Politechnika Wrocławska

**Wydział Informatyki i Zarządzania**
kierunek studiów: Informatyka
specjalność: Inżynieria oprogramowania

Praca dyplomowa - magisterska

# Predykcja defektów na poziomie metod w celu zredukowania wysiłku związanego z zapewnieniem jakości oprogramowania

Mateusz Kutyba

słowa kluczowe:
predykcja defektów
metryki oprogramowania
oceny oparte na wysiłku

krótkie streszczenie:
Przewidywanie błędów w oprogramowaniu z wykorzystaniem algorytmów uczenia maszynowego, na podstawie metryk oprogramowania i danych historycznych.

| | | | |
|---|---|---|---|
| opiekun pracy | dr hab. inż. Lech Madeyski ..... | ..... | ..... |
| dyplomowej | *Tytuł/stopień naukowy/imię i nazwisko* | *ocena* | *podpis* |

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**
*a) kategorii A (akta wieczyste)*
*b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)*
** niepotrzebne skreślić*

Wrocław 2015

# Spis treści

**Streszczenie**

Streszczenie po polsku

**Abstract**

Abctract in english

# Rozdział 1

# Wstęp

Prawdopodobnie nie istnieją programy wolne od błędów. Z całą pewnością istnieją programy, które zawierają zbyt dużą liczbę błędów. Każdy kto tworzy oprogramowanie chciałby aby było ono wolne od wad. Podstawowym narzędziem pozwalającym na sprawdzenie czy program działa poprawnie są testy. Powstają coraz bardziej wyszukane metody i metodyki testowania oprogramowania, a wszystko po to aby oprogramowanie działało zgodnie z oczekiwaniami, czyli aby cechowało się wysoką jakością. Testowanie i inspekcje kodu pozwalają zapewnić odpowiednią jakość oprogramowania, ale są kosztowne. Bazując na zasadzie Pareto [12] wiemy, że około 80% defektów pochodzi z 20% modułów [8]. Wiedząc, które moduły należy poddać inspekcji, można znacznie obniżyć ilość pracy potrzebną do znalezienia większości błędów, a co za tym idzie znacząco zmniejszyć koszt takich inspekcji.

Kod źródłowy oprogramowania zazwyczaj składa się z wielu plików, które są organizowane w pakiety (ang. *packages*). W języku Java pliki zawierają klasy (ang. *class*), a każda klasa może zawierać metody (ang. *method*). Powstało wiele metod predykcji defektów, na różnych poziowach granulacji: od pakietów, przez pliki, klasy, metody, na pojedynczych zmianach (ang. *hunk* [13, 14]) skończywszy [6, 9, 10, 22]. Jak wykazano m.in. w [17] i [27], predykcja błędów na poziomie metod dostarcza dokładniejszych danych na temat lokalizacji błędów, dzięki czemu ich odnajdywanie jest efektywniejsze.

Predykcja defektów oprogramowania wykorzystuje techniki eksploracji danych, głównie są to metody statystyczne i metody uczenia maszynowego. Kluczowym elementem w tych procesach są właśnie dane. To na ich podstawie algorytmy uczenia maszynowego są w stanie formułować reguły decyzyjne. W inżynierii oprogramowania tymi danymi są róznego rodzaju metryki oprogramowania. Podział metryk oraz ich zastosowanie zostały szerzej opisane w rozdziale 2.3.

Gromadzenie danych (metryk) z projektów jest czasochłonne, wymaga dużo pracy — pobierania lub kopiowania projektów, mocy obliczeniowej do wyliczenia metryk. Potrzebne jest stworzenie uniwersalnych rozwiązań służących do tego celu oraz nastawienie na możliwość rozszerzania zestawu narzędzi, które mogą być ze sobą dowolnie zestawiane. Te wymagania spełnia platforma DePress [34], która jest rozwijana przy udziale studentów i pracowników Politechniki Wrocławskiej oraz pracowników Capgemini Polska. Więcej informacji o DePress zawarto w rozdziale 3.3.

Wstępne przeszukiwanie literatury wykazało niewielką ilość źródeł ściśle odpowiadających zagadnieniu predykcji defektów na niskim poziomie granulacji. Jest to główny

kierunek tych badań a ich celem jest przede wszystkim opracowanie nowego modelu, który miałby służyć do efektywnego wskazywania miejsc w oprogramowaniu, w których znajdują się błędy. Pozwoliłoby to na ograniczenie ilości pracy potrzebnej do przejrzenia krytycznych miejsc i naprawienia błędów.

## 1.1. Cele pracy

Cele pracy dyplomowej:

- Przegląd literatury pod kątem predykcji defektów oprogramowania, szczególnie na niskim poziomie granulacji.
- Budowa nowych lub rozbudowa istniejących narzędzi służących do wyliczenia metryk oprogramowania, współpracujących z wersjonowanymi repozytoriami kodu (wsparcie dla Git).
- Zebranie danych z projektów o otwartych źródłach na potrzeby predykcji defektów.
- Budowa modeli predykcji z wykorzystaniem zebranych danych.
- Ocena stworzonych narzędzi oraz zebranych danych.
- Ewaluacja modeli predykcji i ocena ich skuteczności.

### 1.1.1. Przeznaczenie narzędzi

Narzędzia stworzone w ramach pracy dyplomowej wchodzą w skład platformy (ang. *framework*) DePress[1] (*Defect Prediction in Software Systems*). Jest to rozszerzalna platforma pozwalająca na budowanie przepływu pracy (ang. *workflow*) w sposób graficzny, dzięki temu, że jest oparta na projekcie KNIME. Głównym celem DePress jest wspieranie analizy empirycznej oprogramowania. Pozwala na zbieranie, łączenie i analizę danych z różnych źródeł, jak repozytoria oprogramowania czy metryki.

### 1.1.2. Ograniczenia dotyczące realizacji

Poniżej wypisano ograniczenia dotyczące realizacji badań:

- badanie tylko projektów o otwartych źródłach (ang. *open source*),
- badanie tylko projektów napisanych w języku Java,
- wykorzystanie narzędzi platformy DePress, lub stworzenie nowych narzędzi w ramach platformy,
- możliwość łatwego, najlepiej zautomatyzowanego powtórzenia badania,
- wykorzystanie języka R do budowy modeli predykcji defektów.

### 1.1.3. Metoda oceny

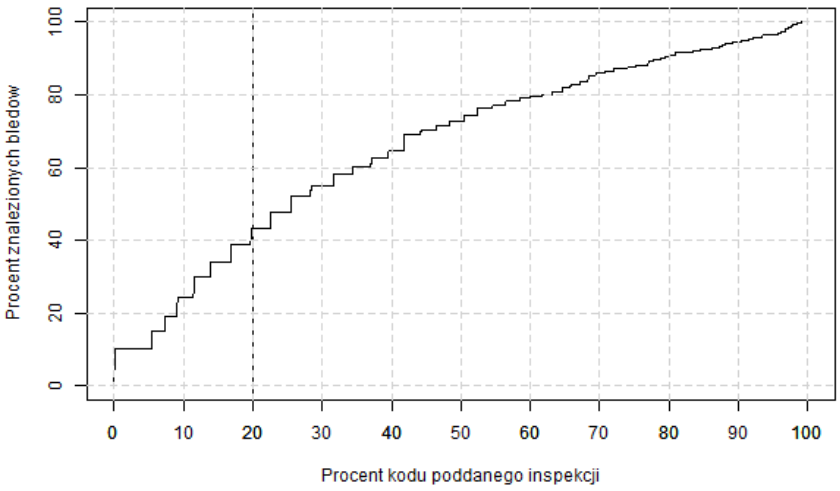
Podstawową oceną efektywności tworzonych modeli była ocena oparta na wysiłku (ang. *effort-based evaluation*). Dla każdego modelu została stworzona krzywa efektywności, przykład takiej krzywej przedstawia rysunek 1.1. Porównanie efektywności modeli polega przede wszystkim na porównaniu procentowej ilości błędów znalezionych w określonej ilości kodu. Przyjęto, żę wartością graniczną kodu poddanego inspekcji będzie 20%. Taka sama wartość jest stosowana w innych badaniach. W przedstawionym przykładzie dokonując przeglądu 20% kodu, znajdzie się w nim 30% encji z defektami (w przypadku tych badań są to metody).

Dodatkowo zastosowano inne miary skuteczności klasyfikatorów, w nawiasie zawarto symbol którym są oznaczane:

- dokładność ($A$),
- współczynnik Kappa Cohena ($\kappa$),

1. http://depress.io

Rys. 1.1. Wykres krzywej efektywności




- powierzchnia pod krzywą ROC (*AUC*).

**Dokładność,** ang. *Accuracy*, *A*. Jest to stosunek poprawnie sklasyfikowanych instancji do wszystkich instancji. Maksymalna dokładność wynosząca 1 oznacza całkowitą zgodność wyniku predykcji z rzeczywistymi klasami. Minimalna wartość to 0.

$$A = \frac{TP + TN}{TP + TN + FP + FN} \tag{1.1}$$

**Współczynnik Kappa Cohena,** $\kappa$. Jest miarą statystyczną określającą zgodność pomiędzy różnymi klasyfikatorami. Bierze pod uwagę przypadkową zgodność, dzięki czemu można określić czy dokładność przewyższa poziom losowej dokładności. Współczynnik sprawdza się dobrze w problemach gdzie liczność instancji w poszczególnych klasach nie jest równa. Maksymalna wartość Kappa to 1 a minimalna to -1.

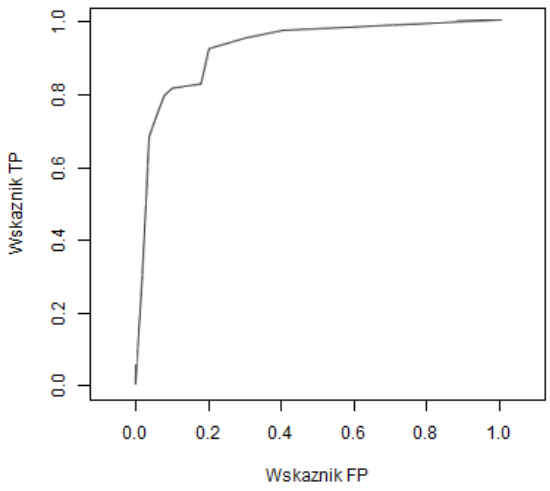
$$\kappa = \frac{A - RA}{1 - RA} \qquad \text{gdzie } RA = \frac{(TP + FN)(TP + FP) + (TN + FP)(TN + FN)}{(TP + TN + FP + FN)^2} \tag{1.2}$$

**Powierzchnia pod krzywą ROC,** ang. *Area Under the Curve*, *AUC*. Klasyfikatory nie określają samej przynależności do klasy, ale wartość prawdopodobieństwa z jakim dana instancja należy do danej klasy. Daje to możliwość wykreślenia krzywej zależności pomiędzy TP i FP. Krzywa na wykresie określana jest jako ROC (ang. *Receiver Operating Characteristic*). Pole pod tą krzywą reprezentuje skuteczność klasyfikatora. Idealny klasyfikator uzyska wynik 1, natomiast losowy klasyfikator powinien uzyskać wynik 0,5. Wykres 1.2 przedstawia przykład krzywej ROC.

## 1.2. Struktura pracy

Dalsza część pracy została podzielona w następujący sposób. W rozdziale 2 opisano przegląd literatury oraz omówiono aktualny stan wiedzy. W rozdziale 3 zawarto charakterystykę wykorzystanych narzędzi, oprogramowania i języków. W rozdziale 4 opisano przebieg badań i ich wyniki, natomiast w rozdziale 5 przeanalizowano uzyskane rezultaty, podsumowano badanie pod kątem jego potencjalnego zastosowania i możliwości

Rys. 1.2. Wykres krzywej ROC




dalszego rozwoju. Rozdział ten zawiera również istotny fragment dotyczący zagrożeń dla wiarygodności przeprowadzonego badania.

