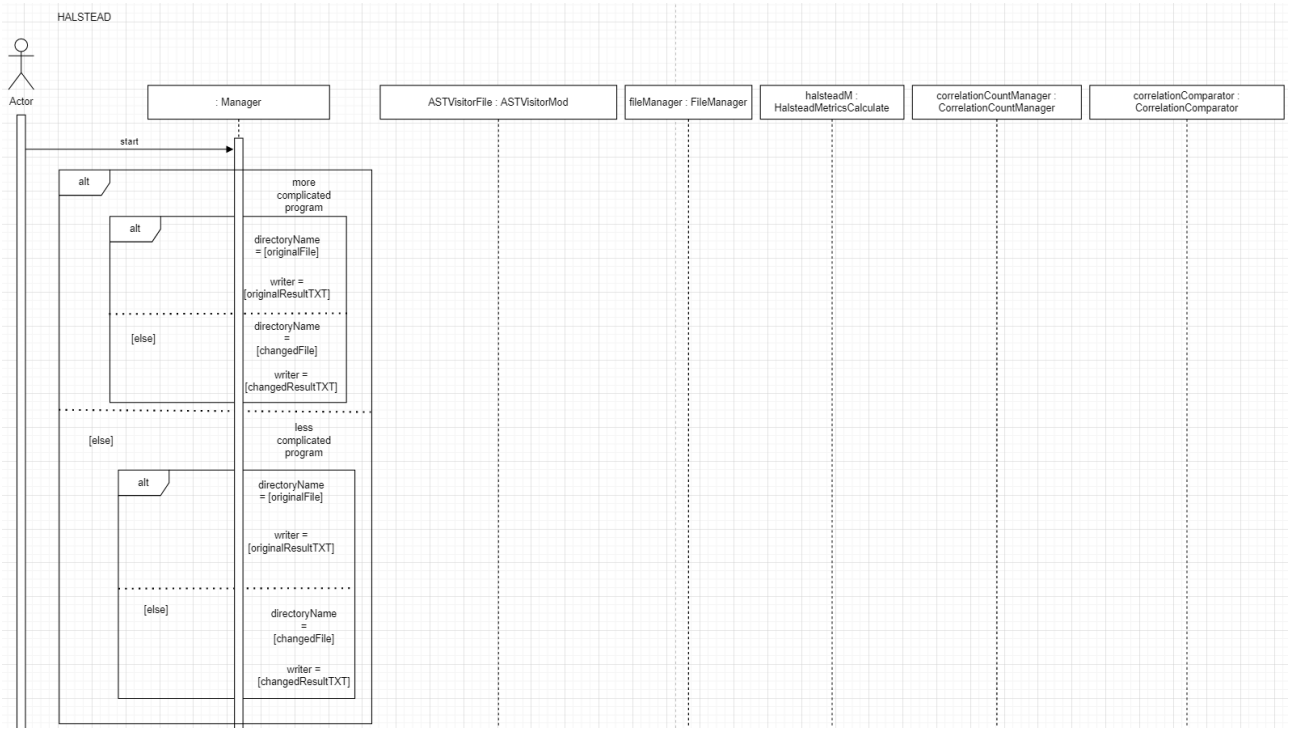
inny format (*isInteger*, *unboxINTtodouble*) – nie jest tak ściśle skorelowana z obliczaniem metryk Halstead'a, jak klas *FileManager*. Starano się, aby ta klasa została zaprojektowana z wykorzystaniem wzorca fasada [28]. Pomaga on w dostępie do potencjalnie złożonego systemu, poprzez uproszczenie interfejsu dostępu. Podczas jej projektowania myślano o przyszłych zmianach, które mogą zajść w projekcie, takie, jak: zmiana algorytmów analizy, język programowania systemu – jest to realizowanie również poprzez rozdzielony dostęp do głównej klasy, jak i obecność interfejsu.

Klasa *CorrelationCountManager* jest stworzona w celu analizy obliczonych metryk oprogramowania przy pomocy współczynnika równoważności. Starano się, aby spełniała warunki wzorca projektowego fasada przy pomocy klasy *CorrelationComparator*. Klasa *CorrelationComparator* mogłaby sama stanowić fasadę, lecz przez to, że klasa *CorrelationCountManager* tworzy bufor pomiędzy głównym algorytmem, a współczynnikiem równoważności, istnieje możliwość dodania w przyszłości innych metod analizy metryk, które nie wiązałby się stricte z obliczaniem korelacji – można by wtedy umieścić je w tym buforze. Klasa *CorrelationComparator* implementuje interfejs, który deklaruje współczynnik równoważności. Starano się, aby klasa ta czerpała korzyści z wzorca projektowego strategia [29], jednocześnie spełniała wymagania funkcyjnych interfejsów [30]. Wzorzec strategia sugeruje ekstrakcje algorytmów analizujących metryki i umieszczenie ich w osobnych konstruktach. Umożliwia to osobie używającej dany system wybór metody do analizy wedle własnego uznania, dodatkowo przyszłą rozbudowę systemu pod kątem dodania większej ilość różnych wzorów związanych z korelacją. Funkcyjne interfejsy natomiast mogą być implementowane przez wyrażenia lambda, które niosą za sobą niezaprzeczalne korzyści [31]:

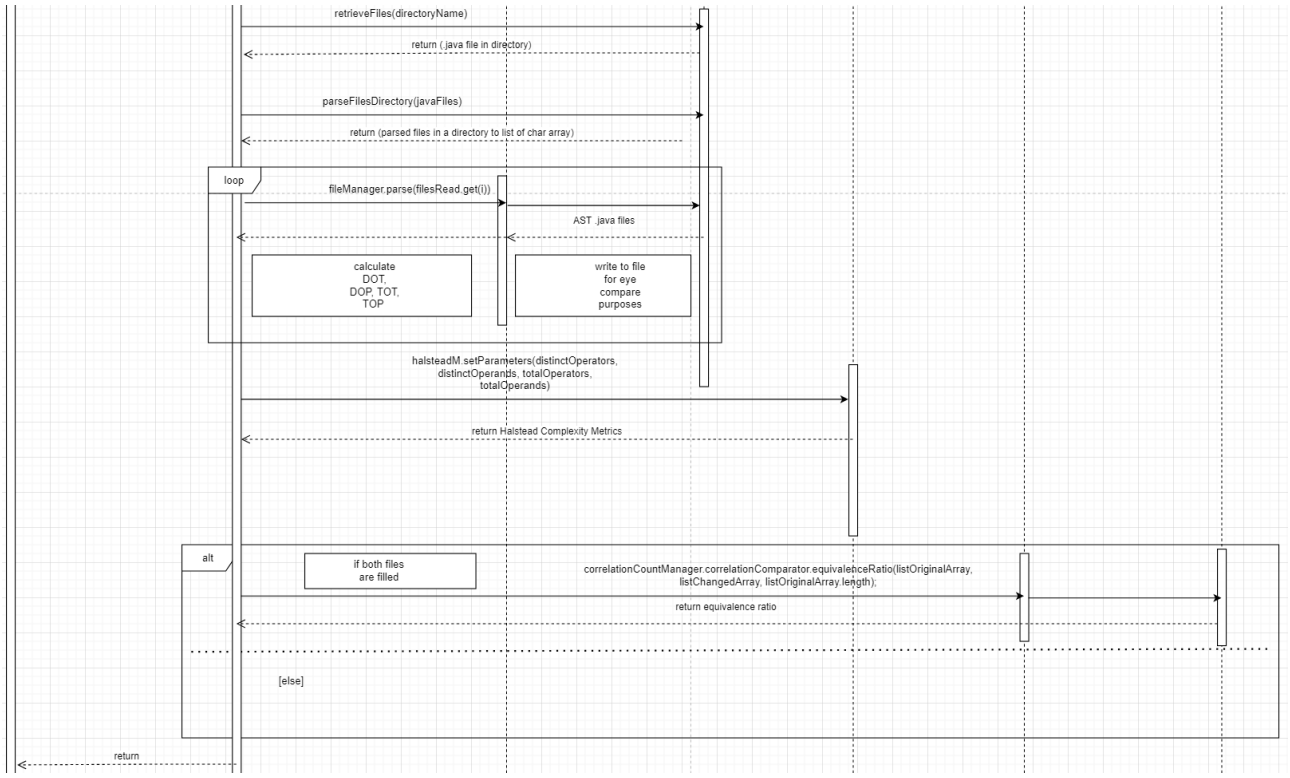
- kod jest bardziej zwięzły i czytelny
- umożliwia obsługę przetwarzania równoległego

Klasa *ASTVisitorMod* oraz *HalsteadMetricsCalculate* są ścisle związane z obliczaniem metryk Halstead'a. Są one uzyskiwane przy pomocy narzędzia [32]. Klasa *ASTVisitorMod* zawiera metody procesowania drzewa AST [33], które pomaga w analizie syntaktycznej kodu. Klasa *FileManager* buduje bezpośrednio dla tej klasy to drzewo. Klasa *HalsteadMetricsCalculate* dokonuje obliczenia metryk złożoności Halstead'a.

