

Uniwersytet Jagielloński
Wydział Matematyki i Informatyki

ROZPRAWA DOKTORSKA

mgr inż. Jarosław Hryszko

**Zastosowanie predykcji defektów
w komercyjnych projektach rozwoju oprogramowania –
od rozważań teoretycznych do codziennej praktyki**

Promotor
dr hab. Adam Roman, prof. UJ

Kraków 2022

Spis treści

Spis rysunków	6
Spis tabeli	8
1. Wstęp	10
1.1. Wprowadzenie i motywacja do podjęcia zagadnienia	10
1.2. Cel rozprawy	12
1.3. Układ rozprawy	13
2. Predykcja defektów oprogramowania	15
2.1. Historia badań nad predykcją defektów oprogramowania	15
2.1.1. Rodzaje badań nad predykcją defektów	17
2.1.2. Predykcja defektów w przemyśle – aktualny stan wiedzy	18
2.2. Miary stosowane do oceny predykcji defektów	19
2.2.1. Miary stosowane do oceny klasyfikacji	19
2.2.2. Miary regresji	25
2.2.3. Odpowiedni wybór miar	25
2.3. Metryki wykorzystywane w predykcji defektów	26
2.3.1. Metryki kodu	26
2.3.2. Metryki procesowe	29
2.3.3. Inne metryki	31
2.3.4. Wybór odpowiedniego rodzaju metryk	32
2.4. Modele predykcji defektów	32
2.5. Dokładność predykcji	33
2.6. Wstępna obróbka zbiorów danych	34
2.6.1. Normalizacja	34
2.6.2. Wybór cech	35
2.6.3. Naprawa powiązań między zmianami w kodzie a zarejestrowanymi defektami	35
2.7. Predykcja międzyprojektowa	36
2.7.1. Uczenie transferowe	36
2.7.2. Wykonalność predykcji międzyprojektowej	38
2.8. Wykorzystanie predykcji defektów	39
2.9. Inne istotne kwestie	40
2.9.1. Niezbalansowanie klas	40
2.9.2. Bezpieczeństwo danych wykorzystywanych do predykcji	40

2.9.3. Porównanie predykcji defektów i statycznych analizatorów kodu	41
2.10. Wyzwania	41
3. Komercyjne projekty wytwarzania oprogramowania i ich słabe strony z perspektywy tematu badań	43
3.1. Pozyskiwanie informacji opisujących projekty komercyjnego wytwarzania oprogramowania	43
3.1.1. Narzędzie DePress i założona forma implementacji predykcji defektów	43
3.1.2. Wymagania stawiane projektom	46
3.2. Wyniki analizy	47
3.2.1. Elementy związane ze śledzeniem defektów	48
3.2.2. Elementy związane ze śledzeniem zmian w kodzie	50
3.3. Ulepszenia pozwalające na zastosowanie predykcji defektów w projektach	52
3.3.1. Elementy związane ze śledzeniem defektów	53
3.3.2. Elementy związane z repozytorium kodu	53
3.4. Podsumowanie tego etapu badań	54
4. Analiza efektywności kosztowej pilotażowego zastosowania predykcji defektów w komercyjnym projekcie wytwarzania oprogramowania	55
4.1. Po co badać efektywność kosztową predykcji defektów?	55
4.2. Komercyjny projekt wytwarzania oprogramowania	56
4.2.1. Charakterystyka wytwarzanego oprogramowania	56
4.3. Budowanie modelu predykcji	58
4.3.1. Wykorzystane zbiory danych	58
4.3.2. Przeciwdziałanie niezbalansowaniu klas	61
4.3.3. Proces wyboru cech	61
4.3.4. Modele predykcji	63
4.4. Metoda oceny efektywności kosztowej	64
4.4.1. Strategia planowania działań w procesie zapewniania jakości	64
4.4.2. Zwrot z inwestycji	65
4.4.3. Współczynnik kosztów i korzyści	66
4.5. Koszta procesu zapewniania jakości – rzeczywiste i symulowane	66
4.5.1. Rzeczywiste koszty zapewniania jakości w projekcie	66
4.5.2. Konstrukcja modelu i predykcja	68
4.5.3. Najwyższa wartość miary F i odpowiadająca jej wartość czułości predykcji	70
4.5.4. Symulacja kosztów predykcji	70
4.5.5. Koszta inwestycji	72
4.6. Wyniki	73
4.7. Podsumowanie tego etapu badań	75