# Rozdział 2

# Przegląd literatury

## 2.1. Związek z innymi pracami

Na początku prac dokonano przeglądu literatury aby określić aktualny stan wiedzy (ang. *state of the art*) w badanej dziedzinie. Przegląd literatury pozwolił udzielić odpowiedzi na następujące pytania:

- **Jakie istnieją metody predykcji defektów na poziomie metod i jaka jest ich skuteczność?**
  Istnieje wiele modeli predykcji defektów, jednak większość z nich opiera się na danych dotyczących klas, pakietów lub modułów. Odpowiedzią na powyższe pytanie jest zbiór modeli predykcji defektów na niskim poziomie granulacji, na przykład metod lub bloków kodu.
- **Jakie są możliwości usprawnienia lub rozwinięcia istniejących metod?**
  Zostały zebrane wszelkie możliwości ulepszenia lub rozszerzenia badań wskazanych przez autorów, określone np. jako "Dalszy rozwój".
- **Jakie są sposoby ekstrakcji zmian kodu źródłowego na poziomie metod?**
  Jakie są sposoby porównywania wersji kodu źródłowego, jakiego rodzaju dane (metryki) są uzyskiwane.

Podczas wstępnego rozpoznania dziedziny zauważono, że liczba publikacji jest niewielka. W związku z tym postanowiono przeprowadzić wyszukiwanie w dwóch etapach. W pierwszym etapie przeszukano elektroniczne zbiory, natomiast w drugim etapie przejrzano bibliografie pozyskanych publikacji a także wszystkie publikacje ich autorów w celu odnalezienia dodatkowych tekstów.

**Przeszukiwalne zbiory cyfrowe.** Przeszukano poniższe zbiory z użyciem ustalonych wyrażeń, za pomocą wyszukiwarek udostępnianych w postaci aplikacji internetowej:

- IEEE Xplore,
- Science Direct,
- ACM Digital Library,
- Springer Link,
- ISI Web of Science.

Wybrano te zbiory ponieważ pokrywają one większość publikacji inżynierii oprogramowania oraz są używane jako źródła w innych przeglądach z tej dziedziny [22, 43].

**Szara literatura.** Wstępne przeszukiwanie wykazało niewielką ilość źródeł ściśle odpowiadających zagadnieniu predykcji defektów na niskim poziomie granulacji. Aby

pokryć znaczną część szarej literatury zdecydowano, aby przeszukać następujące źródła:

- Google Scholar.
- Lista odnośników w znalezionych źródłach pierwotnych.
  Zgodnie z metodą śnieżnej kuli [20] przejrzano listy referencji źródeł pierwotnych w celu odnalezienia dodatkowych istotnych (relewantnych) publikacji.
- Inne publikacje autorów znalezionych źródeł pierwotnych.
  Przeszukano bazę DBLP [32] szukając według nazwisk autorów dotychczas zgromadzonych źródeł pierwotnych.

Po dokonaniu przeglądu literatury oraz oceny znalezionych źródeł, wybrano te najbardziej istotne z punktu widzenia niniejszej pracy:

- *Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes.* [11]
- *Method-level bug prediction.* [17]
- *Comparing fine-grained source code changes and code churn for bug prediction.* [18]
- *Fault-prone Module Prediction Using Version Histories.* [24]
- *Reconstructing fine-grained versioning repositories with git for method-level bug prediction.* [25]
- *Historage: fine-grained version control system for Java.* [26]
- *Bug prediction based on fine-grained module histories.* [27]

## 2.2. Eksploracja danych

Obecnie na świecie gwomadzi się ogrmone ilości cyfrowych danych. Dane są zbierane na każdym kroku. Według badania IDC Digital Universe [16] w 2012 roku cyfrowy wszechświat osiągnął rozmiar 2,8 zettabajtów (1 ZB = $10^{21}$ B), a w latach 2012 do 2020 roku rozmiary cyfrowego wszechświata będą się podwajać co dwa lata. Dane są zbierane na każdym kroku: bank zapisuje wszystkie nasze operacje finansowe — wpłaty, wypłaty, przelewy, historię kredytu, płatności kartą, itd.; podczas przeglądania internetu narzędzia analityczne zapisują każdy nasz krok; firma handlowa zapisuje w systemie CRM (ang. *Customer relationship management*) interakcje z klientami, a w systemie finansowo-księgowym informacje o sprzedaży, zakupach, produktach w magazynie, itd.; dostawca internetu (ISP) zapisuje wszystkie nasze żądania w logach.

Ogromne ilości danych powodują niemożliwość ich analizy przez ludzki rozum oraz odseparowanie użytecznych danych od bezwartościowych. Zgodnie z przywołanym raportem w 2012 roku można było wykorzystać 23% wszystkich danych, pod warunkiem, że byłyby one otagowane i przeanalizowane. Jednak tylko 3% potencjalnie użytecznych danych było otagowane a 0,5% analizowane. Z pomocą przychodzi dziedzina zwana eksploracją danych. Opiera się ona na wykorzystaniu szybkości komputera do znajdowania niewidocznych dla człowieka prawidłowości w zgromadzonych danych. Przykładowe obszary zastosowania eksploracji danych:

- meteorologia — prognozowanie pogody,
- ekonomia — rozpoznawanie trendów na rynkach finansowych,
- medycyna — stawianie diagnozy na podstawie symptomów,
- marketing — tworzenie reklam dopasowanych do odbiorcy,
- bankowość — ocena ryzyka kredytowego,
- biotechnologia — analiza danych genetycznych.

Powyższa lista to bardzo mały wycinek możliwości zastosowania eksploracji danych w dzisiejszym świecie.

Klasyfikacja metod eksploracji danych ze względu na cel eksploracji [39]:

- Odkrywanie asocjacji — odkrywanie interesujących zależności lub korelacji.
- Klasyfikacja i predykcja — odkrywanie modeli opisujących zależności pomiędzy klasyfikacją obiektów a ich charakterystyką, w celu ich wykorzystania do klasyfikacji nowych obiektów.
- Grupowanie — znajdowanie zbiorów obiektów mających podobne cechy.
- Analiza sekwencji i przebiegów czasowych — znajdowanie częstych podsekwencji, trendów, podobieństw, anomalii oraz cykli.
- Odkrywanie charakterystyk — znajdowanie zwięzłych opisów ogólnych własności klas obiektów.
- Eksploracja tekstu i danych semistrukturalnych — analiza danych tekstowych w celu ich grupowania, klasyfikacji, wsparcia przeszukiwania.
- Eksploracja www — znajdowanie wzorców zachowań użytkowników Internetu.
- Eksploracja grafów i sieci społecznościowych — analiza struktur grafowych, które są szeroko wykorzystywane do modelowania złożonych obiektów, takich jak: związki chemiczne, struktury białkowe, sieci społecznościowe, sieci biologiczne, itd.
- Eksploracja danych multimedialnych i danych przestrzennych — analiza i eksploracja danych obejmujących obrazy, mapy, dźwięki, filmy, itp.
- Wykrywanie punktów osobliwych — wykrywanie obiektów, które odbiegają od ogólnego modelu.

W niniejszej pracy wykorzystano metody należące do grupy klasyfikacji i predykcji. Klasyfikacja polega na przypisaniu zadanych elementów do ustalonych klas. Każdy element może być przypisany tylko do jednej klasy. W zadaniu predykcji defektów oprogramowania można wyróżnić dwie klasy: "zawiera błąd"(oznaczana dalej jako *1*, *pozytywna*, *true*) i "nie zawiera błędu"(oznaczana dalej jako *0*, *negatywna*, *false*). Jest to szczególny przypadek klasyfikacji, nazywany klasyfikacją binarną. Wynik klasyfikacji można przedstawić w postaci macierzy pomyłek.

**Tabela** 2.1. Przykład macierzy pomyłek

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | $TP$ | $FN$ |
| | 0 | $FP$ | $TN$ |

Tabela 2.1 jest przykładową macierzą pomyłek, w której wartości liczbowe w komórkach zostały zastąpione etykietami oznaczającymi następująco:

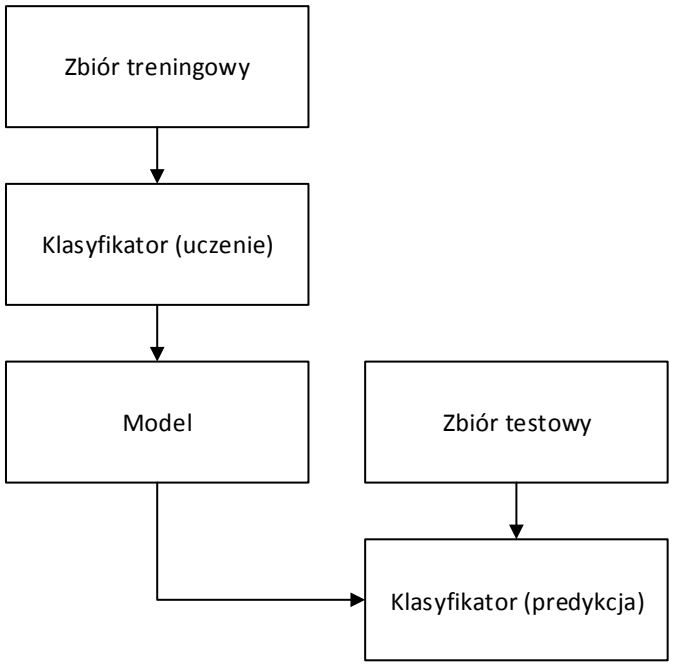
- TP (*True Positive*) — element poprawnie sklasyfikowany jako pozytywny;
- TN (*True Negative*) — element poprawnie sklasyfikowany jako negatywny;
- FP (*False Positive*) — element błędnie sklasyfikowany jako pozytywny;
- FN (*False Negative*) — element błędnie sklasyfikowany jako negatywny.

### 2.2.1. Uczenie maszynowe i klasyfikacja

Uczenie się maszyn jest dziedziną nauki z obszaru sztucznej inteligencji. Polega ono na zastosowaniu algorytmu, który na podstawie danych wejściowych ma za zadanie dostarczać wiedzę i wnioski, a także doskonalić swoje działanie. Dane wejściowe są zmiennymi niezależnymi i są one cechami badanych elementów. Informacja wyjściowa jest zmienną zależną i jest nią wynik klasyfikacji. Algorytm dokonujący klasyfikacji

nazywa się modelem predykcji, a sam proces — predykcją. W uczeniu maszynowym wyróżnia się dwa zbiory danych:

- Zbiór treningowy — składa się z danych wejściowych (zmiennych niezależnych) i danych wyjściowych (poprawnie przypisanych klas). Ten zbiór służy do trenowania modelu predykcji czyli do uczenia.
- Zbiór testowy — składa się z tych samych elementów co zbiór treningowy, natomiast ma inne zastosowanie. Służy do ewaluacji efektywności modelu predykcji. Zmienna zależna jest porównywana z wynikiem klasyfikacji, dzięki czemu możliwe jest obliczenie skuteczności modelu predykcji.




Rys. 2.1. Wykorzystanie danych w modelu predykcji

## 2.3. Rola metryk w inżynierii oprogramowania

Norma IEEE 1061-1998 [2] definiuje metrykę jako "funkcję odwzorowującą jednostkę oprogramowania w wartość liczbową. Ta wyliczona wartość jest interpretowalna jako stopień spełnienia pewnej własności jakości jednostki oprogramowania."

W inżynierii oprogramowania metryki są wykorzystywane we wszystkich fazach procesu wytwarzania oprogramowania. Pozwalają na porównywanie ze sobą różnych elementów lub różnych projektów ponieważ są danymi liczbowymi. W fazie projektowania mogą służyć m.in. do szacowania nakładu pracy potrzebnego do realizacji projektu. W fazie produkcji i testów do mierzenia jakości aplikacji, wydajności pracy czy złożoności programu.