3.5 Interakcje obiektów w zbudowanym systemie



Rysunek 4 – diagram sekwencji dla zbudowanego systemu, część „Halstead” nr 1
źródło [opracowanie własne]



Rysunek 5 – diagram sekwencji dla zbudowanego systemu, część „Halstead” nr 2
źródło [opracowanie własne]

Użytkownik ma do wyboru czy chce liczyć metryki Halstead'a czy też metryki CK i je dalej procesować znajdując plagiat. Nie można jednocześnie obliczać metryk Halstead'a i je analizować, działając potem z metrykami CK. Poniżej będzie znajdywał się opis diagramu sekwencji dla metryk Halstead'a.

Już po wstępnym wybraniu opcji kontynuowania z metrykami Halstead'a pierwsze co ma miejsce to decyzja czy użytkownik chce procesować mniej zaawansowany program (w przypadku pracy - około 47 linii kodu, bez zaawansowanych konstrukcji języka Java, pojedyncza klasa, a w niej jedna metoda) czy bardziej zaawansowany program (w przypadku pracy - około 470 linii kodu, dziedziczenie, polimorfizm, wiele klas z wieloma metodami).

Kolejnym krokiem jest ustalenie właściwych ścieżek do plików (użytkownik podaje ścieżkę do pliku). Użytkownik podaje ścieżkę do pliku, który chce przeanalizować, program dodatkowo przypisuje ścieżkę do pliku gdzie będą trafiały dane (obliczone metryki) do celów dalszej analizy (analiza w postaci wzorów korelacji, porównania wzrokowego tudzież dowolnej dalszej obróbki przez użytkownika). Program dokonuje obliczenia metryk dla jednego programu, do przeprowadzenia analizy potrzebne są dwa programy więc do stworzenia całkowitej analizy potrzebne jest dwukrotne wykonanie programu zakładając, że użytkownik nie przygotował wcześniej zestawu danych. Jeżeli użytkownik przygotował jakiś zestaw danych program przeprowadzi analiza przy jednokrotnym uruchomieniu.

Po uprzednim stworzeniu obiektu klasy *FileManager* wyłuskujemy przy pomocy metody *retrieveFiles* pliki .java znajdujące się w podanej przez użytkownika ścieżce (program najlepiej będzie działał, jeżeli w podanej ścieżce będzie znajdował się jeden plik do analizy). Dla pliku, który otrzymaliśmy z wywołania metody *retrieveFiles* wywołujemy metodę *parseFilesDirectory*, która zamienia plik na sekwencję znaków, którą zapisuje do tablicy. Będzie to potrzebne do przyszłego stworzenia drzewa AST, dzięki któremu obliczymy metryki Halstead'a.

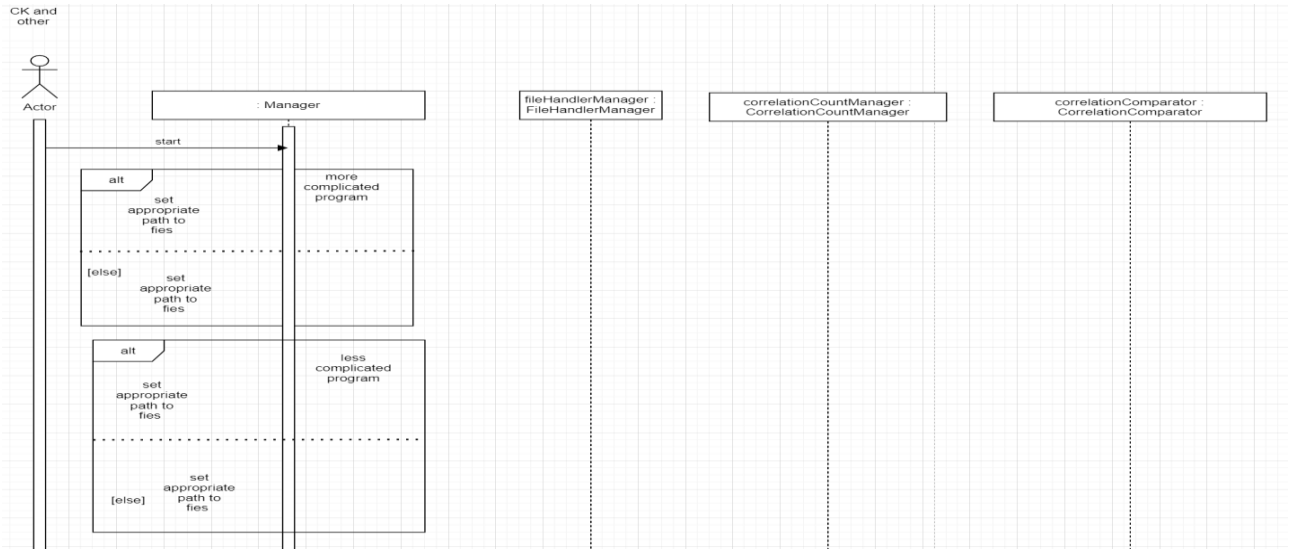
Następnie w pętli ma miejsce algorytm obliczania metryk Halstead'a. Najpierw tworzymy drzewo AST przy pomocy uprzednio stworzonej tablicy znaków z sczytanego programu. Uprzednio stworzony obiekt klasy *ASTVisitorMod* będzie korzystał w pętli z drzewa AST. Po jej zakończeniu będziemy znali następujące metryki dla programu:

- unikatowe operatory (z ang. number of distinct operators)
- unikatowe operandy (z ang. number of distinct operands)
- ogólną liczbę wystąpień operatorów (z ang. total number of operators)
- ogólną liczbę wystąpień operandów (z ang. total number of operands)

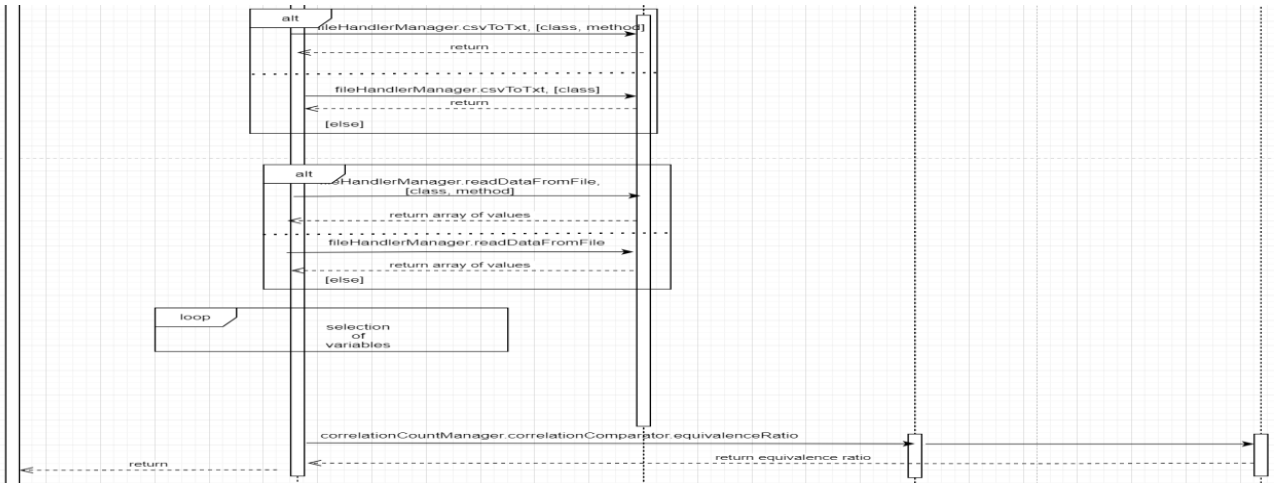
Użytkownik będzie miał możliwość wyboru, które metryki będzie chciał zapisać do pliku.

Kolejno tworzy się obiekt klasy *HalsteadMetricsCalculate*, który obliczy metryki złożoności Halstead'a przy użyciu uprzedni policzonych czterech wartości. Ustawiamy odpowiednie parametry dla klasy poprzez metodę *setParameters*, a następnie obliczamy pożądane wartości. Użytkownik będzie miał możliwość wyboru, które wartości będzie chciał zapisać.

Finalnym krokiem będzie analiza metryk obliczonych dla oryginału i dla programu podejrzewanego o popełnienie oszustwa. Tworzy się obiekt klasy *CorrelationCountManager*, który przy pomocy klasy *CorrelationComparator* będzie obliczał wartości wzorów korelacji dla obliczonych wartości metryk.



