PLAGIATY ROZWIĄZAŃ ZADAŃ PROGRAMISTYCZNYCH W ZDALNYM NAUCZANIU

Krzysztof Barteczko

PJWSTK

Treść

Temat: jak wykrywać i jak traktować plagiaty kodów w rozwiązaniach zadań przesyłanych przez studentów?

Cele: ➔ automatycznie odrzucić nie budzące wątpliwości przypadki ściągania

➔ wykryć kody podejrzane o Copy&Paste po to by zniechęcić do tej metody

Po kolei:

- Czym jest plagiat kodu? Copy & Paste
- Czego brak ogólnie dostępnym metodom wykrywania?
- Propozycja prostego podejścia do wykrywania

Prezentacja całkiem wstępnych wyników prac w zakresie analizy kodów pisanych w języku Java.

Plagiat

Kopaliński: przywłaszczenie cudzego pomysłu, dzieła, utworu, podanie ich (w całości lub części) za własne, opublikowanie ich pod własnym nazwiskiem.

Większość rozwiązań zadań programistycznych jest w tym ogólnym ujęciu plagiatami, bo:

- **studenci uczą się stosowania wymyślonych przez kogoś innego konstrukcji, technik, sposobów, algorytmów ...**
- **i stosują je w swoich rozwiązaniach (nie przywołując autorów :-)**

Plagiat kodu - definicja (1)

Parker, Hamblen: "program, który został wytworzony z innego programu za pomocą niewielkiej liczby rutynowych transformacji".

Dlaczego program? Przecież może być fragment programu.
Co to znaczy niewielka liczba?
Co to znaczy "rutynowe transformacje"?

Plagiat - definicja (2)

Rozwiązanie zadania programistycznego uznamy za **plagiat**, jeśli zawiera kod, który:

powstał przez **skopiowanie** kodu cudzego autorstwa i ewentualnie został zmodyfikowany za pomocą zmian, nie wymagających odpowiednio do treści zadania zaawansowanej wiedzy programistycznej

oraz

został świadomie przdstawiony jako kod własnego autorstwa

Copy & Paste a plagiaty

Powszechny wsród studentów sposób pisania programów: skopiować przykład, kod (z wykładu, z Internetu), realizujący jakąś funkcjonalność i dostosować go do swoich potrzeb.

"PO CO MAM WYWAŻAĆ OTWARTE DRZWI ?"

Nie zawsze wiąże się to z plagiatem, ale jest złą praktyką pisania programów i takie sytuacje warto też wykrywać i zniechęcać do nich.

Największym problemem jest jednak powszechne "ściąganie" rozwiązań (nawet na **studiach podyplomowych**). Na tym trzeba się skupić, bo to nie tylko jest nieuczciwe, ale

ŚCIĄGANIE ZAPRZECZA

CAŁEMU PROCESOWI DYDAKTYCZNEMU

Dlaczego studenci ściągają?

- niski poziom -> dla wielu ściąganie rozwiązań daje nadzieję na zaliczenie
- brak zainteresowania przedmiotem - pomyłka w wyborze kierunku studiów
- lenistwo
- wyrobione przyzwyczajenie z wcześniejszych etapów edukacji
- brak świadomości, że to jest coś niewłaściwego
- autentyczny brak czasu

Skutki:

brak wiedzy i umiejętności

utrwalanie dewiacji zachowań społecznych

Przeciwdziałać nie tylko ze względów etycznych, ale dla dobra samych studentów i jakości kształcenia

Jak ściągają studenci?

→ naiwne proste kopiowanie całych rozwiązań, czasem nawet bez modyfikacji sygnatur autorstwa (!), DOŚĆ CZĘSTE!

→ kopiowanie z następującą modyfikacją:

- nazw klas, zmiennych, funkcji,
- ogólnej struktury kodu (np. umieszczenie ściągniętego kodu w jednym pliku, zamiast w kilku jak oryginał),
- kolejności deklaracji i definicji, bez wpływu na logikę

→ kopiowanie z dopisaniem, usuwaniem, zmianą fragmentów

- w celu ukrycia faktu plagiatu,
- traktowane jako wlasny twórczy wkład (coś tam wiem, to zrobię)

Dostępne narzędzia wykrywania plagiatów

Przykładowe dostępne narzędzia (Jplag, MOSS, CodeMatch, CPD, ...).