Metryki można podzielić według różnych kryeriów. Ze względu na typ artefaktu jaki opisują dzieli się je na:

- Metryki produktu (inaczej metryki kodu źródłowego). Są bezpośrednio wyliczane z kodu źródłowego programu. Przykładem takich metryk są:
  - Zestaw metryk CK [7], do którego należą:

        - uśrednione metody na klasę (ang. *Weighted Methods per Class*, WMC),
        - głębokość drzewa dziedziczenia (ang. *Depth of Inheritance Tree*, DIT),
        - liczba dzieci (ang. *Number of Children*, NOC),
        - zależność między obiektami (ang. *Coupling Between Objects*, CBO),
        - odpowiedzialność danej klasy (ang. *Response For a Class*, RFC),
        - brak spójności metod (ang. *Lack of Cohesion of Methods*, LCOM).
    - OO — metryki obiektowe, np.:
        - liczba atrybutów (ang. *Number of attributes*, NOA),
        - liczba metod (ang. *Number of methods*, NOM),
        - liczba dziedziczonych metod (ang. *Number of methods inherited*, NOMI).
    - LOC — liczba linii kodu.
- Metryki procesu (inaczej metryki zmian). Określają zmienność atrybutu w czasie. Oblicza się je dla zadanych przedziałów czasowych. Niezbędna do ich obliczenia jest historia projektu, którą można uzyskać dzięki systemom kontroli wersji (jak SVN czy Git). Przykłady metryk procesu:
    - liczba modyfikacji (rewizji) pliku (ang. *Number of Revisions*, NR),
    - liczba autorów zmieniających plik (ang. *Number of Distinct Commiters*, NDC),
    - liczba zmienionych linii kodu (ang. *Number of Modified Lines*, NML),
    - wiek pliku (ang. *Age*, AGE),
    - liczba refaktoryzacji pliku (ang. *(Number of Refactorings*, NREF),
    - liczba dodanych (usuniętych, zmienionych) metod,
    - liczba dodanych (usuniętych, zmienionych) atrybutów.

Dodatkowo można podzielić metryki z uwagi na cel pomiaru [21]:

- metryki złożoności,
- metryki szacowania nakładu,
- metryki funkcjonalności.

Model predykcji defektów to narzędzie, które na podstawie wartości metryk danego projektu dokonuje wskazania defektów znajdujących się w tym projekcie. Aby poprawnie zinterpretować wskazania dostarczane przez model predykcji defektów należy określić czym jest defekt. Norma 982.2 IEEE/ANSI [1] definiuje defekt jako anomalię w produkcie, która może być:

- zaniechaniami i niedoskonałościami znalezionymi podczas wczesnych faz cyklu życia oraz
- błędami zawartymi w oprogramowaniu wystarczająco dojrzałym do testowania lub działania.

Istniejące badania wykazały, że metryki procesu przewyższyły metryki produktu w kontekście budowania modeli predykcji defektów [13,17,28,35]. Z tego powodu w dalszej części pracy zrezygnowano z wykorzystania metryk produktu, biorąc pod uwagę jedynie metryki procesu.

## 2.4. Koszty zapewnienia jakości

W inżynierii oprogramowania występuje kilka różnych definicji jakości. Na potrzeby niniejszej pracy przyjęto definicję Kana [29] "brak defektów w produkcie". Zarządzanie jakością oprogramowania polega na podejmowaniu działań mających na celu zapewnienie jakości tworzonego oprogramowania poprzez szereg testów, które wspierają cały proces rozwoju oprogramowania.

- Na etapie zbierania wymagań — weryfikacja czy określone wymagania będą możliwe do zweryfikowania (przetestowania).

- Na etapie projektowania — zaplanowanie procesu testowego, wybór środowisk testowych.
- Na etapie kodowania — definiowanie i realizacja scenariuszy i przypadków testowych oraz rejestracja defektów.
- Na etapie zamknięcia projektu — testy integracyjne, testy akceptacyjne, testy operacyjne.

Jak wykazano w [3] koszty zapewnienia jakości są prawie proporcjonalne do wielkości modułu. Dlatego badacze biorą pod uwagę wysiłek związany z działaniami mającymi na celu zapewnienie jakości [30, 37, 42]. Zmniejszenie wysiłku i kosztu związanego z zapewnieniem jakości to obecnie jeden z głównych kierunków badań [27].

Podstawowym celem pracy dyplomowej jest stworzenie narzędzi w postaci wtyczek do środowiska KNIME, służących do gromadzenia metryk oprogramowania z systemów kontroli wersji oraz zgromadzenie jak największej ilości metryk w publicznym repozytorium. Następnym krokiem jest stworzenie modelu (modeli) predykcji defektów oraz ich ewaluacja, biorąc pod uwagę wysiłek związany z zapewnieniem jakości oprogramowania. Stworzenie narzędzi pozwalających na zautomatyzowane gromadzenie metryk z dostępnych projektów (na przykład open source) pozwoli rozszerzyć publiczne zbiory danych. Dzięki temu będzie możliwe wykorzystanie tych danych do tworzenia modeli predykcji defektów oprogramowania dzięki: większym zbiorom uczącym; ewaluacji modeli na większych zbiorach danych.

## 2.5. Systemy kontroli wersji jako źródło danych o projektach

Jak wspomniano wcześniej w rozdziale 2.3 aby obliczyć metryki procesu, konieczne jest uzyskanie historii projektu. Przegląd literatury pozwolił na wyodrębnienie sposobów i narzędzi, które pozwalają na porównywanie różnych wersji kodu źródłowego. Jak wykazano w [28, 40, 41] predykcja na poziomie plików jest bardziej efektywna niż na poziomie pakietów. Idąc dalej w kierunku uszczegółowienia wyników predykcji, można przypuszczać, że predykcja na poziomie metod byłaby skuteczniejsza niż na poziomie plików. Badanie [27] wykazało, że pliki zawierające błędy zawierają prawie lub ponad 10 metod, natomiast tylko kilka metod zawiera błędy (mediana 1–2). Jest to nie tylko odpowiedź na pytanie czy predykcja na poziomie metod jest skuteczniejsza, ale również wskazanie przyczyny takiego stanu rzeczy. Jednakże aby w pełni wykorzystać możliwości ograniczenia kosztów jakości poprzez predykcję na poziomie metod, potrzebne są skuteczne modele, dostarczające wiarygodnych wyników.

Ze względu na powyższe zależności, podjęto decyzję o prowadzeniu dalszych prac w kierunku budowy modeli predykcji na poziomie metod. Poniżej wypisano techniki porównywania kodu źródłowego na poziomie metod.

- ChangeDistiller [15] — polega na odwzorowaniu kodu źródłowego Java w strukturze drzewiastej, jaką jest AST (ang. *Abstract Syntax Tree*) a następnie wyodrębnieniu zmian pomiędzy dwiema wersjami przy użyciu algorytmów porównywania drzew.
- Historage [26] — wykorzystuje system kontroli wersji Git do przechowywania zidentyfikowanych zmian w kodzie na niskim poziomie.
- APFEL [45] — jest wtyczką do środowiska Eclipse, która zbiera w bazie danych niskopoziomowe zmiany w kodzie. Działa z systemem kontroli wersji CVS i źródłami Java.
- C-REX [23] — wyodrębnia fakty z historii kodu źródłowego języka C, a następnie porównuje ze sobą kolejne wersje.

- Kenyon [5].
- Beagle [19].

### 2.5.1. Historia błędów w projektach

Metryki, które stanowią dane wejściowe w modelach predykcji są zmiennymi niezależnymi (ang. *independent variables*). Pełny zestaw danych potrzebny do wytrenowania modelu obejmuje także zmienne zależne (ang. *dependent variables*). Zmienna niezależna reprezentuje wyjście (wynik), oraz może być używana do testowania modelu, żeby ocenić jego skuteczność. W predykcji defektów oprogramowania zmienną zależną jest liczba błędów lub zmienna określająca czy występuje błąd.

Aby uzyskać informacje o błędach w projekcie stosuje się metody linkowania błędów. Linkowanie polega na odszukaniu powiązań pomiędzy zmianą zapisaną w repozytorium kodu, a błędem zgłoszonym w systemie śledzenia zmian (ang. *Issue Tracking System*, ITS), takim jak JIRA, Bugzilla, IBM Rational ClearQuest czy innym.

Metoda używana w tej pracy opiera się na metodzie SZZ [44]. Jej podstawową zaletą jest porównywanie czasu naprawienia błędu zapisanego w ITS z czasem wysłania poprawki do systemu kontroli wersji. Dzięki takiemu porównaniu wyklucza się dużą liczbę błędnych wskazań, które wynikają z niewłaściwego lub przypadkowego przyporządkowania błędu do zmiany kodu. Przyczyną takich błędnych dopasowań może być umieszczenie w opisie zmiany ciągu numerycznego niebędącego numerem błędu.

# Rozdział 3

# Omówienie infrastruktury pomiarowej

Badania zgodnie z założeniami przeprowadzono w środowisku KNIME rozszerzonym o wtyczki z DePress. Dodatkowo w KNIME użyto algorytmów klasyfikacji z pakietu Weka.

## 3.1. KNIME

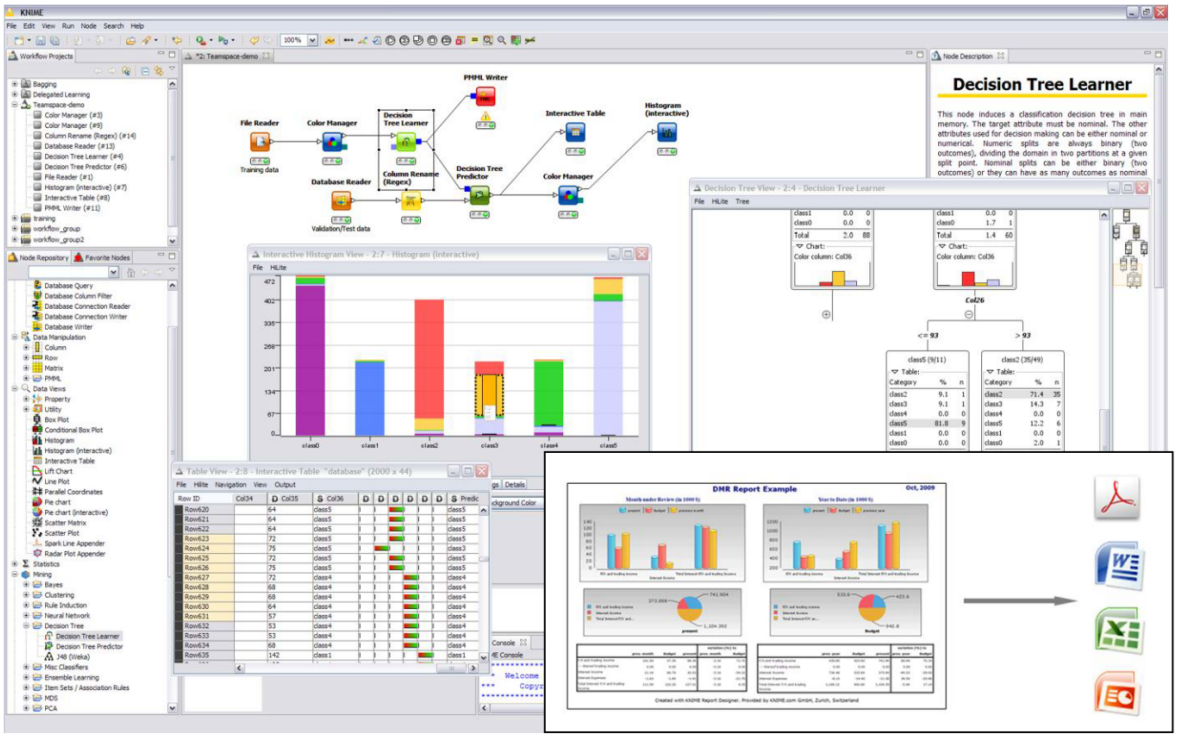
KNIME jest platformą do analizy danych, która umożliwia wykonywanie zaawansowanych statystyk i eksploracji danych, w celu analizy trendów i przewidywania wyników. Wizualny zestaw narzędzi pozwala na pozyskiwani danych, przekształcanie ich, wstępne rozpoznanie, analizy predykcyjne i wizualizację. KNIME daje również możliwość tworzenia raportów na podstawie zgromadzonych informacji. KNIME Desktop jest oprogramowaniem o otwartym kodzie (ang. *open source*) udostępnianym na licencji GPL [4].