Rysunek 6 – diagram sekwencji dla zbudowanego systemu, część „CK" nr 1
źródło [opracowanie własne]



Rysunek 7 – diagram sekwencji dla zbudowanego systemu, część „CK" nr 2
źródło [opracowanie własne]

Jeżeli użytkownik wybierze procesowanie metryk CK, program będzie działa bardzo analogicznie. Jedyną główną zmianą będzie sposób obliczania metryk. Metryki będą obliczane z zewnątrz programu przy pomocy narzędzia [27].

Zewnętrzy program zapisuje dane w formacie *csv*. Zbudowane narzędzie przekonwertuje je do postaci *txt* przy pomocy funkcji *csvToTxt* w celu wczytania do programu. Obliczanie współczynników korelacji ma miejsce przy pomocy klasy *CorrelationCountManager*, analogicznie, jak w przypadku z metrykami Halstead'a.

4. Implementacja

Program stworzono i testowano w języku programowania Java. Zdecydowano się na taki ruch z wielu powodów. Do nich należą m.in:

- wieloparadygmatowość
- duże wsparcie społeczności internetowej
- duża ilość gotowych narzędzi, które usprawniają prace nad kodem
- relatywna „łatwość" tego języka
- ciągłe usprawnianie oraz dodawanie wielu funkcjonalności do języka
- niezależność od architektury
- typowanie statyczne

Wieloparadygmatowość Javy odgrywała dużą rolę podczas testowania zbudowanego urządzenia. Dzięki paradygmatowi obiektowemu, byliśmy w stanie obliczać metryki Chidamber & Kemerer, które to głównie na obiektowości bazują.

Niewątpliwie duże wsparcie społeczności internetowej oraz duża ilość gotowych narzędzi do użycia „od zaraz" bardzo pomaga w budowaniu, testowaniu i debugowaniu kodu. Dzięki bibliotekom związanymi z np.: obsługą plików, pisanie aplikacji, gdzie pewną jej częścią jest ich obsługa nie stanowi dużego problemu dla programisty. Obliczanie samych metryk w zbudowanym systemie jest liczone przy pomocy gotowych narzędzi – metryki Halstead'a [32], metryki CK i metryki rozmiaru [27]. Duża społeczność oraz ilość gotowych rozwiązań do napotkanych problemów też jest olbrzymim plusem przy pracy z tym narzędziem, można to zobaczyć np.: na stronie *Stack Overflow*.

Język Java został zaprojektowany tak, aby pisanie w nim programów było łatwiejsze niż programowanie np.: w języku C++. Jest to realizowane przez np.: dynamiczną alokacje pamięci, automatyczne usuwanie ze sterty nieużywanych już obiektów (jest to realizowane przez tzw. „garbage collector", gdzie dla porównania w języku C++ musimy dokładnie poinformować kompilator, że zwalniamy dynamiczne zaalokowaną pamięć), czy też poprzez brak możliwości dziedziczenia po wielu klasach, gdzie np.: w języku C++ mogło to generować niepotrzebne problemy, gdy zostało niepoprawnie użyte.

Poprzez ciągłe usprawnianie oraz dodawanie kolejnych funkcjonalności do języka można było użyć funkcyjnych interfejsów, które są skorelowane z użyciem wyrażeń lambda, niosących za sobą wiele korzyści.

System operacyjny, który posiada zainstalowaną maszynę wirtualną Javy (z ang. Java Virtual Machine (JVM)), będzie w stanie uruchomić aplikację w niej napisaną. Potencjalnie może to ułatwić uruchomienie zbudowanego narzędzia na innym komputerze.

Java to język typowany statycznie, znaczy to tyle, że, aby zadeklarować jakąś zmienną musimy podać jej typ – wygląda to inaczej niż w np.: języku Python. Typy danych mogą nieść ze sobą bardzo ważną informację w kontekście metryk oprogramowania i późniejszej analizy.

4.1 Liczenie metryk oprogramowania

Liczenie metryk oprogramowania w zbudowanym programie opiera się na dwóch narzędziach:

- narzędzie do liczenia metryk Halstead'a [32]
- narzędzie nazwane „CK" obliczające metryki Chidamber & Kemerer i metryki rozmiaru

Liczenie metryk Halstead'a jest niejako zawarte w kodzie, gdzie narzędzie „CK" i metryki, które ono produkuje jest wywoływane na zewnątrz programu przez instrukcje [27]:

java -jar ck-x.x.x-SNAPSHOT-jar-with-dependencies.jar <project dir> <use jars:true|false> <max files per partition, 0=automatic selection> <variables and fields metrics? True|False> <output dir> [ignored directories...]

gdzie:

- <project dir> - katalog, gdzie mają znaleźć się pliki .java do procesowania
- <user jars:true|false> - szukać jakichkolwiek plików .jar, aby użyć ich do lepszego rozpoznawania typów
- <max files per partition, 0=automatic selection> - informuje JDT (z ang. Eclipse Java development tools) o wielkości partii do przetworzenia
- <variables and fields metrics? True|False> - czy chcemy zmiennych na poziomie zmiennych i pól
- <output dir> - gdzie będą znajdywały się wyniki

Obydwa narzędzia do policzenia metryk od strony technicznej używają abstrakcyjnego konstruktu jakim jest drzewo składniowe (z ang. abstract syntax tree (AST)). Kiedy piszemy kod,

staramy się, aby był on zwięzły, jak najbardziej się da. Kompilatory, aby dokładnie zrozumieć kodu potrzebują znacznie więcej informacji. Aby zbudować drzewo składniowe z kodu najpierw musimy przeprowadzić jego analizę leksykalną poprzez tokenizację kodu. Potem po tokenizacji, bierzemy tokeny i parsujemy je do postaci drzewa składniowego. Otrzymane drzewo składniowe może być użyte przez np.: kompilator języka C++, aby przekształcić je w kod maszynowy. W naszym przypadku, drzewo składniowe będzie idealnym narzędziem do analizy jakikolwiek metryk, dlatego też jest używane przez te narzędzia.

W programie głównymi elementami zajmującymi się obliczaniem metryk są klasy *FileManager*, a dokładnie metoda *parse* oraz klasa *ASTVisitorMod*.

```
140         for (int i = 0; i < filesRead.size(); i++) {
141
142             System.out.println("AST parsing for : " + javaFiles.get(i));
143             ASTVisitorFile = fileManager.parse(filesRead.get(i));
144             distinctOperators += ASTVisitorFile.oprt.size();
145             distinctOperands += ASTVisitorFile.names.size();
146
147             operatorCount = 0;
148             for (int f : ASTVisitorFile.oprt.values()) {
149                 operatorCount += f;
150             }
151
152             totalOperators += operatorCount;
153
154             operandCount = 0;
155             for (int f : ASTVisitorFile.names.values()) {
156                 operandCount += f;
157             }
158
159             totalOperands += operandCount;
```

Rysunek 8 – fragment kodu gdzie tworzone jest drzewo składniowe
źródło [opracowanie własne]

Linia 143 na rysunku 12 prezentuje użycie metody *parse* na obiekcie *fileManager* przyjmując argument w postaci tablicy znaków stworzonych z przetworzonego pliku programu. Klasa *ASTVisitorMod* jest niczym innym, jak klasa *ASTParser* z biblioteki Javy [34]. Na podstawie tego drzewa możemy później w pętli zaprezentowanej na Rysunku 12 obliczyć takie parametry, jak: unikatowe operatory, unikatowe operandy, ogólną liczbę wystąpień operatorów i ogólną liczbę

wystąpień operandów, które są niezbędne do obliczenia metryk złożoności Halstead'a. Narzędzie „CK" używa podobnego algorytmu do obliczenia metryk.

4.2 Działania na plikach

W pracy na różne sposoby działano na plikach, zarówno od strony współpracy z obliczaniem metryk Halstead'a (klasa *FileManager*), jak i „zwykłym" działaniu (odczyt z pliku, zapis do pliku, konwersja formatów plików – klasa *FileHandlerManager*).