Podejścia:

- dokładne porównywanie
- tokenizacja i wykrywania podobieństw
  - n-gramy
  - algorytmy: Robin-Karp, winnowing, itp.
- analiza AST,
- analiza PDG (program dependence graph)

Ogólnie:

porównanie metryk,

porównanie struktur,

porównanie logiki

Różna odporność na:

- zmiany nazw,
- zmiany kolejności,
- dopisywanie,
- zastępowanie instr.

Również pod hasłem wykrywania duplikatów kodu – ważne w procesie rozwoju aplikacji, systemów; nawet większe znaczenie niż wykrywanie plagiatow

Co mają a czego im brak?

Powszechnie dostępne rozwiązania

- akcentują efektywność przeszukiwania dużych baz kodów kosztem odporności na wiele rodzajów zmian kodu,
- mogą dawać fałszywie pozytywne wyniki (ze względu na rozwiązania wzmagające efektywność),
- mają ogólne nastawienie: wykryć jak najwięcej manipulacji
- są słabo konfigurowalne w zakresie indywidualnych potrzeb (co i jak traktujemy jako wyznaczniki podobieństwa kodów)
- są przygotowane do działania na indywidualnych plikach, a nie na zestawach plików danego rozwiązania (co jest częstym przypadkiem w Javie)

Kontekst bieżącej pracy

Niewielka baza kodów -> efektywność mniej ważna.

Unikać obiektywnych „false positive”, bo:

- proste kody ("znajdź max z tablicy"),
- kodowanie algorytmów zaczerpniętych z literatury ("sort").

E-learning zmniejsza nacisk na wykrywanie manipulacji.

Wykrywać na pewno to co jest na pewno plagiatem. I automatycznie odrzucać.

Propozycja (w fazie realizacji)

Własne, eksperymentalne, narzędzia umożliwiające:

- działanie w sytuacji, gdy rozwiązanie jest w wielu plikach
- dowolne zmiany w ustalaniu wyznaczników podobieństwa
- uwzględnienie specyficznych cech zadań do rozwiązania,
- eliminacja nie budzących wątpliwości plagiatów i to nawet w sytuacji, gdy kody są niewielkie/schematyczne i normalne środki kwalifikowałyby wszystkie rozwiązania jako plagiaty.

Bazujące na równoległych i wspierających się **podejściach**:

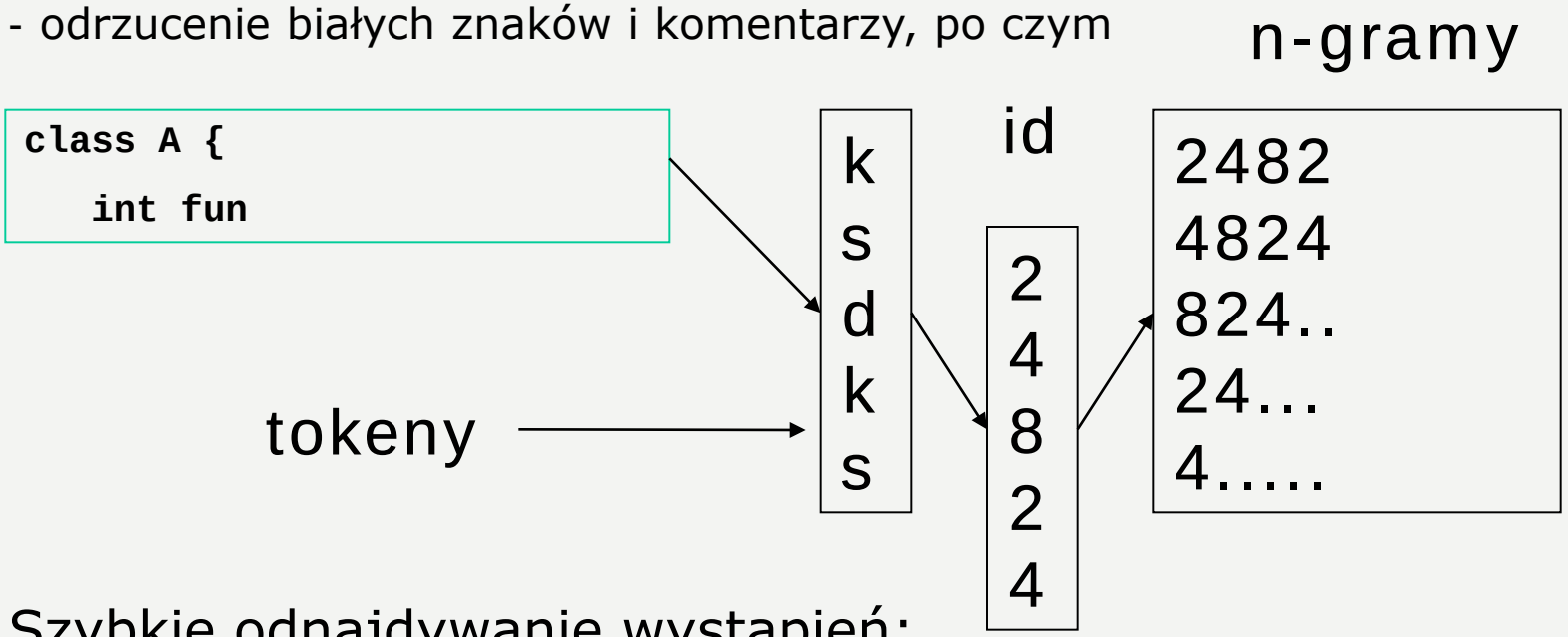
- **elastycznej tokenizacji** i wykrywaniu wspólnych fragmentów
- analizie **AST** z dodatkowym wykrywaniem podobieństw logiki
- porównań wyników w/w z **WZORCAMI**

Tokenizacja (1)

Zamiana kodów na ciągi symboli wg określonych reguł.

Zazwyczaj (w ogólnie dostępnych narzędziach):

- odrzucenie białych znaków i komentarzy, po czym



Szybkie odnajdywanie wystąpień:
mapa n-gram -> lista częstości wystąpień w plikach
Jak wybrać n?
EFEKTYWNOŚĆ na dużych zbiorach – próbkowanie n-gramów.

Tokenizacja (2)

Nie potrzebujemy (na razie):

- efektywności
- wykrywania wszelkich manipulacji.

Potrzebujemy (przede wszystkim):