5. Metodyka zapewniania jakości w komercyjnym projekcie wytwarzania oprogramowania wykorzystująca predykcję defektów	80
5.1. Wprowadzenie	80
5.2. Wybrany komercyjny projekt rozwoju oprogramowania	80
5.2.1. Charakterystyka projektu	80
5.2.2. Poziom jakości oprogramowania rozwijanego w ramach projektu	82
5.3. Metodyki zapewniania jakości w przedmiotowym projekcie rozwoju oprogramowania	84
5.3.1. Metodyka zapewniania jakości w czasie tworzenia wydania A	84
5.3.2. Metodyka zapewniania jakości wykorzystująca predykcję defektów oprogramowania (DPDQA) wykorzystana podczas tworzenia wydania B	85
5.4. Techniki zapewniania jakości w metodyce DPDQA	88
5.4.1. Wzorce projektowe	88
5.4.2. Pryncypia architektoniczne	89
5.4.3. Przeglądy kodu	90
5.4.4. Refaktoryzacja	91
5.4.5. Praktyki SOLID	92
5.4.6. Test Case-Driven Development	93
5.5. Model predykcji zbudowany przy pomocy narzędzia DePress	94
5.5.1. Wykorzystane dane	94
5.5.2. Klasyfikacja	95
5.5.3. Działania przeciwdziałające niezbalansowaniu klas	96
5.5.4. Procedura wyboru cech	97
5.6. Wyniki i podsumowanie tego etapu badań	97
5.6.1. Predykcja plików podatnych na defekty	98
5.6.2. Koszta implementacji	99
5.6.3. Spostrzeżenia z pracy z metodyką DPDQA	100
5.6.4. Zaobserwowany wpływ zastosowania predykcji defektów na jakość oprogramowania komercyjnego	103
6. Modyfikacja procesu predykcji defektów uwzględniająca zapachy kodu	110
6.1. Cel modyfikacji	110
6.2. Proces badawczy	111
6.2.1. Narzędzie CodeAnalysis jako źródło metryk związanych z zapachami kodu	112
6.3. Wyniki	113
6.3.1. Zaobserwowane anomalie wynikające z automatycznej generacji kodu	113
6.3.2. Różnice w metrykach i tego konsekwencje	115
6.3.3. Wybór optymalnego procesu predykcji defektów	115
6.3.4. Ewaluacja zbiorów danych: metryki zapachów kodu w porównaniu do metryk kodu źródłowego	117
6.4. Podsumowanie tego etapu badań	120

7. Podsumowanie	122
Dodatki	124
A. Charakterystyki badanych projektów	125
B. Wyniki analizy numeracji wersji oprogramowania widocznej w narzędziach śledzenia defektów	128
Bibliografia	132