4.2.1 Działania na plikach związane z obliczeniem metryk

Obliczenie metryk Halstead'a zarówno dla programu mniej, jak i bardziej zaawansowanego zaczyna się od podania przez użytkownika systemu ścieżki do katalogu, w którym znajduje się plik .java dla którego chce obliczyć owe metryki. Ścieżka do katalogu jest przyjmowana przez zmienną *directoryName*, która jest typu *String*. W tym samym czasie, kiedy użytkownik poda ścieżkę do katalogu, do kolejnej zmiennej już *Writer* przypisywana jest ścieżka do pliku w którym będziemy przechowywać dane z obliczania metryk. Ten plik jest stworzony po to, ponieważ użytkownik może zechcieć zrobić z tymi metrykami co uważa np.: załadować ten plik do jakiegoś innego kalkulatora, który jest spoza programu. Dodatkowo można dokonać porównania organoleptycznego (wzrokowego) metryk, które też jest pewnym sposobem ich analizy pod kątem szukania plagiatu.
Najpierw sczytujemy pliki .java z podanego katalogu przez użytkownika przy pomocy metody *retrieveFiles*. Na rysunku 13 między liniami 75-79 wykonuje się pętla, która szuka wszystkich plików .java i zapisuje je do odpowiedniej listy. Program będzie najlepiej działał, jeżeli w katalogu będzie znajdował się pojedynczy plik .java.

```java
69      public List<String> retrieveFiles(String newDirectory) {
70
71          File newDir = new File(newDirectory);
72
73          List<String> newFiles = new ArrayList<~>();
74
75          for (File file : newDir.listFiles()) {
76              if (file.getName().endsWith((".java"))) {
77                  newFiles.add(file.getAbsolutePath());
78              }
79          }
80
81          return newFiles;
82      }
```

Rysunek 9 - fragment kodu gdzie zwracamy plik .java z podanego katalogu
źródło [opracowanie własne]

Następnie po tym kroku ma miejsce przetwarzanie plików (tokenizacja) do postaci gotowej do przetworzenia na drzewo składniowe (AST). Tokenizacja odbywa się przy pomocy dwóch metod: *parseFilesDirectory* oraz *fileToCharArr*. Na rysunku 14 w pętli między liniami 59-62 ma miejsce aplikacja metody *fileToCharArr* do plików .java, która zwraca plik w postaci tablicy znaków. Plik w postaci znaków będzie przygotowany do przekształcenia na drzewo AST, które jest niezbędne do obliczenia metryk Halstead'a.

```java
55 @    public List<char[]> parseFilesDirectory(List<String> newJavaFiles) throws IOException{
56
57          List<char[]> newFilesRead = new ArrayList<>();
58
59          for(int i = 0; i < newJavaFiles.size(); i++)
60          {
61              newFilesRead.add(fileToCharArr(newJavaFiles.get(i)));
62          }
63
64          return newFilesRead;
65      }
```

Rysunek 10 – parsowanie plików .java do postaci tablicy znaków
źródło [opracowanie własne]

4.2.2 Działania na plikach związane z przygotowaniem pod wzór korelacji

„Zwykłe” działania na plikach (odczyt, zapis) jest realizowane przez klasę *FileHandlerManager*, która implementuje interfejs *FileHandler*.

Metody *isInteger* oraz *unboxINTtodouble* odpowiadają za przystosowanie danych (już policzonych metryk) do obróbki przez współczynnik równoważności.

Zadaniem metod *readDataFromFile* oraz *csvToTxt* jest sczytanie już obliczonych metryk do programu w celu analizy przez współczynniki korelacji zaimplementowane w programie. Jako, że użytkownik może zrobić z obliczonymi metrykami (w plikach) co chce bez nich nie byłoby możliwe wyłuskanie wartości do analizy przez program (w przypadku metryk obliczonych przez narzędzie „CK”).

Narzędzie „CK” [27] generuje pliki *csv* w postaci:

file	class	type	cbo	cboModified	fanin	fanout	wmc	dit
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	0	0	0	0	2	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	enum	0	0	0	0	0	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	1	1	0	1	3	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	0	0	0	0	9	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	0	0	0	0	21	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	class	1	1	0	1	1	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	1	1	0	1	3	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	1	1	0	1	3	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	1	1	0	1	4	
C:\halstead-metrics\Halstead-Complexity-Measures\test_datasets\TestCorrectenessChanged10\Main5.java	com.plura	innerclass	8	8	0	8	18	

Rysunek 11 – generowanie plików *csv* przez narzędzie „CK”
źródło [opracowanie własne]

Na rysunku 16 znajduje się pętla, która jest fragmentem metody *csvToTxt*. Jest to główny trzon algorytmu przekształcającego plik *csv* do *txt*. Algorytm korzysta z wbudowanej klasy *BufferedReader*, dzięki której będziemy mogli „skanować” aż nie będzie jego końca i nie będzie pusty. Następnie przy pomocy *temporaryArray* będziemy sczytywać dane z poszczególnych kolumn, pilnując, aby dana nie była typu *String* oraz była konwertowalna do typu *Integer* (jest to kluczowe, jeżeli chodzi o dalsze działanie całego programu). Algorytm kończy się wypełnieniem pliku *txt* liczbami, które reprezentują obliczone metryki.

```
63          while ((line = newBufferedReader.readLine()) != null) {
64
65              temporaryArray = line.split( regex: ",");
66
67              // user for loop to iterate String Array and write data to text file
68              for (String str : temporaryArray){
69
70                  if(str instanceof String){
71                      continue;
72                  }
73
74                  // check if convertible to Integer (only Integers are written to file b
75                  if(isInteger(str) && !str.equals(newExcludedPath) && !str.equals("")){
76                      writer.write( str: str + "");
77                      // writer.write(System.lineSeparator());
78                      writer.write( str: "\n");
79                  }
80                  // writer.write(str + " ");
81              }
82              // Write each line of CSV file to multiple lines
83              // writer.write("\n");
84          }
85          System.out.println("test");
86          writer.close();
87      }
```

Rysunek 12 – konwersja pliku *csv* z wynikami z uruchomienia narzędzia „CK” do *txt* źródło [opracowanie własne]

4.3 Wzory korelacji

W programie metodą porównywania metryk jest obliczanie współczynnika równoważności Jest to realizowane przy pomocy dwóch klas: *CorrelationCountManager* oraz *CorrelationComparator*, gdzie *CorrelationComparator* implementuje funkcyjne interfejsy, które są stworzone stricte pod konkretny wzór korelacji.

W strumieniu głównego programu obiekt klasy *CorrelationCountManager* wywołuje obiekt klasy *CorrelationComparator* z tablicami wypełnionymi wartościami odpowiednich metryk. Klasa *CorrelationCountManager* stanowi bufor pomiędzy wywołaniami wzorów korelacji, ponieważ potencjalnie istnieje w przyszłości możliwość rozbudowy systemu poprzez dodanie dodatkowych opcji analizy metryk, które nie będą związane ze wzorami korelacji i one będą umieszczone właśnie w tym buforze.

Klasa *CorrelationComparator* implementuje trzy funkcyjne interfejsy (które deklarują tylko jedną metodę). Ma to na celu potencjalnie wykorzystanie wyrażeń lambda, które niosą za sobą dużo wartości optymalizacyjnych. Dodatkowo w przyszłości, istnieje możliwość odmiennej implementacji wzorów korelacji w klasie *CorrelationComparator*, ze względów np.: optymalizacyjnych. W programie użyto implementacji z biblioteki *org.apache.commons.math3.stat.correlation*.

5. Testowanie

Testy zaprojektowanego narzędzia odbyły się dla dwóch zestawów kodów źródłowych – jeden był zaawansowany, drugi mniej. Testowany był kod zmieniony z kodem oryginalnym. Dla każdego z oryginalnych kodów przygotowano pięć wersji zmienionego kodu gdzie poszczególne wersje różniły się od oryginalnego pod innym kątem, ale nie zmieniały zasady działania programu.

Symbole EQ1, EQ2, EQ3, EQ4 i EQ5 symbolizowały odpowiednio wyniki analizy plagiatowej dla kolejnych wersji zmienionego kodu względem oryginału.

Zdeformowano kody źródłowe w kolejnych wersjach w następujący sposób:

- zmiana nazw zmiennych, pętli *while* na *for* i dodanie komentarzy (wersja nr 1, wynik analizy – EQ1)
- dodanie pustych znaków, zmiana typów zmiennych oraz zmiana kolejności wykonania kodu (wersja nr 2, wynik analizy – EQ2)
- zmiana konstrukcji *if* na *switch* i na odwrót (wersja nr 3, wynik analizy – EQ3)
- zmiana nazw zmiennych, struktury kodu oraz tekstów w zmiennych tekstowych (wersja nr 4, wynik analizy – EQ4)
- połączenie wszystkich poprzednich zmian w jedno (wersja nr 5, wynik analizy – EQ5)