Platforma KNIME zawiera setki węzłów pozyskiwania danych, przetwarzania i filtrowania, analizy i eksploracji danych, wizualizacji i innych. Oprogramowanie bazuje na platformie Eclipse[1] i jest rozszerzalne poprzez system wtyczek, dzięki czemu znajduje zastosowanie w komercyjnych środowiskach produkcyjnych jak i środowiskach badawczych. Przykład procesu analizy i eksploracji danych przedstawiono na rysunku 3.2.

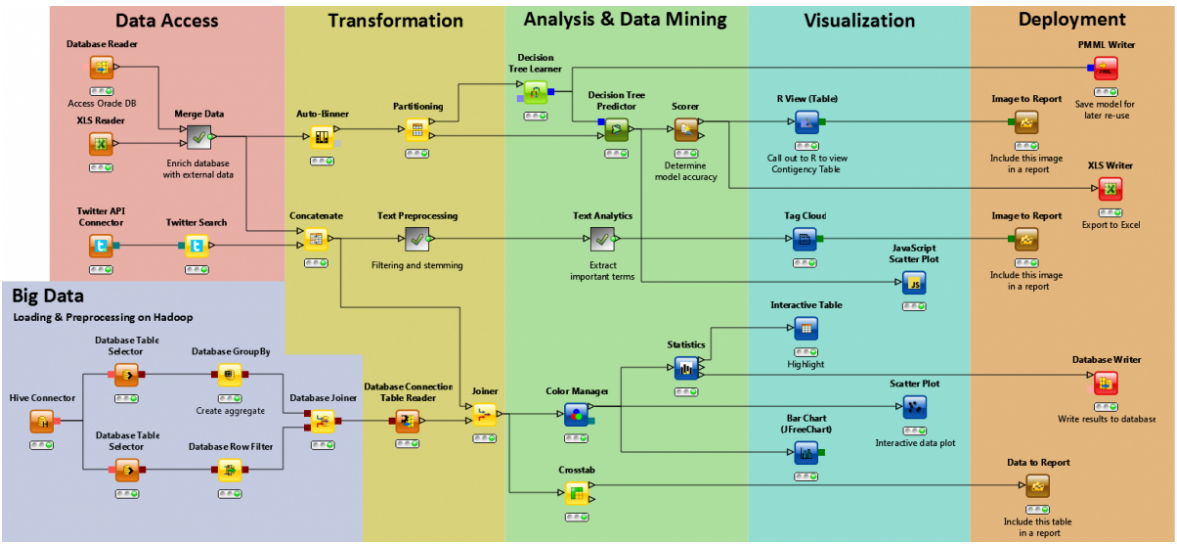
**Budowanie procesu.** Proces jest tworzony z węzłów dostępnych w programie. Węzły umieszcza się w edytorze i łączy ze sobą. Są one podstawowymi jednostkami w procesie. Każdy węzeł może mieć porty wejściowe i wyjściowe. Dane są przekazywane pomiędzy węzłami z portu wyjściowego jednego węzła do portu wejściowego innego węzła. Wyjście węzła mogą stanowić na przykład dane tabelaryczne, obraz, model zapisany w PMML (ang. *Predictive Model Markup Language*) i inne.

Węzeł przed użyciem musi zostać skonfigurowany. Każdy rodzaj węzła posiada swój zestaw parametrów konfigurowalnych za pomocą interfejsu graficznego. Prawidłowo skonfigurowany węzeł może utowrzyć na wyjściu tabelę danych, kótra może być oglądana w wewnętrznym edytorze. Oprócz tabeli danych niektóre węzły udostępniają widoki. Widokiem może być np. wykres, graf, tabela, itd. Widoki mogą zawierać elementy interaktywne pozwalające na dostosowywanie prezentowanych danych, zmianę układu, kolorów, typu wykresu, itp.

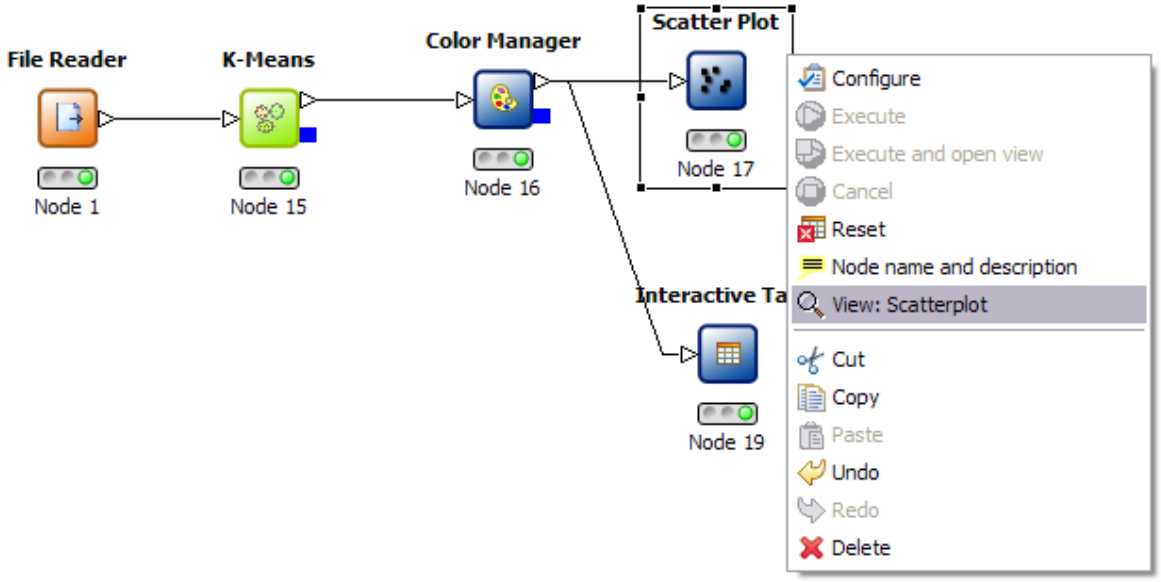
1. http://eclipse.org




Rys. 3.1. Interejs KNIME. Źródło [4]




Rys. 3.2. Proces zbudowany w KNIME. Źródło [4]




Rys. 3.3. Węzły KNIME. Źródło [4]

## 3.2. R

R jest darmowym środowiskiem i językiem do obliczeń statystycznych i wizualizacji. Jest podobny do języka i środowiska S. Ma szerokie zastosowanie na świecie, jest podstawowym narzędziem w bioinformatyce, używa się go w takich firmach jak Facebook, Google, Form, Mozilla czy Twitter. Wiele pakietów statystycznych (np. RapidMiner, KNIME) oferuje mechanizmy zapewniające współpracę z R.

R jest zbudowany w architekturze modułowej. Jest rozszerzalny za pomocą pakietów zawierających dodatkowe funkcje lub dodatkowe narzędzia przeznaczone dla poszczególnych dziedzin nauki. Zaletą R jest także możliwość tworzenia wysokiej jakości grafik gotowych do publikacji. Tworzenie grafik jest uproszczone poprzez stosowanie gotowych szablonów, z jednoczesnym zachowaniem możliwości pełnej kontroli i personalizacji przez użytkownika. Ma także swój własny format dokumentacji podobny do LaTeX.

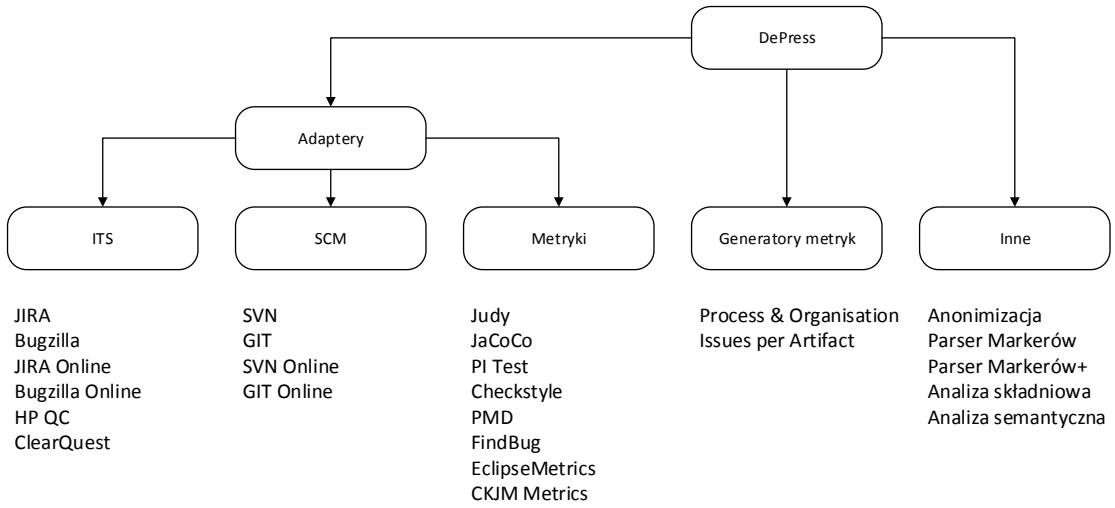
## 3.3. DePress

DePress jest oparty wyłącznie na architekturze wtyczek KNIME, dzięki czemu możliwa jest integracja i współdziałanie z istniejącymi wtyczkami. Głównym zadaniem pakietu jest predykcja defektów oprogramowania. Poszczególne wtyczki dostarczają odpowiednich funkcji w zależności od ich przeznaczenia. Podzielono je na trzy główne grupy (rysunek 3.4):

- Adaptery — pozwalają na pobieranie danych z zewnętrznych narzędzi.
- Generatory metryk — wyliczają metryki na podstawie danych wejściowych.
- Inne — dostarczają metod do dodatkowych przekształceń danych.

Dzięki separacji wtyczek jest możliwa łatwa zamiana jednej wtyczki na inną. Na przykład gdy analizując projekt okaże się, że został on przeniesiony z SVN do Git, wystarczy użyć odpowiedniej wtyczki adaptera aby na nowo pobrać dane o projekcie.

W badaniu użyto wtyczek adapterów Jira (Online) i Bugzilla (Online) do pobierania danych o zgłoszonych błędach z systemów śledzenia zagadnień (ang. *Issue Tracking*




Rys. 3.4. Struktura DePress. Źródło [34]

*System*, ITS). Istniejące wtyczki adapterów do współpracy z systemami kontroli wersji (SCM) okazały się nie wystarczające, ponieważ dostarczają tylko danych o zmianach na poziomie klas. Potrzebne było stworzenie nowej wtyczki AstMetrics do ekstrakcji zmian na niskim poziomie granulacji.