Spis rysunków

2.1.	Przykład krzywej ROC.	23
2.2.	Przykładowe krzywe opłacalności, opisujące modele predykcji: optymalny (O), rzeczywisty (R) i losowy (L).	24
3.1.	Okno platformy KNIME wraz z narzędziem DePress (widocznym w lewym dolnym rogu), wraz z przykładowym diagramem przepływu danych.	44
3.2.	Uproszczony diagram przepływu danych, użyty w narzędziu DePress	45
3.3.	Separowanie różnego rodzaju modyfikacji kodu (pojedynczych napraw defektów, implementacji nowych funkcjonalności, itp.) poprzez wysyłanie ich do repozytorium kodu w osobnych „commitach” pozwala zaobserwować, które części kodu w pierwotnej wersji aplikacji były odpowiedzialne za później wykryte defekty.	46
3.4.	Rozróżnienie obszarów defektogennych i niedefektogennych (por. rysunek 3.3) umożliwia przygotowanie danych uczących – można przyjąć, że na powyższym przykładzie jeden wiersz w tabeli reprezentuje określony obszar kodu źródłowego, np. plik, klasę.	47
3.5.	Procentowy udział raportów o defektach bez informacji o wersji, z podziałem na projekty .	49
3.6.	Zła praktyka korzystania z systemu kontroli wersji: wiele zmian w jednym „commicie” wraz z komentarzem nie niosącym użytecznej informacji – por. rysunek 3.7.	50
3.7.	Przykładowe wyniki analizy komentarzy do commitów z komercyjnego projektu nie będącego przedmiotem dalszych badań. Na wykresie podano rzeczywiste komentarze wysyłane wraz z kodem do repozytorium. Zwraca uwagę duży odsetek komentarzy nie niosących użytecznych informacji, takich jak „CONVERTION”, „TEST” czy „THOMAS”.	52
4.1.	Diagram przepływu narzędzia DePress przeciwdziałający zjawisku niezbalansowania klas .	61
4.2.	Diagram przepływu narzędzia KNIME realizujący mechanizm wyboru cech	62
4.3.	Diagram przepływu narzędzia DePress dostarczający informacji o rozkładzie zmian w kodzie źródłowym, które nastąpiły w wyniku napraw defektów.	68
4.4.	Koszty zapewniania jakości w rozważanym projekcie: rzeczywiste w porównaniu z symulowanymi, gdy stosowane jest przewidywanie defektów na podstawie DePress.	75
4.5.	Wartość współczynnika ZwrotNetto w zależności od <i>Czułości</i> predykcji	76
5.1.	Liczba przypadków wsparcia dla wersji A systemu ROPTIS	83
5.2.	Metodyka zapewniania jakości stosowana podczas przygotowywania wersji A, zorientowana na osi czasu trwania projektu	84

5.3.	Proponowana metodyka zapewniania jakości wykorzystująca predykcję defektów oprogramowania (DPDQA) wykorzystana podczas tworzenia wydania B	86
5.4.	Zmodyfikowany diagram przepływu z rys. 3.2, pozwalający uzyskać informacje o rozkładzie zmian w kodzie źródłowym projektu ROPTIS, powstałych w wyniku napraw defektów.	96
5.5.	Ilustracja działania algorytmu SMOTE [177].	97
5.6.	W dolnej części okna środowiska programowania widoczny komunikat ostrzegawczy o pliku podatnym na uszkodzenia („This file contains a hot class!”), dostarczany przez wtyczkę korzystającą z wyników predykcji defektów. Po prawej stronie widoczne okno dialogowe ustawień wtyczki.	101
5.7.	Liczba przypadków wsparcia dla wersji B systemu ROPTIS.	104
5.8.	Charakterystyka funkcji wprowadzonych w wersji A systemu ROPTIS. Średnica koła jest proporcjonalna do całkowitej liczby funkcji zaklasyfikowanych do tej samej grupy ryzyka i złożoności.	107
5.9.	Charakterystyka funkcji wprowadzonych w wersji B systemu ROPTIS. Średnica koła jest proporcjonalna do całkowitej liczby funkcji zaklasyfikowanych do tej samej grupy ryzyka i złożoności.	108
6.1.	Anomalie w wartościach metryki „liczba ostrzeżeń o zapachach kodu” per rozmiar pliku (mierzony w LOC), wprowadzone w wyniku stosowania automatycznie generowanego kodu; później pominięte w analizie.	114
6.2.	Finalny rozkład wartości „liczba ostrzeżeń o zapachach kodu” per rozmiar pliku (mierzony w LOC) po wprowadzeniu poprawek.	115
B.1.	Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie B	128
B.2.	Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie P	129
B.3.	Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie T (Xates)	130
B.4.	Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie V	131