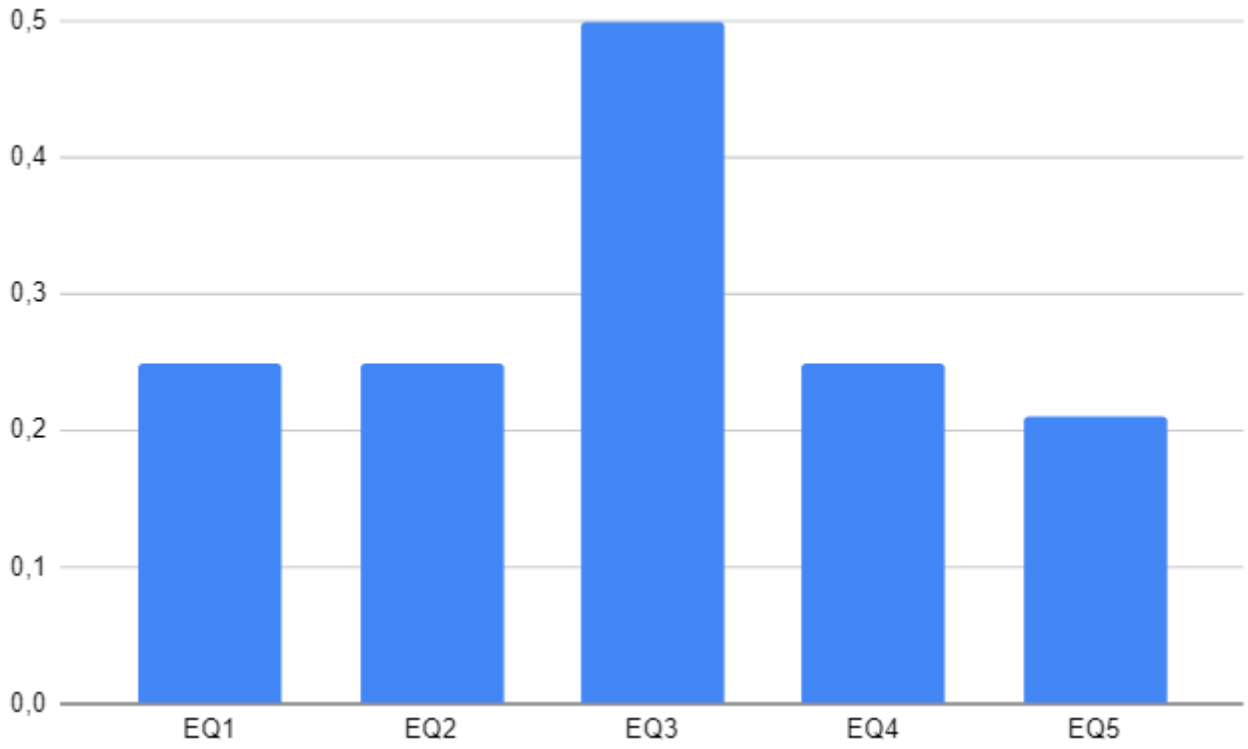
Kod mniej zaawansowany miał około 50 linijek, nie zawierał „zaawansowanych" konstrukcji języka Java, takich, jak np.: dziedziczenie, polimorfizm. Składał się z jednej klasy i jednej metody. Kod bardziej zaawansowany składał się z około 450 linijek, zawierał takie konstrukcje języka, jak dziedziczenie i polimorfizm, posiadał wiele klas i wiele metod.

Obliczone metryki wprowadzaliśmy do współczynnika równoważności (w testach oznaczany jako EQ). Im wyższa wartość współczynnika tym bardziej kody są ze sobą powiązane.

5.1 Przypadek testowy 1

Przypadek testowy 1 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk Halstead'a w wersji łatwiejszej kodu źródłowego (50 linijek kodu). Program poprawnie znajduje plagiat pomiędzy kodami (EQ1 = 0.25, EQ2 = 0.25, EQ3 = 0.5, EQ4 = 0.25 i EQ5 = 0.211). Różnice widać pomiędzy poszczególnymi wersjami analiz. Program w wersji nr 3 wykazuje największe podobieństwo względem oryginału. Jest to spowodowane tym, że w programie nie

wprowadzono do kodu takich zmian, które by powodowały drastyczną zmianę operatorów i operandów, na której to podstawie liczone są metryki Halstead'a. Z drugiej strony, przeprowadzenie w kodzie zmian z wszystkich poprzednich analizy i ich połączenie spowodowało zmianę tych wartości na tyle, że metryki Halstead'a różniły się w znacznym stopniu od oryginalnych wartości. Programy w wersji nr 1, 2 i 4 miały taką samą wartość współczynnika równoważności – zmiany zastosowane w tych programach miały podobny charakter pod kątem ilości operatorów i operandów w programie.

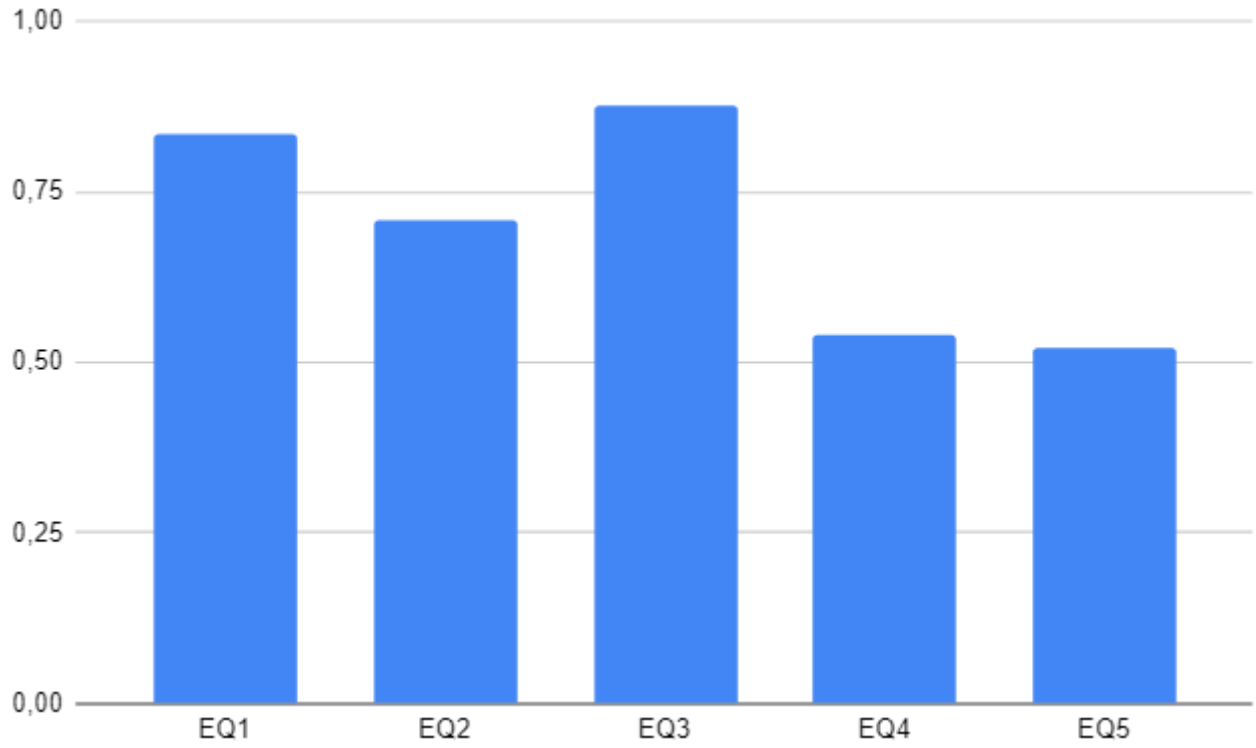


Rysunek 13 – testowanie dla metryk Halstead'a w wersji łatwiejszej
źródło [opracowanie własne]

5.2 Przypadek testowy 2

Przypadek testowy 2 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk CK i rozmiaru w wersji łatwiejszej kodu źródłowego (50 linijek kodu). Program poprawnie znajduje plagiat pomiędzy kodami (EQ1 = 0.83, EQ2 = 0.7083, EQ3 = 0.875, EQ4 = 0.5416 i EQ5 = 0.52). Różnice widać pomiędzy poszczególnymi wersjami analiz. Program w wersji nr 3 wykazuje największe podobieństwo względem oryginału, chociaż nie odbiega daleko od wyniku nr 1. Jest to spowodowane tym, że w programie w wersji nr 1 i 3 nie wprowadzono do kodu takich zmian, które by powodowały wystarczającą zmianę metryk CK (skupiają się wokół obiektowości, te zmiany stricte

nie wpływają na ten aspekt) i rozmiaru (zmiana nazw zmiennych oraz dodanie komentarzy ma znikomy wpływ na metryki rozmiaru – np.: parametr LOC nie bierze w ogóle pod uwagę komentarzy). Programy w wersji nr 4 i 5 miały najniższy współczynnik korelacji względem oryginału, gdzie program w wersji nr 5 miał go na poziomie najniższym. Jest to spowodowane tym, że program w wersji nr 5 aplikuje wszystkie zmiany, które miały miejsce we wcześniejszych wersjach, co znacznie wpływa na głównie na metryki rozmiaru, a program w wersji nr 4 wprowadza zmiany analogiczne, jak do programu w wersji nr 5. Manipulujemy tutaj metrykami rozmiarów oraz CK (poprzez np.: zamianę wywołania metody ciałem, operujemy na metrykach WMC i RFC). Program w wersji nr 2 wpasowuje się pomiędzy wersje wcześniej wymienione – nie wprowadza on takich „diametralnych" zmian, które angażują metryki (w tym wypadku głównie metryki rozmiaru), jak wersje 4 i 5, z drugiej strony zmiany zastosowane w tym programie nie są tak obojętne dla metryk, jak w wersjach nr 1 i 3.