## 3.4. AstMetrics

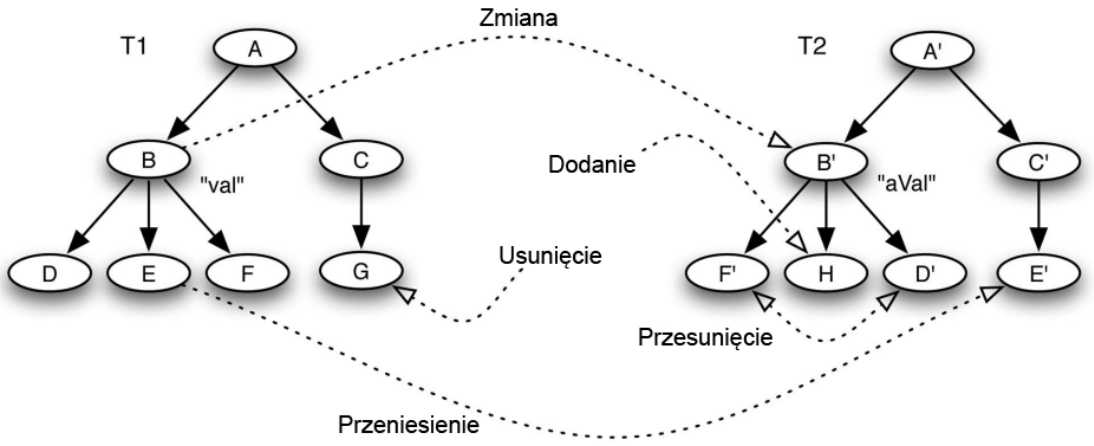
Wtyczka AstMetrics w pierwszej wersji została napisana przez Piotra Mitkę [38]. Na potrzeby tego badania została w dużej części napisana od nowa i poszerzona o nowe metryki. Bazując na oprogramowaniu Change Distiller [15] dokonuje ona porównania historycznych wersji kodu źródłowego każdego pliku pobranego z repozytorium. Każda wersja pliku jest odwzorowywana za pomocą drzewa AST (ang. *Abstract Syntax Tree*) a następnie drzewa są porównywane pomiędzy kolejnymi wersjami (patrz rysunek 3.5). Na tej podstawie jest możliwe wykrycie zmian w kodzie źródłowym na niskim poziomie granulacji. Zmiany są zapisywane w tymczasowej bazie danych a następnie są wyliczane z nich metryki dla każdej metody z każdej klasy znajdującej się w kodzie źródłowym.

Metryki wyliczane w pierwszej wersji wtyczki zostały zawarte w tabeli 3.1 a dodane w drugiej wersji wtyczki w tabeli 3.2.

Dodatkowo zaimplementowano drugie wyjście wtyczki z wszystkimi historiami metod, tj. z informacjami o tym która metoda została zmieniona w danej wersji historycznej w repozytorium kodu. W związku z tym, że do repozytorium są zapisywane zmiany w pojedycznych plikach, a nie bardziej szczegółowe, wystąpiła potrzeba stworzenia takiego zestawienia. Struktura tworzonej tabeli danych została przedstawiona w tabeli 3.3. Dane historyczne służą do wyszukiwania defektów w kodzie źródłowym, które zostało opisane w rozdziale 3.4.2.

Do prawidłowego działania wtyczki wymagane jest określenie wartości parametrów:

- dirname - katalog ze sklonowanym repozytorium git (np. *C:/ant* lub */home/user/ant*,
- package - pakiet, który zostanie uwzględniony w tabelach wyjściowych (np. *org.apache.ant*),
- exclude_dir - katalog, który zostanie wyłączony z pobieranai danych (np. *src/tests*),
- bottom_commit - identyfikator rewizji początkowej, dozwolone jest podanie iden-




Rys. 3.5. Porównanie drzewa AST. Źródło [15]

**Tabela** 3.1. Metryki wyliczane w pierwszej wersji wtyczki AstMetrics

| **Nazwa metryki** | **Opis** |
|---|---|
| allMethodHistories | ile razy metoda była zmieniana w historii |
| methodHistories | ile razy metoda była zmieniana w danych przedziale czasowym |
| authors | liczba autorów |
| stmtAdded | łączna liczba dodanych instrukcji |
| maxStmtAdded | maksymalna liczba dodanych instrukcji |
| avgStmtAdded | średnia liczba dodanych instrukcji |
| stmtUpdated | łączna liczba zmienionych instrukcji |
| maxsSmtUpdated | maksymalna liczba zmienionych instrukcji |
| avgStmtUpdated | średnia liczba zmienionych instrukcji |
| stmtDeleted | łączna liczba usuniętych instrukcji |
| maxStmtDeleted | maksymalna liczba usuniętych instrukcji |
| avgStmtDeleted | średnia liczba usuniętych instrukcji |
| stmtParentChanged | łączna liczba zmienionych instrukcji nadrzędnych |
| churn | suma stmtAdded i stmtDeleted we wszystkich wersjach |
| maxChurn | maksymalna wartość churn na przestrzeni wersji |
| avgChurn | średnia wartość churn na przestrzeni wersji |
| decl | łączna liczba zmian deklaracji metody |
| cond | łączna liczba zmian deklaracji warunkowych |
| elseAdded | łączna liczba dodanych części *else* |
| elseDeleted | łączna liczba usuniętych części *else* |

**Tabela** 3.2. Metryki dodane w drugiej wersji wtyczki AstMetrics

| **Nazwa metryki** | **Opis** |
|---|---|
| loopsAdded | łączna liczba dodanych pętli |
| loopsUpdated | łączna liczba zmienionych pętli |
| loopsDeleted | łączna liczba usuniętych pętli |
| variablesAdded | łączna liczba dodanych instrukcji deklaracji zmiennej |
| variablesUpdated | łączna liczba zmienionych instrukcji deklaracji zmiennej |
| variablesDeleted | łączna liczba usuniętych instrukcji deklaracji zmiennej |
| assigmentsAdded | łączna liczba dodanych instrukcji przypisania |
| assigmentsUpdated | łączna liczba zmienionych instrukcji przypisania |
| assigmentsDeleted | łączna liczba usuniętych instrukcji przypisania |
| returnsAdded | łączna liczba dodanych instrukcji *return* |
| returnsUpdated | łączna liczba zmienionych instrukcji *return* |
| returnsDeleted | łączna liczba usuniętych instrukcji *return* |
| nullsAdded | łączna liczba dodanych instrukcji zawierających *null* |
| nullsUpdated | łączna liczba zmienionych instrukcji zawierających *null* |
| nullsDeleted | łączna liczba usuniętych instrukcji zawierających *null* |
| casesAdded | łączna liczba dodanych instrukcji warunkowych *case* |
| casesUpdated | łączna liczba zmienionych instrukcji warunkowych *case* |
| casesDeleted | łączna liczba usuniętych instrukcji warunkowych *case* |
| breaksAdded | łączna liczba dodanych instrukcji *break* |
| breaksUpdated | łączna liczba zmienionych instrukcji *break* |
| breaksDeleted | łączna liczba usuniętych instrukcji *break* |
| objectsAdded | łączna liczba dodanych instrukcji tworzących obiekt |
| objectsUpdated | łączna liczba zmienionych instrukcji tworzących obiekt |
| objectsDeleted | łączna liczba usuniętych instrukcji tworzących obiekt |
| catchesAdded | łączna liczba dodanych bloków przechwytywania wyjątku |
| catchesUpdated | łączna liczba zmienionych bloków przechwytywania wyjątku |
| catchesDeleted | łączna liczba usuniętych bloków przechwytywania wyjątku |
| throwsAdded | łączna liczba dodanych bloków rzucania wyjątku |
| throwsUpdated | łączna liczba zmienionych bloków rzucania wyjątku |
| throwsDeleted | łączna liczba usuniętych bloków rzucania wyjątku |

**Tabela** 3.3. Struktura tabeli wyjściowej z danymi historycznymi wtyczki AstMetrics

| **Nazwa kolumny** | **Opis** |
|---|---|
| MethodName | Nazwa metody |
| Author | Autor zmiany |
| Message | Opis zmiany |
| Date | Data przesłania |
| CommitID | Unikalny identyfikator zmiany |

tyfikatora SHA-1, tagu poprzedzonego napisem *tag:* lub napis *initial* oznaczającego pierwszą (najstarszą) rewizję w repozytorium,

- top_commit - identyfikator rewizji końcowej, czyli rewizji dla której zostaną obliczone metryki, dozwolone jest podanie identyfikatora SHA-1 lub tagu poprzedzonego napisem *tag:*,
- top_commit_post_rel - identyfikator rewizji zamykającej, pomiędzy tą wersją a wersją *top_commit* zostanie pobrana historia repozytorium, dozwolone jest podanie identyfikatora SHA-1, tagu poprzedzonego napisem *tag:* lub napis *current* oznaczającego ostatnią (najnowszą) rewizję w repozytorium.

Działanie wtyczki obrazuje rysunek 3.6.

### 3.4.1. Gromadzenie metryk

Do trenowania i oceny algorytmów klasyfikacji często stosuje się walidację krzyżową, polegającą na podziale zbioru danych na dwa podzbiory — treningowy i testowy — a następnie wykorzystaniu ich zgodnie z przeznaczeniem. W niniejszej pracy zastosowano inne podejście, polegające na wykorzystaniu dwóch niezależnych zbiorów danych z dwóch różnych wydań oprogramowania. Metryki wersji A posłużyły jako zbiór treningowy a metryki wersji B jako zbiór testowy. Zaletą takiego podejścia jest lepsze dopasowanie do rzeczywistych warunków występujących w procesie wytwarzania oprogramowania. W tabeli 3.4 zawarto informacje o przebadanym oprogramowaniu wraz z oznaczeniami wersji A i B. Metryki zostały wyliczone z użyciem wtyczki AstMetrics. Wprowadzono również dodatkowoą metrykę pomocniczą LOC (ang. *ang. Lines of code*) - liczba linii kodu.

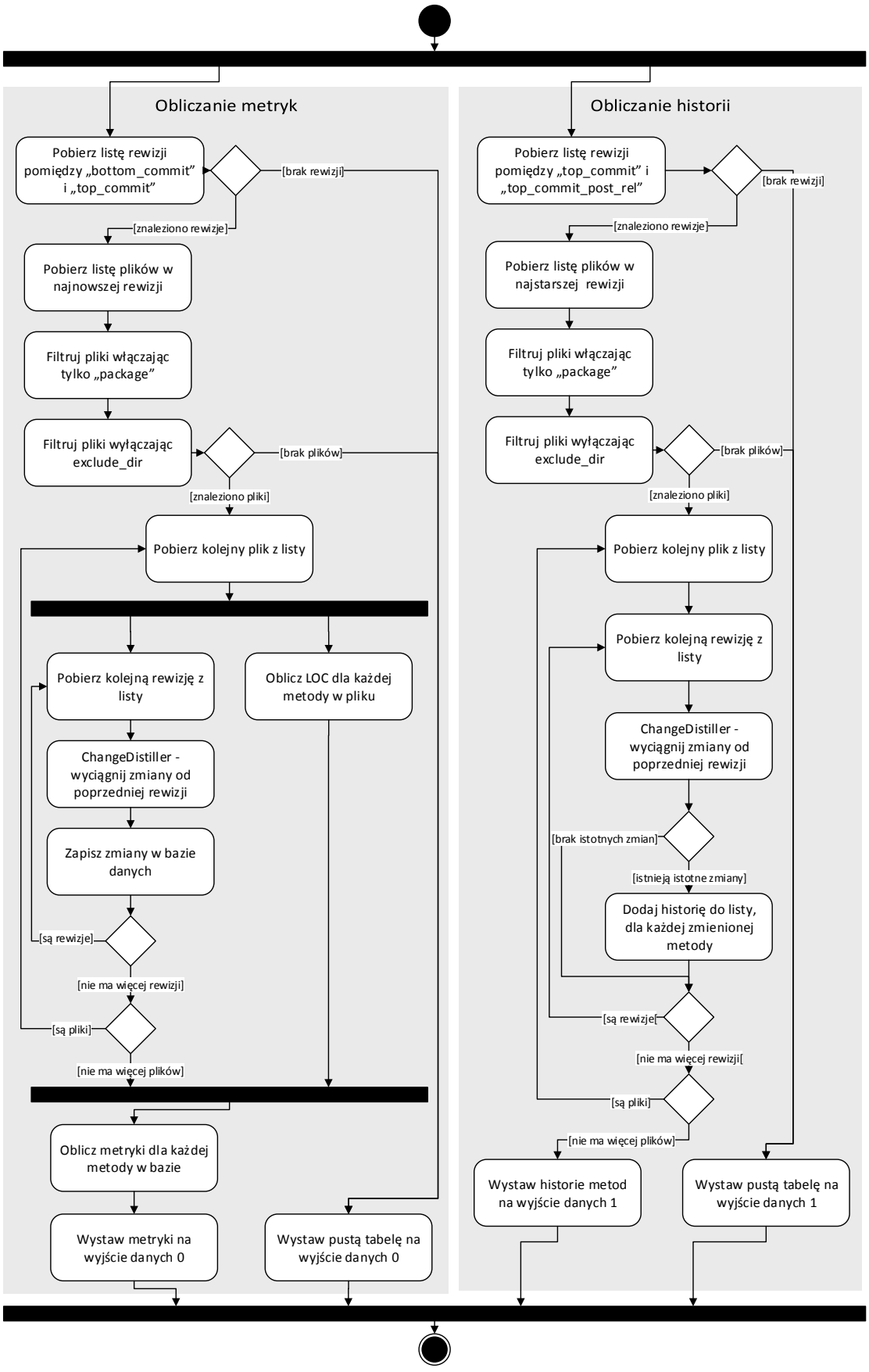
### 3.4.2. Gromadzenie danych o defektach