Spis tabeli

2.1.	Macierz pomyłek dla predykcji defektów oprogramowania	20
4.1.	Metryki kodu (opisane szerzej w punkcie 2.3.1) opisujące klasy języka Java zbierane przez narzędzie Eclipse Metrics 2.	58
4.2.	Liczba defektów i modułów (plików źródłowych) dla każdej z wersji oprogramowania . . .	60
4.3.	Średnie koszty naprawy defektu	67
4.4.	Rzeczywiste zasoby wykorzystane do naprawy defektów (QA = zapewnianie jakości)	67
4.5.	Wyniki predykcji: Czułość i miara F (w nawiasach) dla wszystkich zestawów eksperymentalnych	70
4.6.	Symulowane koszty zapewnienia jakości przy założeniu, że zastosowano predykcję defektów (QA = zapewnianie jakości).	72
4.7.	Koszty inwestycji w predykcję defektów (w projekcie)	73
5.1.	Całkowita liczba defektów wykryta podczas czasu użytkowania wersji A programu ROPTIS	82
5.2.	Wyniki walidacji modelu predykcji obszarów w kodzie podatnych na defekty – wykorzystanie próbkowania klasy większościowej.	99
5.3.	Wyniki walidacji modelu predykcji obszarów w kodzie podatnych na defekty – wykorzystanie SMOTE.	100
5.4.	Koszty wprowadzenia predykcji defektów w projekcie ROPTIS zarejestrowane przy użyciu oprogramowania SAP CATS.	100
5.5.	Liczba defektów w porównywanych wersjach ROPTIS.	103
5.6.	Liczba defektów związanych z wydajnością programu.	105
5.7.	Zaobserwowane zmiany w wydajności po wprowadzeniu proponowanej metodyki.	105
5.8.	Dzięsiątka największych przyrostów wydajności w wersji B.	106
6.1.	Ostrzeżenia dotyczące zapachów kodu w narzędziu CodeAnalysis	113
6.2.	Przykład wycinka zbioru danych z podejścia pierwszego – pojedyncza klasa per rekord (wartości metryk z narzędzia SourceMonitor są sztucznie podzielone przez liczbę klas w pliku źródłowym).	116
6.3.	Przykład zbioru danych z podejścia drugiego – pojedynczy plik per rekord (wartości metryk zapachów kodu zostały zsumowane dla plików wieloklasowych).	116
6.4.	Kombinacje różnych rozwiązań.	116
6.5.	Wyniki tej części eksperymentu, której celem było ustalenie najlepszego zestawienia rozwiązań używanych w procesie predykcji (dla klasy „defektogennej”).	118
6.6.	Pomiary dla rekordów określonych jako podatne na defekty.	119

A.1. Charakterystyka projektu V	125
A.2. Charakterystyka projektu P	126
A.3. Charakterystyka projektu T (Xates)	126
A.4. Charakterystyka projektu B	127
A.5. Charakterystyka projektu M	127

1. Wstęp

1.1. Wprowadzenie i motywacja do podjęcia zagadnienia

W 1956 roku, dwudziestoosmioletni wówczas pisarz amerykański, Philip K. Dick (okrzyknięty później „Szekspirem science fiction”) opublikował krótki utwór pod tytułem „Raport mniejszości”. Przedstawił tam wizję społeczeństwa przyszłości, w którym wybrane osoby o nadprzyrodzonych zdolnościach przewidują zbrodnie, zanim one nastąpią. Jasnowidzący, podłączeni do wielkiego komputera, pozwalają policji aresztować podejrzanych, zanim ci zdążą popełnić jakiegokolwiek przestępstwo [36].