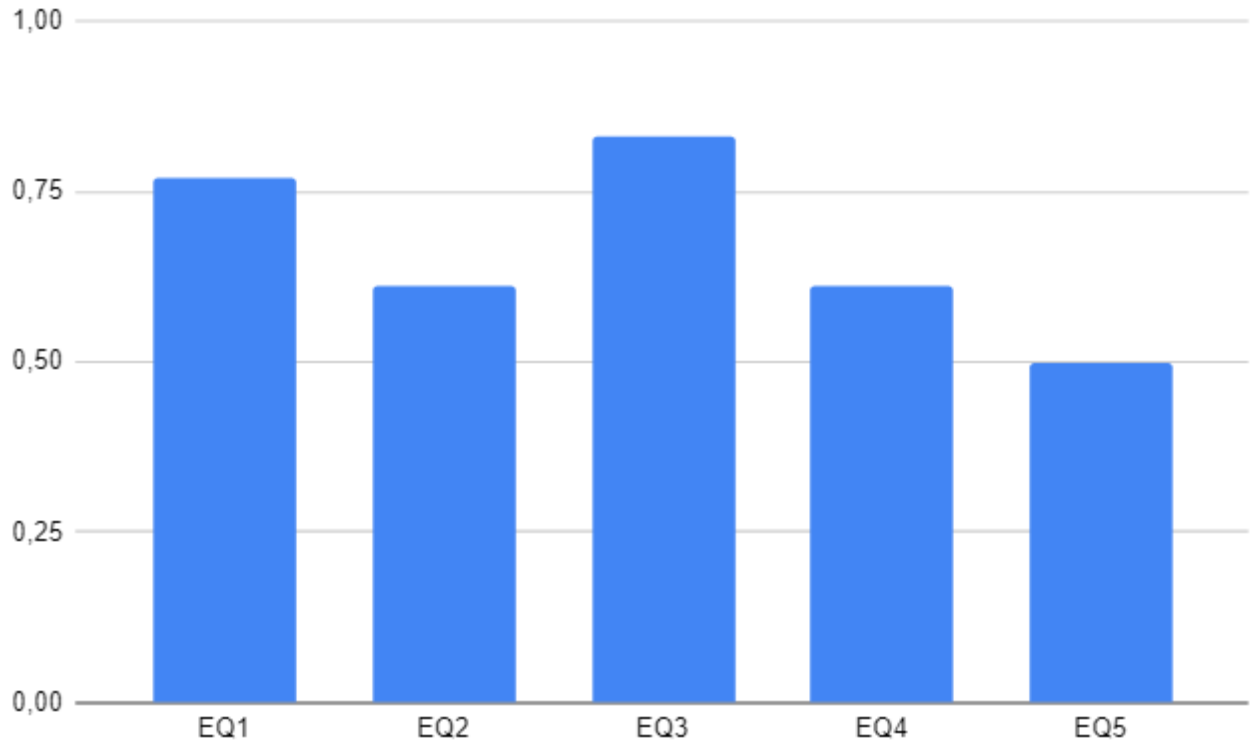


Rysunek 14 – testowanie dla metryk CK i rozmiaru w wersji łatwiejszej
źródło [opracowanie własne]

5.3 Przypadek testowy 3

Przypadek testowy 3 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk rozmiaru w wersji łatwiejszej kodu źródłowego (50 linijek kodu). Program poprawnie

znajduje plagiat pomiędzy kodami (EQ1 = 0.77, EQ2 = 0.61, EQ3 = 0.83, EQ4 = 0.61 i EQ5 = 0.5). Różnice widać pomiędzy poszczególnymi wersjami analiz. Wyniki są podobne, jak w przypadku testowym 2 – pokazuje to, że metryki CK nie były tak istotnym parametrem podczas obliczania korelacji dla poszczególnych wersji w wariancie łatwiejszym programów (warianty łatwiejsze programów nie zawierały w sobie dużo charakteru obiektowości), aczkolwiek nie były całkowicie obojętne na końcowy rezultat. W porównaniu z przypadkiem testowym 2 największe zmiany dotknęły wersji 2 i 4.

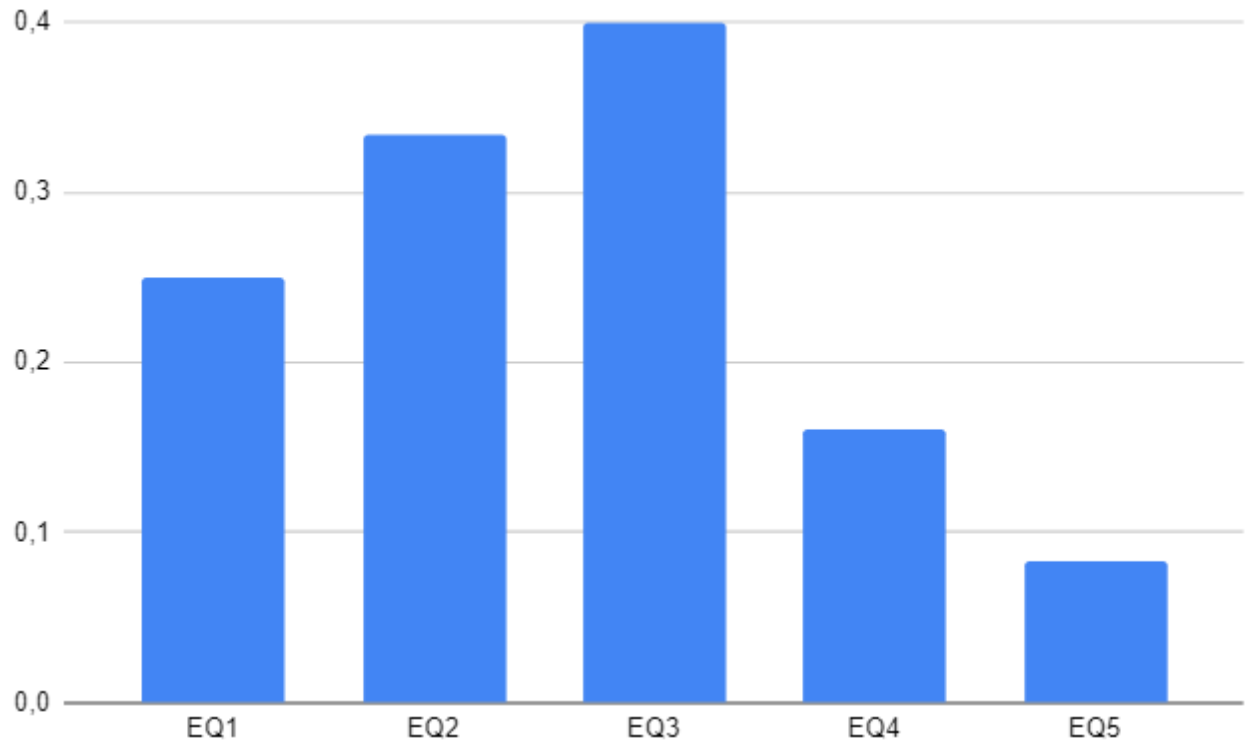


Rysunek 15 – testowanie dla metryk rozmiaru w wersji łatwiejszej
źródło [opracowanie własne]

5.4 Przypadek testowy 4

Przypadek testowy 4 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk Halstead'a w wersji trudniejszej kodu źródłowego (450 linijek kodu). Program poprawnie znajduje plagiat pomiędzy kodami (EQ1 = 0.25, EQ2 = 0.33, EQ3 = 0.4, EQ4 = 0.16 i EQ5 = 0.083). Różnice widać pomiędzy poszczególnymi wersjami analiz. Program w wersji nr 3 wykazuje największe podobieństwo względem oryginału. Jest to spowodowane tym, że w programie w wersji nr 3 nie wprowadzono do kodu takich zmian, które by powodowały drastyczną zmianę operatorów i operandów, na której to podstawie liczone są metryki Halstead'a – zamiana konstrukcji *if* na *switch* i na odwrót z perspektywy tak dużego programu nie jest tak angażująca dla operatorów i operandów,

szczególnie gdy nie zachodzą tam zmiany np.: zmiennych (operand), czy symboli (operatory). Programy w wersji nr 1 i 2 odpowiednio miały mniejszy i większy stopień korelacji względem kodu oryginalnego – zmiana np.: symboli oraz nazw zmiennych bardziej wpływa na zmianę ogólnej liczby operatorów i operandów niż np.: zamiana *if* na *switch*. Potęguje to też rozmiar programu. Programy w wersji nr 4 i 5 pokazały najniższy poziom korelacji względem oryginału, gdzie wersja nr 5 była najniższa. Jest to spowodowane faktem, że wprowadzały one zmiany połączone ze wcześniejszych analiz. Wynik „zbliżony" dla tych programów wynika z fakty, że aplikują one relatywnie podobne zmiany.