Pełna informacja wejściowa do trenowania i testowania algorytmu klasyfikacji musi zawierać jeszcze dane o błędach. Aby uzyskać te dane zastosowano metodę linkowania błędów według poniższego algorytmu:

1. Wylistowanie wszystkich metod wchodzących w skład wydanej wersji oprogramowania.
2. Pobranie historii metod za pomocą AstMetrics od daty wydania wersji do daty końcowej projektu.
3. Znalezienie rewizji naprawiających błąd.
4. Porównanie daty naprawienia błędu z datą ustawioną w systemie śledzenia zagadnień i wyeliminowanie niezgodnych wpisów.
5. Zliczenie unikalnych numerów błędu dla danej metody i zapisanie wartości jako liczba błędów.
6. Oznaczenie metod, które zostały zmienione w commitach naprawiających błąd jako metod zawierających błąd (*1*).
7. Oznaczenie pozostałych metod jako nie zawierających błędu (*0*).

Proces został zaprojektowany w środowisku KNIME, w dwóch wersjach: z systemem śledzenia zagadnień Bugzilla i JIRA. Proces był uruchamiany automatycznie z poziomu środowiska R, dzięki wykorzystaniu trybu wsadowego KNIME (ang. *batch mode*). Parametry konfiguracyjne były przekazywane za pomocą zmiennych (ang. *workflow variables*). Współdziałanie środowiska KNIME i R zostało zobrazowane na rysunku 3.7.

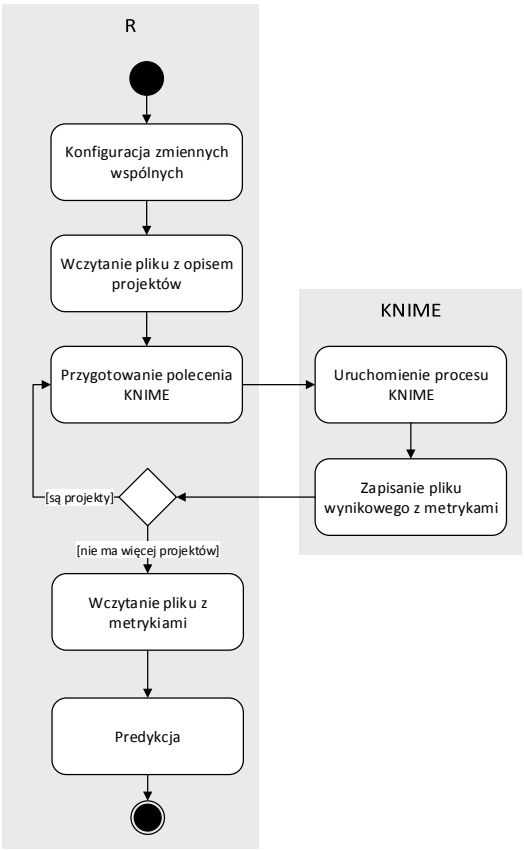
W tabeli 3.5 zebrano dane dotyczące wielkości projektów oraz liczby błędów. Dane dotyczą wersji B każdego projektu.




Rys. 3.6. Diagram aktywności wtyczki AstMetrics

**Tabela** 3.4. Przegląd projektów wykorzystanych do badania

| **Nazwa projektu** | **Wersja A** | **Wersja B** | **ITS** |
|---|---|---|---|
| ECF | Root_Release_3_0 | Root_Release_3_1 | Bugzilla |
| Ant | ANT_170 | ANT_180 | Bugzilla |
| JMeter | v2_8 | v2_9 | Bugzilla |
| Commons Lang | LANG_3_0 | LANG_3_1 | JIRA |
| AspectJ | V1_5_0_final | V1_6_0 | Bugzilla |
| CXF | cxf-2.7.0 | cxf-3.0.0 | JIRA |
| Isis | isis-1.7.0 | isis-1.8.0 | JIRA |
| Commons Email | EMAIL_1_2 | EMAIL_1_3 | JIRA |
| Commons JXPath | JXPATH_1_1 | JXPATH_1_2 | JIRA |
| Commons Logging | LOGGING_1_0 | LOGGING_1_1_0 | JIRA |
| Commons Net | NET_2_0 | NET_2_2 | JIRA |
| Accumulo | 1.5.2 | 1.6.0 | JIRA |
| Karaf | karaf-2.3.0 | karaf-2.4.0 | JIRA |
| ActiveMQ | activemq-5.8.0 | activemq-5.9.0 | JIRA |
| Hive | release-1.0.0 | release-1.1.0 | JIRA |
| James Mailbox | apache-james-mailbox-0.3 | apache-james-mailbox-0.4 | JIRA |
| Jackrabbit FileVault | jackrabbit-filevault-3.0.0 | jackrabbit-filevault-3.1.0 | JIRA |
| OpenJPA | 1.0.0 | 2.0.0 | JIRA |
| Jackrabbit Oak | jackrabbit-oak-1.0.0 | jackrabbit-oak-1.1.0 | JIRA |
| TomEE | tomee-1.5.0 | tomee-1.6.0 | JIRA |
| Tika | 1.5 | 1.6 | JIRA |
| Lucene - Core | lucene_solr_4_9_0 | lucene_solr_4_10_0 | JIRA |
| Solr | lucene_solr_4_9_0 | lucene_solr_4_10_0 | JIRA |
| Mahout | mahout-0.6 | mahout-0.7 | JIRA |
| Apache Gora | apache-gora-0.3 | apache-gora-0.4 | JIRA |
| Flume | release-1.4.0 | release-1.5.0 | JIRA |
| Nutch | release-2.1 | release-2.2 | JIRA |
| Apache Knox | v0.4.0-release | v05.0-release | JIRA |
| Phoenix | v4.2.0 | v4.3.0 | JIRA |




Rys. 3.7. Przebieg badania w środowisku R i KNIME

**Tabela** 3.5. Statystyki projektów

| **Nazwa projektu** | **LOC** | **Ilość metod** | **Ilość błędów** | **Błędnych metod** | **Udział błędnych metod** |
|---|---|---|---|---|---|
| ECF | 54 264 | 9 306 | 290 | 236 | 2,54% |
| Ant | 60 715 | 9 120 | 480 | 407 | 4,46% |
| JMeter | 50 181 | 7 592 | 447 | 365 | 4,81% |
| Commons Lang | 15 802 | 2 555 | 37 | 34 | 1,33% |
| AspectJ | 120 367 | 18 090 | 595 | 435 | 2,40% |
| CXF | 313 115 | 40 141 | 1 225 | 759 | 1,89% |
| Isis | 111 699 | 25 449 | 19 | 18 | 0,07% |
| Commons Email | 1 202 | 210 | 13 | 9 | 4,29% |
| Commons JXPath | 12 652 | 1 531 | 35 | 35 | 2,29% |
| Commons Logging | 1 638 | 243 | 6 | 6 | 2,47% |
| Commons Net | 5 536 | 1 169 | 94 | 78 | 6,67% |
| Accumulo | 38 483 | 3 816 | 231 | 153 | 4,01% |
| Karaf | 192 685 | 28 565 | 661 | 544 | 1,90% |
| ActiveMQ | 66 861 | 10 582 | 116 | 102 | 0,96% |
| Hive | 104 202 | 15 254 | 381 | 296 | 1,94% |
| James Mailbox | 14 258 | 2 130 | 20 | 20 | 0,94% |
| Jackrabbit FileVault | 25 861 | 3 344 | 55 | 34 | 1,02% |
| OpenJPA | 174 748 | 36 954 | 1 378 | 944 | 2,55% |
| Jackrabbit Oak | 116 458 | 14 971 | 983 | 731 | 4,88% |
| TomEE | 9 535 | 1 091 | 63 | 56 | 5,13% |
| Tika | 31 166 | 3 797 | 226 | 212 | 5,58% |
| Lucene - Core | 238 454 | 27 438 | 650 | 493 | 1,80% |
| Solr | 145 004 | 14 889 | 161 | 131 | 0,88% |
| Mahout | 60 917 | 7 812 | 370 | 336 | 4,30% |
| Apache Gora | 11 621 | 2 236 | 11 | 11 | 0,49% |
| Flume | 40 956 | 4 791 | 115 | 107 | 2,23% |
| Nutch | 15 013 | 1 776 | 122 | 89 | 5,01% |
| Apache Knox | 25 857 | 3 359 | 23 | 22 | 0,65% |
| Phoenix | 102 397 | 12 909 | 345 | 246 | 1,91% |

### 3.4.3. Omówienie zebranych danych pomiarowych

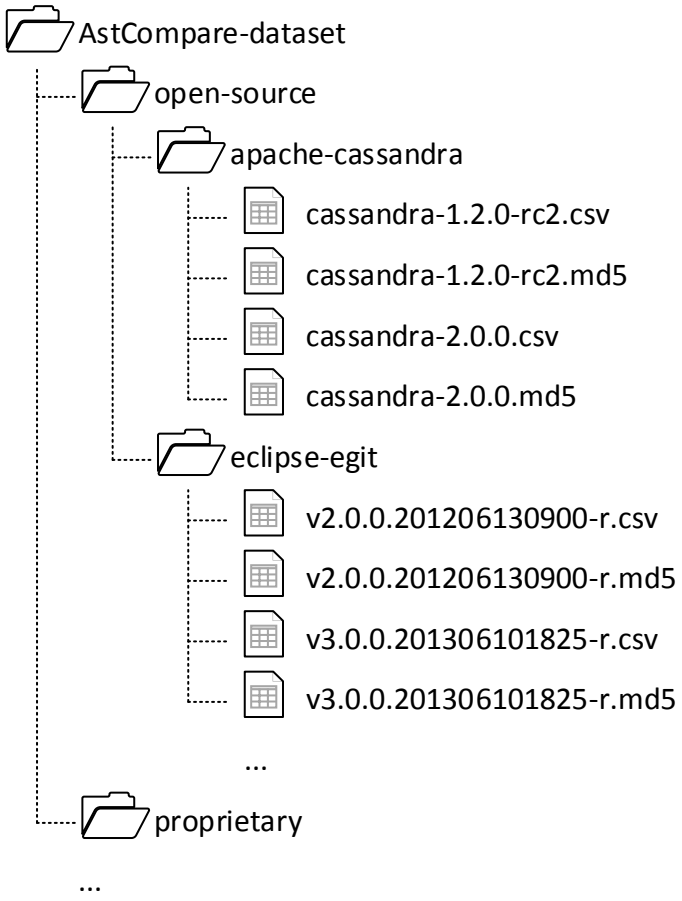
Zebrane dane zawierają 54 kolumny. Pierwsza zawiera sygnaturę metody, kolejne 50 to metryki wymienione w rozdziale 3.4. Są to wartości liczbowe typu zmiennoprzecinkowego. Kolejne kolumny to: *loc* zawierająca miarę wielkości metody (wartości liczbowe całkowite), *NumOfBugs* zawierająca liczbę błędów (wartości liczbowe całkowite), *Buggy* zawierająca informacje o tym czy metoda zawiera błąd czy nie (wartości *1* lub *0*). Żadna z kolumn nie zawiera nieokreślonych lub brakujących wartości.