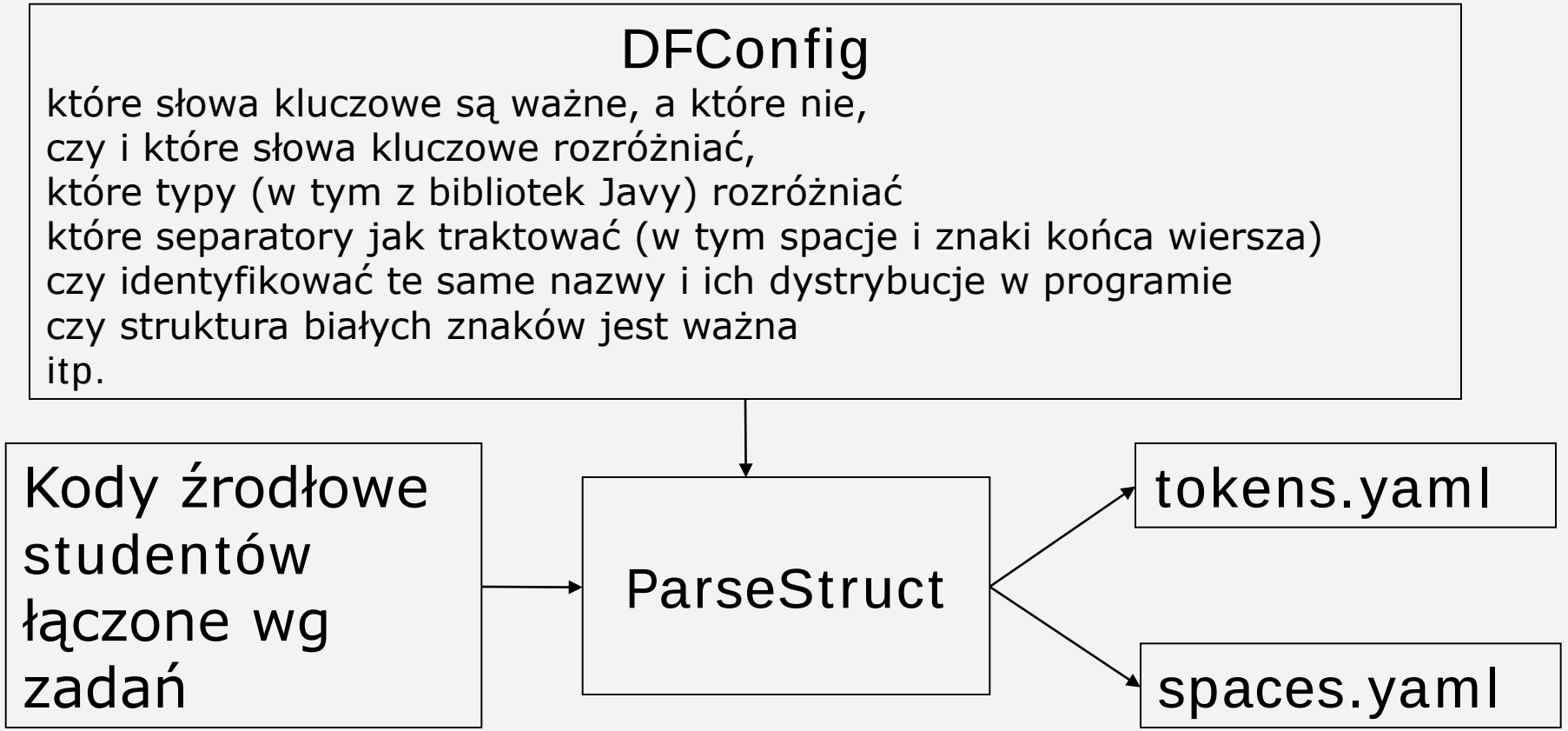
wykrycia oczywistych duplikatów

DLATEGO:

- **sposób tokenizacji powinien być dostosowany do sprawdzanych zadań,**
- **a porównywanie ciągów tokenów precyzyjne i naturalne**.

Tokenizacja (3)

Zatem: elastycznie konfigurowalna tokenizacja



Wykrywanie wspólnych fragmentów

Najprostszy z możliwych sposobów:

Longest Common Substring.

Wady:

- słaba efektywność (ale całkiem do przyjęcia przy stosunkowo niedużej bazie kodowej – np. 100 programów po 300 linii)
- brak odporności na przestawienia i dopisywania

DLATEGO MODYFIKACJA:

wielokrotne LCS (wyszukiwanie kolejnych najdłuższych wspólnych fragmentów) do chwili gdy długość kolejnego wspólnego fragmentu jest większa od 10% stokienizowanego kodu

Przykład – pismo na przyciskach

Stosunkowo proste zadanie (kody rozwiązań są niewielkie):

W oknie umieszczone są dwa przyciski z napisami "Przycisk 1" i "Przycisk 2".Kliknięcie w każdy z nich zwiększa rozmiar pisma na przycisku o 1 pkt.