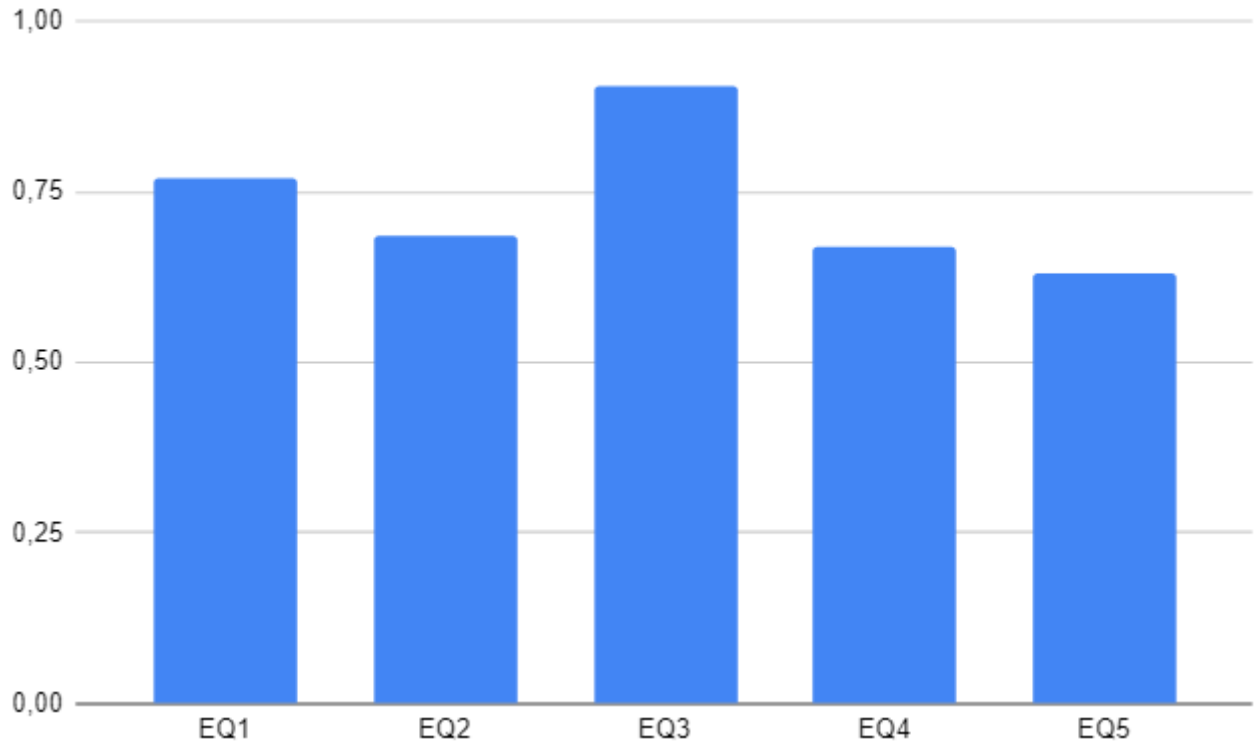


Rysunek 16 – testowanie dla metryk Halstead'a w wersji trudniejszej
źródło [opracowanie własne]

5.5 Przypadek testowy 5

Przypadek testowy 5 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk CK i rozmiaru w wersji trudniejszej kodu źródłowego (450 linijek kodu). Program poprawnie znajduje plagiat pomiędzy kodami (EQ1 = 0.77, EQ2 = 0.687, EQ3 = 0.9041, EQ4 = 0.67 i EQ5 = 0.63). Różnice widać pomiędzy poszczególnymi wersjami analiz. Program w wersji nr 3 wykazuje największe podobieństwo względem oryginału. Jest to spowodowane tym, że zamiana *if* na *switch* nie wprowadza aż tak dużych zmian w metrykach rozmiaru, nie mówiąc o metrykach z kontekstu

obiektowego (CK) – istotna też jest kwestia, że programie nie ma dużej ilości tych instrukcji. Pozostałe wersje programu prezentują relatywnie równy sobie, względnie wysoki poziom korelacji względem wersji oryginalnej. Wersje programów 4 i 5 znowu prezentują najniższe wyniki korelacji w zestawieniu – aplikują wszystkie zmiany, które miały miejsce wcześnie (ulegają zmianie metryki CK oraz rozmiaru). Relatywnie podobny niski wynik prezentuje wersja programu nr 2 – zmiany zastosowane w niej angażują metryki rozmiaru do zmiany. Względnie wysoki wynik dla każdej z wersji jest spowodowany tym, że bierzemy pod uwagę dużą ilość metryk – im więcej metryk porównujemy, zakładając, że mają tę samą wagę tym trudniej będzie oszukać system. Na ten fakt działa również poziom zaawansowania kodu.

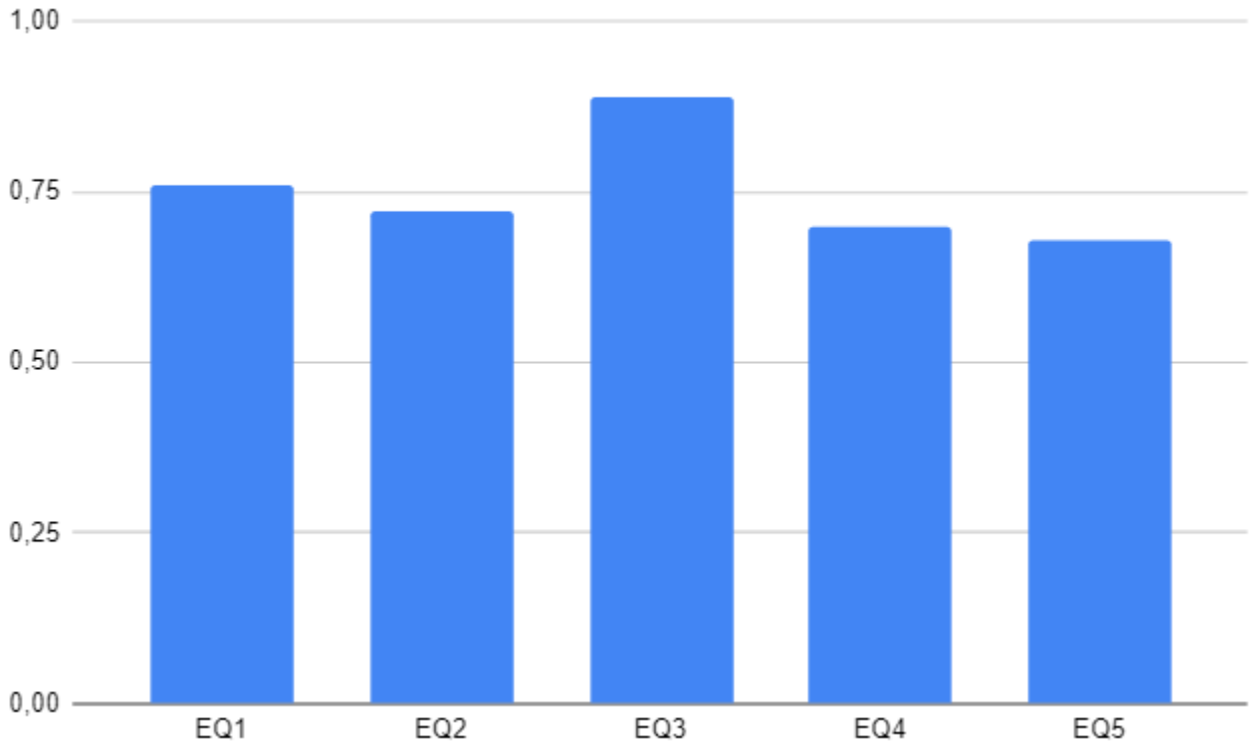


Rysunek 17 – testowanie dla metryk CK i rozmiaru w wersji trudniejszej
Źródło [opracowanie własne]

5.6 Przypadek testowy 6

Przypadek testowy 6 przedstawia wyniki obliczone przez współczynnik równoważności dla metryk rozmiaru w wersji trudniejszej kodu źródłowego (450 linijek kodu). Program poprawnie znajduje plagiat pomiędzy kodami (EQ1 = 0.76, EQ2 = 0.72, EQ3 = 0.89, EQ4 = 0.7 i EQ5 = 0.68). Różnice widać pomiędzy poszczególnymi wersjami analiz. Wyniki są relatywnie podobne, jak w przypadku testowym 5 – metryki CK grały istotną rolę podczas obliczania korelacji dla poszczególnych wersji w wariancie trudniejszym programów, aczkolwiek nie są czynnikiem

decydującym podczas obliczania korelacji ze względu na poziom skomplikowania programu oraz zastosowanie zmian głównie wpływających na metryki rozmiaru.