Zebrano dane z 29 projektów o otwartych źródłach, w dwóch wersjach każdego projektu.

**Wymagania dla repozytorium danych.** Dane powinny być zgromadzone w miejscu, które umożliwia łatwy dostęp poprzez protokół HTTP, bez uwierzytelniania lub logowania. Jest to istotne z punktu widzenia wykorzystania narzędzi do analizy i przetwarzania tych danych. Pliki danych powinny być oznaczone sumami kontrolnymi aby było możliwe sprawdzenie poprawności ich pobierania.

**Struktura danych i zawartość zbioru.** Zbiór metryk został zaprojektowany w celu umożliwienia przechowywania danych z projektów różnego typu. Na najwyższym poziomie został podzielony na 2 grupy: projekty o otwartym kodzie i projekty własnościowe (odpowiednio katalogi *open source* i *proprietary*). Katalog na kolejny poziomie zagnieżdżenia określa nazwę oprogramowania, a wewnątrz znajdują się arkusze danych w postaci plików CSV. CSV (ang. *Comma-separated values*) jest to plik tekstowy rozdzielany przecinkami. Pliki z danymi są nazywane numerem/ nazwą wydania. W przypadku repozytoriów Git zazwyczaj jest to nazwa tagu. W pierwszej linii każdego pliku znajdują się nagłówki kolumn. W tym samym katalogu co plik CSV powinien znajdować się plik z sumą kontrolną obliczoną funkcją skrótu MD5. Plik z sumą kontrolną powinien mieć taką samą nazwę jak plik z danymi, z rozszerzeniem *.md5*.

Na rysunku 3.8 przedstawiono ogólną strukturę zbioru danych wraz z kilkoma przykładowymi projektami.

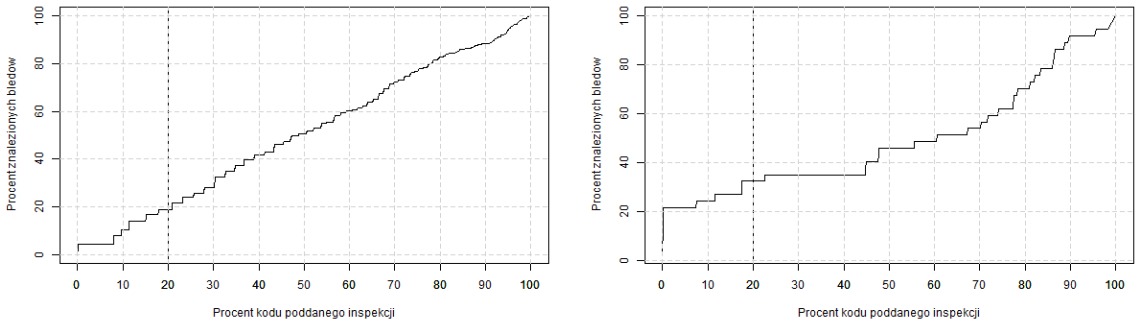



Rys. 3.8. Struktura repozytorium metryk AstMetrics

# Rozdział 4

# Modele predykcji i ich ewaluacja

**LOC.** Przyjęto, że podstawową metodą wyszukiwania błędów, stanowiącą punkt odniesienia, było przeglądanie kodu źródłowego od najmniejszej metody do największej. Zatem sortując dane wejściowe według kolumny *LOC* niemalejąco można obliczyć a następnie wykreślić skuteczność takiego przeglądu. Wykresy na rysunku 4.1 pokazują, że jest to w przybliżeniu zależność liniowa, co udowadnia niską efektywność takich przeglądów. Taka metoda przeglądu będzie dalej nazywana *LOC*.




Rys. 4.1. Skuteczność przeglądu LOC dla projektów Ant i Commons Lang

**Random Forest.** W licznych badaniach, m.in. [27, 28, 31, 33, 36] wykazano wysoką skuteczność metody Random Forest w predykcji defektów. Ponadto udział metod z błędami jest niewielki, w związku z czym modele predykcji mają tendencję do klasyfikowania wszystkich elementów do klasy *0*. Zauważono takie zachowanie podczas stosowania regresji logistycznej w [27]. Problem ten nie dotyka tej metody. Metoda Random Forest polega na tworzeniu dużej liczby drzew decyzyjnych a następnie przypisaniu klasy na podstawie dominanty (najczęstszej wartości) spośród wszystkich wyników częściowych.

W tabeli 4.1 zawarto wyniki predykcji z użyciem metody LOC oraz Random Forest. Wyniki powyżej 60% zaznaczono pogrubioną czcionką. Uzyskane rezultaty potwierdzają wysoką skuteczność algorytmu Random Forest. W kolejnej tabeli 4.2 umieszczono wyniki pozostałych miar skuteczności dla algorytmu Random Forest.

Metoda podstawowa (LOC) cechuje się efektywnością na poziomie średnim 27,4% ($\sigma = 10,4$). Wynik uzyskany dla RandomForest wynosi średnio 54,4% ($\sigma = 9,8$).

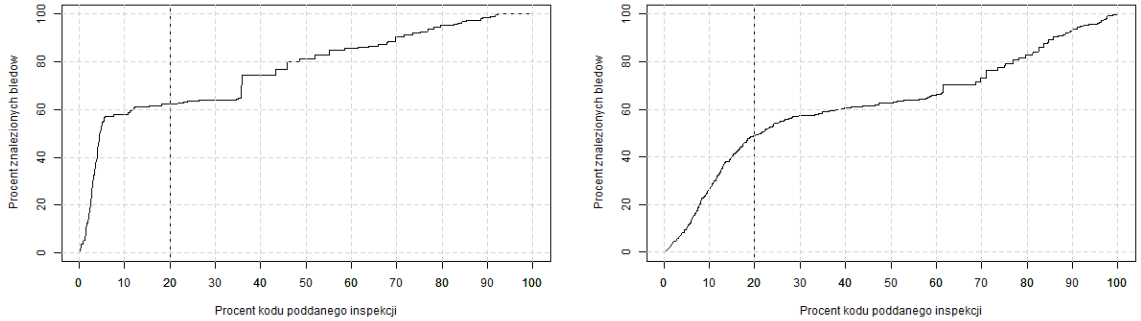
**Tabela** 4.1. Efektywność predykcji defektów pod względem wysiłku. Procent znalezionych błędów w 20% kodu.

| **Nazwa projektu** | **LOC** | **RandomForest** |
|---|---|---|
| ECF | 30 | **63** |
| Ant | 19 | 49 |
| JMeter | 25 | **68** |
| Commons Lang | 32 | 49 |
| AspectJ | 18 | 59 |
| CXF | 43 | 58 |
| Isis | 26 | 32 |
| Commons Email | 15 | 54 |
| Commons JXPath | 34 | **60** |
| Commons Logging | 0 | 50 |
| Commons Net | 28 | 40 |
| Accumulo | 32 | **68** |
| Karaf | 24 | 43 |
| ActiveMQ | 19 | 55 |
| Hive | 40 | 54 |
| James Mailbox | 35 | 50 |
| Jackrabbit FileVault | 38 | **64** |
| OpenJPA | 34 | 50 |
| Jackrabbit Oak | 34 | **66** |
| TomEE | 19 | 41 |
| Tika | 38 | 46 |
| Lucene - Core | 44 | 56 |
| Solr | 43 | 50 |
| Mahout | 30 | 52 |
| Apache Gora | 9 | **64** |
| Flume | 18 | 46 |
| Nutch | 25 | **60** |
| Apache Knox | 17 | 52 |
| Phoenix | 24 | **69** |

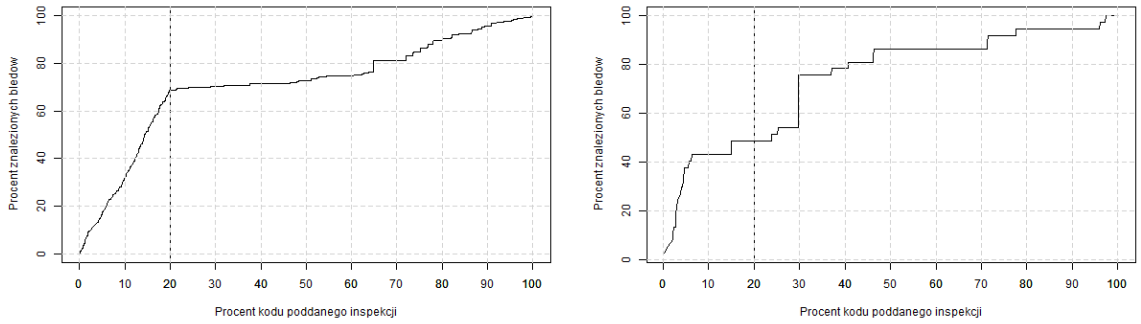
**Tabela** 4.2. Skuteczność predykcji defektów z wykorzystaniem algorytmu Random Forest

| **Nazwa projektu** | $A$ | $\kappa$ | $AUC$ |
|---|---|---|---|
| ECF | 0.985 | 0.594 | 0.760 |
| Ant | 0.964 | 0.473 | 0.758 |
| JMeter | 0.971 | 0.639 | 0.815 |
| Commons Lang | 0.989 | 0.444 | 0.737 |
| AspectJ | 0.979 | 0.399 | 0.776 |
| CXF | 0.975 | 0.255 | 0.795 |
| Isis | 0.999 | 0.105 | 0.547 |
| Commons Email | 0.957 | 0.000 | 0.697 |
| Commons JXPath | 0.988 | 0.622 | 0.790 |
| Commons Logging | 0.971 | 0.447 | 0.767 |
| Commons Net | 0.941 | 0.320 | 0.757 |
| Accumulo | 0.960 | 0.466 | 0.829 |
| Karaf | 0.982 | 0.094 | 0.631 |
| ActiveMQ | 0.994 | 0.580 | 0.747 |
| Hive | 0.982 | 0.130 | 0.691 |
| James Mailbox | 0.991 | 0.329 | 0.783 |
| Jackrabbit FileVault | 0.990 | 0.057 | 0.637 |
| OpenJPA | 0.962 | 0.283 | 0.817 |
| Jackrabbit Oak | 0.962 | 0.488 | 0.817 |
| TomEE | 0.963 | 0.528 | 0.785 |
| Tika | 0.960 | 0.425 | 0.693 |
| Lucene - Core | 0.973 | 0.288 | 0.760 |
| Solr | 0.993 | 0.425 | 0.719 |
| Mahout | 0.969 | 0.450 | 0.747 |
| Apache Gora | 0.992 | 0.316 | 0.832 |
| Flume | 0.977 | 0.323 | 0.704 |
| Nutch | 0.967 | 0.582 | 0.788 |
| Apache Knox | 0.995 | 0.450 | 0.747 |
| Phoenix | 0.978 | 0.461 | 0.875 |

Na kolejnych rysunkach zamieszczono wykresy krzywej efektywności dla algorytmu Random Forest, dla pierwszych sześciu projektów. Im bardziej stroma jest krzywa efektywności tym lepsza efektywność przeszukiwania kodu źródłowego.