Zapewnić:

aby zmiany pisma były widoczne natychmiast na przyciskach,

aby zawsze w oknie były widoczne oba całe przyciski (niedopuszczalne jest, by na skutek zwiększenia pisma w oknie była widoczna tylko część przycisku), przy czym szerokość i wysokość okna ma się płynnie dostosowywać do zmian rozmiarów przycisku (tzn. jeśli okno ma wystarczającą wysokość, ale brakuje mu szerokości, żeby pokazać cały przycisk - należy zmienić tylko szerokość i tylko o tyle ile trzeba; tak samo z wysokością)

Wyniki

Na 25 rozwiązań tylko w czterech rozwiązaniach metoda wykazała ok. 90% pokrycia kodów. Tylko te przypadki ekspert uznal za plagiaty.

Lp	A	B	SumLen	MaxLen	PrcA	PrcB	Ekspert
117	R18	R19	392	129	88	87	P
45	R8	R10	338	160	91	87	P
50	R8	R15	203	160	54	60	N
46	R8	R11	139	95	37	28	N
68	R10	R11	139	95	35	28	N
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
94	R13	R22	32	32	9	10	N
98	R14	R16	31	31	4	10	N
28	R6	R18	30	30	10	6	N
39	R7	R20	30	30	10	8	N
108	R16	R20	30	30	10	8	N
122	R18	R26	30	30	6	12	N
130	R23	R25	30	30	10	7	N
67	R9	R26	29	29	6	11	N
127	R19	R26	29	29	6	11	N
129	R21	R23	29	29	6	10	N
87	R11	R26	25	25	5	10	N

Wniosek: można tak ustawić czułość analizy, by z dużą dozą pewności automatycznie traktować za plagiaty rozwiązania z ponad 85% pokryciem kodu.

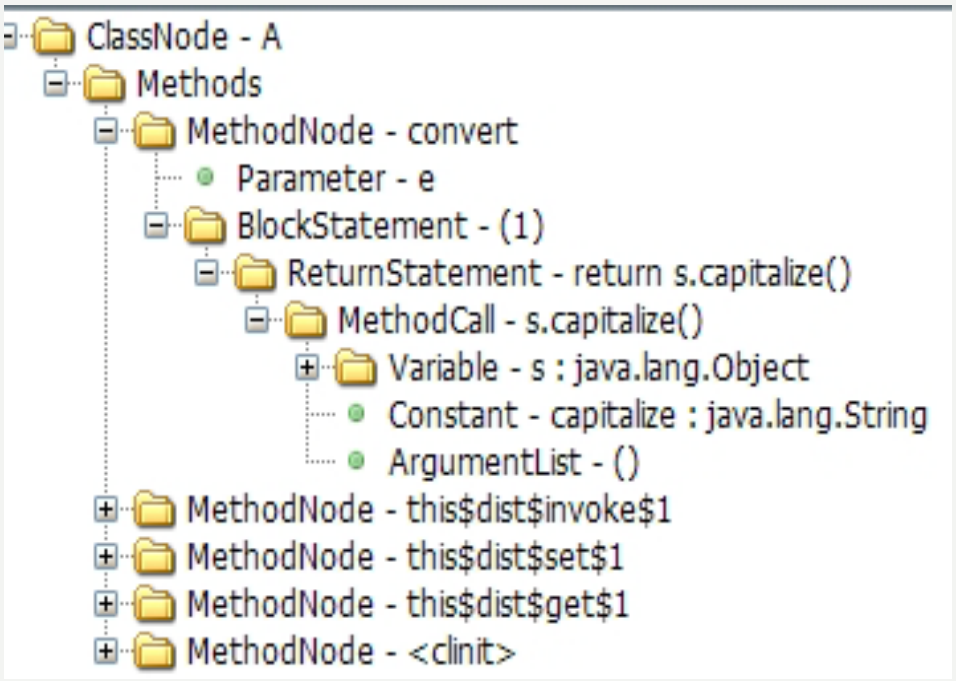
A jednak ...

(1) Gdy rozwiązanie zadania jest w wielu plikach z kodem klas ich połączenie jest przypadkowe (chyba, że zadanie specyfikuje dokładnie jak się klasy mają nazywać), wobec tego oceny długości wspólnych fragmentów mogą być bardzo zaniżone (np. gdy jest wiele małych plików-klas)

(2) Przenazywanie identyfikatorów mozna traktować tylko globalnie (w kolejności ich występowania w połączonych plikach) bez podziału na zakresy (np. nazwy klas, pól, metod, zmiennych lokalnych).