Rysunek 18 – testowanie dla metryk rozmiaru w wersji trudniejszej
źródło [opracowanie własne]

6. Podsumowanie

W ramach pracy stworzono strukturę i wykonano implementacje programu, który na podstawie obliczonych metryk oprogramowania analizuje je pod kątem zbieżności przy użyciu różnych narzędzi. Aplikacja pozwala na wybór plików do analizy, obliczenie dla nich metryk oprogramowania, oraz procesowania ich pod kątem szukania plagiatu przy użyciu wzoru korelacji. Również przez zapis danych do plików aplikacja może posłużyć użytkownikowi tylko do liczenia metryk, a potem ich użycia wedle uznania. Dodatkowo przedstawiono historię ewolucji zarówno metryk i jak i urządzeń, które służyły do posługiwania się nimi w kontekście znajdywania podobieństw oraz zarysowano, jak się je oblicza.

Analizując wyniki można powiedzieć, że aplikacja działa, ale nie jest wolna od wad. Dla każdej zmienionej wersji, niezależnie czy jest to bardziej lub mniej zaawansowany kod i od procesowanych metryk aplikacja wykrywa plagiat. Każdy z poszczególnych wyników analizy, więcej lub bardziej różni się od siebie – jest to spowodowane użyciem innych metryk, które są odrębnie wydajne dla poszczególnych kodów źródłowych (a do testów używano kody, które diametralnie różniły się ilością linii, jak i koncepcją). Warto mieć na uwadze, że również zmiany, które wprowadzono w poszczególnych wersjach programów inaczej będą oddziaływać na metryki, a więc finalnie na wynik. W programie użyto różnych metryk do analizy w celu przeprowadzenia badania na szeroką skalę oraz aby potencjalnie przyszłe programy, które będą testowane nie skompromitowały narzędzia. Wadą aplikacji jest niewątpliwie to, że może być podatna na kody, dla których ciężko będzie obliczyć wartościowe metryki. Kolejną rzeczą, która może zakłócić działanie aplikacji jest poziom skomplikowania kodu oraz ilość metryk – program będzie mniej dokładniejszy im mniej weźmie pod uwagę metryk tej samej wagi.

Możliwości dalszego rozwoju projektu są bardzo duże. Można dokonać zmiany języka programowania (np.: na bardziej efektywny pod kątem szybkości), użyć lepszych struktur w aplikacji, dodać obsługę błędów. Od strony metryk, można poszukać metryki, które lepiej opisywałby dany problem, bardziej pasowałby pod konkretny język programowania tudzież paradygmat. Analizując już policzone metryki też mamy gargantuiczny wybór z wachlarzu ulepszeń – używając, rozwijając metody analizy opisane w historii, które już miały miejsce albo kontynuując podejście zastosowane w programie i dodając kolejne wzory korelacji, które lepiej zachowywałyby się z konkretnymi zestawami danych.

7. Bibliografia

[1] Ottenstein K.J.: An algorithmic approach to the detection and prevention of plagiarism, ACM SIGCSE Bulletin, vol. 8(4), pp. 30–41, 1976. https://doi.org/ 10.1145/382222.382462, [dostęp: 05.12.2021]

[2] Finding Plagiarism among a Set of Programs with JPlag, Lutz Prechelt, Guido Malpohl, Michael Philippsen, Journal of Universal Computer Science, http://jucs.org/jucs_8_11/finding_plagiarisms_among_a/Prechelt_L.pdf [dostęp: 06.12.2021]

[3] Winnowing: Local Algorithms for Document Fingerprinting, Saul Schleimer, Daniel S. Wilkerson, Alex Aiken, http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf [dostęp: 07.12.2021]

[4] GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, Chao Liu, Chen Chen, Jiawei Han, Philip S. Yu, https://www.researchgate.net/publication/221653862_GPLAG_Detection_of_software_plagiarism_by_program_dependence_graph_analysis [dostęp: 07.12.2021]

[5] PlagDetect: A Java Programming Plagiarism Detection Tool, Al-Khanjari, Z.A, Fiaidhi, J.A., Al-Hinai, R.A., Kutti, N.S., https://dl.acm.org/doi/10.1145/1869746.1869766 [dostęp: 07.12.2021]

[6] A tool that detects plagiarism in Pascal programs, Sam Grier, https://doi.org/10.1145/800037.800954, [dostęp: 07.12.2021]

[7] IEEE standard for a software quality metrics methodology – IEEE Std 1061-1992, doi: 10.1109/IEEESTD.1993.115124 [dostęp: 08.12.2021]

[8] The Research on Software Metrics and Software Complexity Metrics, Tu Honglei, Sun Wei, Zhang Yanan, doi:10.1109/ifcsta.2009.39 [dostęp: 08.12.2021]

[9] A Complexity Measure, Thomas J. McCabe, doi: 10.1109/TSE.1976.233837 [dostęp: 09.12.2021]

[10] ‘Software Science’ revisited: rationalizing Halstead’s system using dimensionless units, David Flater, https://doi.org/10.6028/NIST.TN.1990 [dostęp: 10.12.2021]

[11] OOP with C++ Lecture Notes – Prof. Dharminder Kumar Edition, https://www.vidyarthiplus.com/vp/Thread-OOP-with-C-Lecture-Notes-Prof-Dharminder-Kumar-Edition [dostęp: 10.12.2021]

[12] Budgeting and accounting of software cost: Part 1, A. Seetharaman, M. Senthilvelmurugan, T. Subramanian [dostęp: 10.12.2021]

[13] http://lunatractor.com/blog/2011/03/24/tom-demarco-principles-of-control/, [dostęp: 11.12.2021]

[14] Software metrics and plagiarism detection, Geoff Whale, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.5047&rep=rep1&type=pdf [dostęp: 12.12.2021]

[15] M. H. Halstead, Elements of Software Science, North-Holland, New York, 1977, [dostęp: 12.12.2021]

[16] H. L. Berghel and D. L. Sallach, Measurements of Program Similarity in Identical Task Environments, https://doi.org/10.1145/988241.988245, [dostęp: 15.11.2021]

[17] H. T. Jankowitz, Detecting Plagiarism in Student Pascal Programs, https://doi.org/10.1093/comjnl/31.1.1, [dostęp: 16.11.2021]

[18] S. S. Robinson and M. L. Soffa, An Instructional Aid for Student Programs, [dostęp: 18.11.2021]

[19] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, [dostęp: 22.11.2021]

[20] Ewan Tempero, Emilia Mendes: The “CK” Metrics, https://web.archive.org/web/20100331064300/http://www.cs.auckland.ac.nz/compsci702s1c/lectures/ewan/cs702-notes-lec08-ck.pdf, [dostęp: 22.11.2021]

[21] https://lemelson.mit.edu/resources/john-backus, [dostęp: 22.11.2021]

[22] https://docs.oracle.com/javase/7/docs/technotes/guides/language/, [dostęp: 22.11.2021]

[23] http://sunnyday.mit.edu/16.355/metrics.pdf, [dostęp: 24.11.2021]

[24] http://www0.cs.ucl.ac.uk/staff/A.Finkelstein/d22.html, [dostęp: 24.11.2021]

[25] S.H. Kan, Metryki i modele w inżynierii jakości oprogramowania, [dostęp: 24.11.2021]

[26] A Plagiarism Detection System, John L. Donaldson, Ann-Marie Lancaster, Paula H. Sposato, [dostęp: 24.11.2021]

[27] https://github.com/mauricioaniche/ck, [dostęp: 28.11.2021]

[28] https://refactoring.guru/pl/design-patterns/facade, [dostęp: 28.11.2021]

[29] https://refactoring.guru/pl/design-patterns/strategy, [dostęp: 28.11.2021]

[30] https://java-8-tips.readthedocs.io/en/stable/funcinterfaces.html, [dostęp: 28.11.2021]

[31] https://www.tutorialspoint.com/why-we-use-lambda-expressions-in-java, [dostęp: 28.11.2021]

[32] https://github.com/aametwally/Halstead-Complexity-Measures, [dostęp: 28.11.2021]

[33] http://www.cs.put.poznan.pl/bpietrzak/sat/AbstractSyntaxTree.pdf, [dostęp: 28.11.2021]

[34] https://www.ibm.com/docs/he/rational-soft-arch/9.6.1?topic=SS8PJ7_9.6.1/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html, [dostęp: 28.11.2021]