Rys. 4.2. Krzywa efektywności — ECF i Ant



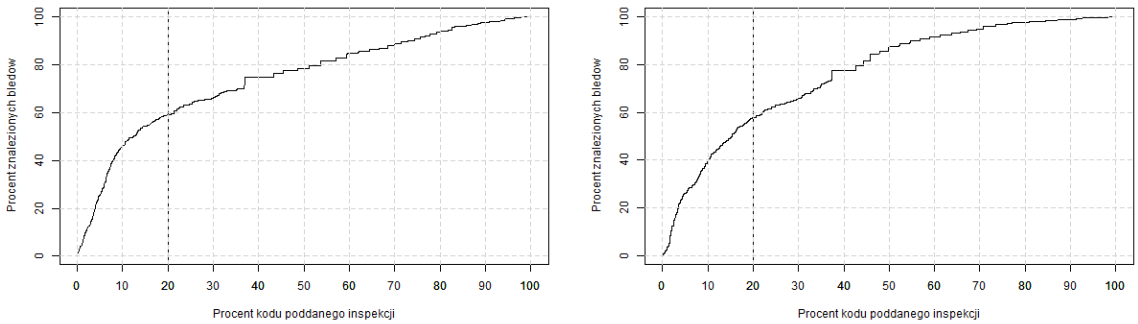

Rys. 4.3. Krzywa efektywności — JMeter i Commons Lang

Tabele 4.3 - 4.5 zawierają macierze pomyłek dla predykcji defektów w poszczególnych projektach z wykorzystaniem algorytmu Random Forest.

**Tabela** 4.3. Macierz pomyłek — ECF i Ant

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 9060 | 131 |
| | 0 | 10 | 105 |

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 8642 | 255 |
| | 0 | 71 | 152 |




Rys. 4.4. Krzywa efektywności — AspectJ i CXF

**Tabela** 4.4. Macierz pomyłek — JMeter i Commons Lang

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 208 | 57 |
| | 0 | 157 | 7170 |

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 11 | 2 |
| | 0 | 23 | 2519 |

**Tabela** 4.5. Macierz pomyłek — AspectJ i CXF

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 134 | 72 |
| | 0 | 301 | 17583 |

| | | przewidywany | |
|---|---|---|---|
| | | 1 | 0 |
| rzeczywisty | 1 | 182 | 409 |
| | 0 | 577 | 38973 |

# Rozdział 5

# Podsumowanie

**Trudności podczas realizacji pracy.** Podczas uruchamiania obliczeń zaplanowanych do wykonania w ramach projektu, okazało się, że czas przetwarzania na średniej klasy komputerze osobistym jest zdecydowanie zbyt długi. Po wykonaniu części obliczeń i zebraniu danych z 6 projektów zdecydowano o przeniesieniu wykonania programu do chmury obliczeniowej. Uruchomiono instację maszyny wirtualnej w chmurze Windows Azure. Niezbędne było uruchomienie instancji dysponującej odpowiednią ilością pamięci operacyjnej, przydzielono 14 GB pamięci RAM, z czego 4 GB przeznaczono na przechowywanie plików na wirtualnym dysku RAM.

Kolejnym wyzwaniem napotkanym w trakcie tworzenia rozwiązania było wykorzystanie wtyczek z pakietu DePress w środowisku KNIME, pomimo przeprowadzenia predykcji i obliczeń statystycznych w środowisku R. Dzięki wykorzystaniu trybu wsadowego KNIME uzyskano zadowalający rezultat, dający możliwość automatycznego przeprowadzenia obliczeń. Wymagane do tego jest jednak zainstalowanie KNIME oraz wtyczek z pakietu DePress.

**Zagrożenia dla wiarygodności.** Istnieją zagrożenia dla wiarygodności przeprowadzonego badania. Metoda wyszukiwania błędów bazuje na zgłoszeniach w systemie kontroli wersji oraz opisach rewizji w systemie wersjonowania kodu. Od jakości tych danych zależy jakość uzyskanych informacji o błędach w kodzie źródłowym. Część błędów mogła zostać źle opisana przez co niemożliwe było dopasowanie rewizji do zgłoszonego błędu. Z drugiej strony część zgłoszonych błędów może nie być błędami, co także wpływa na niedoskonałość zebranych danych.

Filtrowanie danych wejściowych także może być nieprecyzyjne. Opracowano roziązanie bazujące na wyłączaniu określonego jednego katalogu (razem z podkatalogami), co może być niewystarczające ze względu na układ zawartości niektórych repozytoriów.

**Podsumowanie.** Stworzone modele predykcji defektów na poziomie metod charakteryzują się wysoką skutecznością. Dzięki wykorzystaniu dużej liczby metryk procesu oraz odpowiedniego algorytmu klasyfikacji, można znacząco ograniczyć koszty przeglądu kodu, co z kolei pozwala na ograniczenie kosztów zapewnienia jakości w procesie tworzenia oprogramowania. Uzyskane wyniki dają podstawy do pozytywnej oceny stworzonych modeli predykcji. Dzięki zastosowaniu opracowanej metody w procesie wytwarzania oprogramowania, czas zaoszczędzony na pełnych przeglądach kodu, można przeznaczyć na inne zadania realizowane przez zespół programistów i testerów.

W ramach niniejszej pracy dyplomowej stworzono również wtyczkę KNIME służącą do wyliczenia metryk procesu AST (AstMetrics). Zbudowano też repozytorium metryk oraz zebrano metryki z 29 projektów. Pozyskane dane mogą posłużyć do budowania i ewaluacji nowych modeli predykcji w celu uzyskania jeszcze lepszych rezultatów. Repozytorium metryk opublikowano pod adresem
https://github.com/mkutyba/AstMetrics-dataset.

**Propozycja dalszych badań.** Zauważono możliwości dalszego rozwoju badania w celu osiągnięcia większej efektywności. Można dokonać wstępnej obróbki zmiennych wejściowych, np. stosując metody nadpróbkowania lub podpróbkowania albo przefiltrować predyktory wyłączając te, które mają wariancję bliską zeru (stałą wartość), są skorelowane, lub w inny sposób.

## Bibliografia


[1] Ieee guide for the use of ieee standard dictionary of measures to produce reliable software. *IEEE Std 982.2-1988*, 1989.

[2] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[3] Arisholm E., Briand L. C., Johannessen E. B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[4] Berthold M. R., Cebron N., Dill F., Gabriel T. R., Kötter T., Meinl T., Ohl P., Sieb C., Thiel K., Wiswedel B. KNIME: The Konstanz Information Miner. W: *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[5] Bevan J., Whitehead Jr E. J., Kim S., Godfrey M. Facilitating software evolution research with kenyon. W: *ACM SIGSOFT Software Engineering Notes*, 30(5):177–186. ACM, 2005.

[6] Catal C., Diri B. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[7] Chidamber S. R., Kemerer C. F. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[8] Concas G., Marchesi M., Murgia A., Tonelli R., Turnu I. On the distribution of bugs in the eclipse system. *Software Engineering, IEEE Transactions on*, 37(6):872–877, 2011.

[9] D'Ambros M., Lanza M., Robbes R. An extensive comparison of bug prediction approaches. W: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 31–41. IEEE, 2010.

[10] D'Ambros M., Lanza M., Robbes R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[11] Dyer R., Rajan H., Nguyen T. N. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. W: *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, 23–32. ACM, 2013.

[12] Endres A., Rombach H. D. *A handbook of software and systems engineering: empirical observations, laws and theories*. Pearson Education, 2003.

[13] Ferzund J., Ahsan S. N., Wotawa F. Empirical evaluation of hunk metrics as bug predictors. W: *Software Process and Product Measurement*, 242–254. Springer, 2009.

[14] Ferzund J., Ahsan S. N., Wotawa F. Software change classification using hunk metrics. W: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 471–474. IEEE, 2009.

[15] Fluri B., Wursch M., PInzger M., Gall H. C. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[16] Gantz J., Reinsel D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2012.

[17] Giger E., D'Ambros M., Pinzger M., Gall H. C. Method-level bug prediction. W: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 171–180. ACM, 2012.

[18] Giger E., Pinzger M., Gall H. C. Comparing fine-grained source code changes and code churn for bug prediction. W: *Proceedings of the 8th Working Conference on Mining Software Repositories*, 83–92. ACM, 2011.

[19] Godfrey M. W., Zou L. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2):166–181, 2005.

[20] Goodman L. A. Snowball sampling. *The annals of mathematical statistics*, 148–170, 1961.

[21] Górski J. *Inżynieria oprogramowania: w projekcie informatycznym.* Wydaw. MIKOM, 2000.

[22] Hall T., Beecham S., Bowes D., Gray D., Counsell S. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[23] Hassan A. E., Holt R. C. C-rex: an evolutionary code extractor for c. W: *CSER Meeting.* Citeseer, 2004.

[24] Hata H. Fault-prone module prediction using version histories. 2012.

[25] Hata H., Mizuno O., Kikuno T. Reconstructing fine-grained versioning repositories with git for method-level bug prediction. *IWESEP '10*, 27–32, 2010.

[26] Hata H., Mizuno O., Kikuno T. Historage: fine-grained version control system for java. W: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, 96–100. ACM, 2011.

[27] Hata H., Mizuno O., Kikuno T. Bug prediction based on fine-grained module histories. W: *Proceedings of the 2012 International Conference on Software Engineering*, 200–210. IEEE Press, 2012.

[28] Kamei Y., Matsumoto S., Monden A., Matsumoto K.-i., Adams B., Hassan A. E. Revisiting common bug prediction findings using effort-aware models. W: *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 1–10. IEEE, 2010.

[29] Kan S. H. *Metrics and models in software quality engineering.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[30] Koru A. G., El Emam K., Zhang D., Liu H., Mathew D. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473–498, 2008.

[31] Lessmann S., Baesens B., Mues C., Pietsch S. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[32] Ley M., Herbstritt M., Ackermann M. R., Hoffmann O., Wagner M., von Keutz S., Hostert K. Dblp bibliography, 2014.

[33] Liaw A., Wiener M. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[34] Madeyski L., Majchrzak M. Software measurement and defect prediction with depress extensible framework. *Foundations of Computing and Decision Sciences*, 39(4):249–270, 2014.

[35] Mende T., Koschke R. Revisiting the evaluation of defect prediction models. W: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 7. ACM, 2009.

[36] Mende T., Koschke R. Effort-aware defect prediction models. W: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 107–116. IEEE, 2010.

[37] Menzies T., Milton Z., Turhan B., Cukic B., Jiang Y., Bener A. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[38] Mitka P. Budowa modeli predykcji defektów w oparciu o metryki zmian na poziomie metod. Praca magisterska, Politechnika Wrocławska, 2014.

[39] Morzy T. *Eksploracja danych*. Wydawnictwo Naukowe PWN, Warszawa, 2013.

[40] Nguyen T. H., Adams B., Hassan A. E. Studying the impact of dependency network measures on software quality. W: *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 1–10. IEEE, 2010.

[41] Posnett D., Filkov V., Devanbu P. Ecological inference in empirical software engineering. W: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 362–371. IEEE Computer Society, 2011.

[42] Rahman F., Posnett D., Hindle A., Barr E., Devanbu P. Bugcache for inspections: hit or miss? W: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 322–331. ACM, 2011.

[43] Riaz M., Mendes E., Tempero E. A systematic review of software maintainability prediction and metrics. W: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 367–377. IEEE Computer Society, 2009.

[44] Śliwerski J., Zimmermann T., Zeller A. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[45] Zimmermann T. Fine-grained processing of cvs archives with apfel. W: *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, 16–20. ACM, 2006.