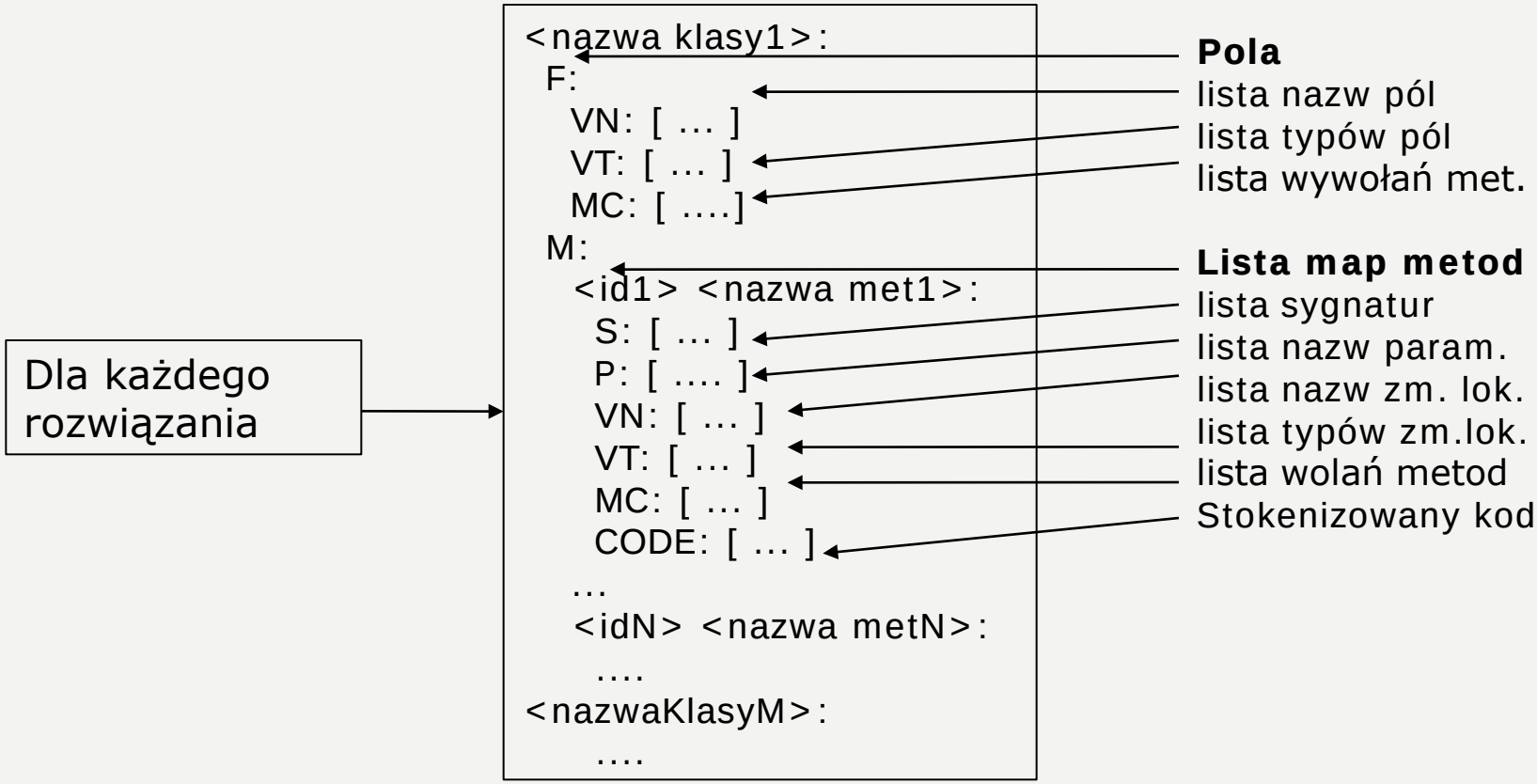
Analiza AST (1)

AST (Abstract Syntax Tree) – drzewo abstrakcyjnej struktury skladniowej kodu źródłowego.



Analiza AST (2)

Proponowane narzędzie łączy miary atrybutowe programów-rozwiązań z elastyczną tokenizacją kodów na poziomie metod.



Analiza AST (3)

Czego można się dowiedzieć i co porównywać:

- wszelkie nazwy w podziale na zakresy,
- typy w podziale na zakresy i miejsce użycia,
- wywolania metod (jakie, z jakich klas, w jakich blokach),
- kody poszczególnych metod (samą logikę lub ciągi tokenów)

Możliwość różnorakiego porównywania kodów rozwiązań w celu wykrycia duplikatów.

Przykład analizy AST (1)

Dla każdego rozwiązania budujemy zbiór opisów wszystkich metod:

$Z(i) = \{ [S, VT, MC, CODE] \}, i = 1..N$

i wyliczamy współczynnik podobieństwa Jacarda:

$$\mathbf{JC(i, j) = common(i,j) / (common(i,j) + diff(i,j) + diff(j,i))}$$

gdzie:

common(i,j) – liczba elementów przecięcia zbiorów Z(i) oraz Z(j)

diff(i,j) – liczba elementów zbioru Z(i) nie należących do Z(j)

diff(j,i) – liczba elementów zbioru Z(j) nie należących do Z(i)

Przykład analizy AST (2)

Dla niewielkiej liczby opisów metod JC mało informacyjny

Uwzględniając rozmiary opisów wyliczymy:

$$JS(i, j) = commL(i,j) / (commL(i,j) + diffL(i,j) + diffL(j,i))$$

$$prc(i)(i,j) = commL(i,j) / ZL(i) * 100$$

$$prc(j)(i,j) = commL(i,j) / ZL(j) * 100$$

gdzie:

commL(i,j) – dlugość połączonych elementów przec. Z(i) i Z(j)

diffL(i,j) – dlugość połączonych elementów różnicy Z(i) i Z(j)

diffL(j,i) – dlugość połączonych elementów różnicy Z(j) i Z(i)

ZL(i) – długość połaczonych elementów zbioru Z(l)

Przykład analizy AST (3)

Zadanie programistyczne:

napisać edytor tekstu z opcjami w menu

wczytywania, zapisu plików

zmiany kolorów tła i pisma

zmiany rozmiaru pisma

Spore kody od 300 do 500 wierszy, od kilkunastu do kilkudziesięciu metod.

Przykład analizy AST (4)

Wyniki:

Lp	A	B	JS	JC	Comm Count	ACount	BCount	Comm Size	PrcA	PrcB	Eks
105	R8	R26	1.00	1.00	37	37	37	1476	100.0	100.0	P
96	R8	R15	0.74	0.95	36	37	37	1264	85.6	84.7	P
174	R15	R26	0.74	0.95	36	37	37	1264	84.7	85.6	P
59	R6	R8	0.72	0.90	35	37	37	1236	83.2	83.7	P
74	R6	R26	0.72	0.90	35	37	37	1236	83.2	83.7	P
65	R6	R15	0.71	0.90	35	37	37	1236	83.2	82.8	P
120	R11	R12	0.31	0.82	9	10	10	768	48.6	45.7	N
114	R10	R19	0.18	0.74	14	16	17	391	30.7	29.2	N
X	X	X	X	X	X	X	X	X	X	X	X

Co z krótkimi prostymi zadaniami ?

Trudno napisać w bardzo różny sposób np. klasę Person z atrybutami name i age. A zdarzają się plagiaty!

Możliwości ich wykrycia:

- analiza AST w dwóch krokach: określenie zbioru rozwiązań różniących się od **wzorca**, porównanie tych rozwiązań
- identyfikacja takich samych osobliwości np. w formatowaniu, użyciu nawiasów, zestawie nieużywanych zmiennych