Sześćdziesiąt sześć lat po publikacji „Raportu mniejszości” nasza cywilizacja praktycznie w każdym aspekcie życia korzysta z komputerów i ich nieodłącznego składnika – oprogramowania. Często firmy wytwarzające oprogramowanie muszą utrzymywać duże i skomplikowane programy komputerowe i ciągle dbać o ich jakość – zapewnianie jakości może pochłaniać nawet 40-50% wszystkich kosztów projektów rozwoju oprogramowania [81]. Przeciwdziałanie defektom we wczesnych stadiach procesu wytwarzania oprogramowania, poprzez stosowanie najlepszych praktyk z zakresu zapewniania jakości na etapie projektowania i pisania kodu, znacząco obniża całkowite koszty wytwarzania – mówi o tym tzw. zasada Boehm’a: kosztu usuwania defektów rosną wraz z kolejnymi stadiami procesu wytwarzania oprogramowania [20]. Ograniczony czas i zasoby sprawiają, że jednakowo dokładne zastosowanie metod zapewniania jakości dla każdego elementu wytwarzanego oprogramowania jest trudne do osiągnięcia, a nawet niemożliwe. Dodatkowo, niektórzy badacze [42, 168, 180] argumentują, że w wytwarzaniu oprogramowania można zaobserwować zasadę Pareto – defekty w kodzie źródłowym programów nie są rozłożone jednolicie i jedynie około 20% kodu powoduje około 80% wykrywalnych defektów. Dlatego też posiadanie informacji o tym, które elementy oprogramowania są (będą) szczególnie podatne na defekty pozwala na odpowiednio wczesne – a tym samym tańsze – zastosowanie szeregu praktyk zapewniania jakości, skupiając się w szczególności na powyższych, potencjalnie defektogennych elementach. Pozwoliłoby to znacząco obniżyć ryzyko wystąpienia defektów, zanim w ogóle się one pojawią.

Wizja Philipa K. Dicka o posiadaniu informacji o *przykrych* zjawiskach, *zanim* się one przydarzą, nigdy i nigdzie nie była bardziej nęcąca, niż dziś, w świecie wytwarzania oprogramowania.

Począwszy od lat siedemdziesiątych XX wieku, podejmowano wiele prób przewidzenia najbardziej podatnych na defekty fragmentów oprogramowania. Na początku obserwowano metryki kodu źródłowego, takie jak liczba linii kodu [2] czy złożoność cyklomatyczna [131] i wiązano je z ryzykiem wystąpienia defektu. W 1992 roku Munson opublikował badania opisujące zastosowanie procesu uczenia maszynowego do predykcji obszarów w kodzie potencjalnie podatnych na defekty [145].

Od tamtego czasu predykcja defektów stała się dynamicznie rozwijającą się gałęzią inżynierii oprogramowania [34, 109, 134, 150, 225, 173]. Modele predykcyjne są w stanie wskazać elementy oprogramowania szczególnie podatne na defekty, dzięki czemu zespoły zapewniania jakości mogą w efektywny sposób wykorzystać ograniczone środki przeznaczone na testy i analizę danego oprogramowania [134, 109, 225].

Jednakowoż, metody predykcji defektów bazujące na uczeniu maszynowym jak do tej pory nie zdobyły popularności w komercyjnym wytwarzaniu oprogramowania. Jako jeden z głównych powodów tego stanu rzeczy wskazuje się fakt, iż modele predykcji są trudno przenaszalne między kontekstami [67] oraz to, że narzędzia, które pozwalają na budowę modeli predykcji nie są przystosowane do specyficznych warunków panujących w wielkoskalowym, komercyjnym wytwarzaniu oprogramowania.

Dodatkowo, Lanza et al. [108] przekonywali, że ocena skuteczności predykcji defektów poprzez konfrontowanie jej z faktycznie znalezionymi defektami (w tym samym kodzie źródłowym) jest sama w sobie błędna i że predykcja defektów ma sens tylko wtedy, jeśli jest użyta *in vivo*, np. gdy modele predykcji stosowane są w komercyjnych projektach wytwarzania oprogramowania, jako istotna część metodyki zapewniania jakości tamże. Wpływ zastosowania predykcji natomiast powinno oceniać się poprzez obserwację, jak zmieniła się jakość wytwarzanego oprogramowania po wdrożeniu predykcji defektów jako ważnego elementu metodyki zapewniania jakości.

Co zaś tyczy się samych metodyk zapewniania jakości w wielkoskalowym wytwarzaniu oprogramowania (nie wykorzystujących predykcji defektów), są one szeroko udokumentowane w literaturze. Istnieje duże zainteresowanie komercyjnymi zastosowaniami metodyk stosujących model CMMI [60, 58, 215]. Z kolei komercyjne wykorzystanie innego zorientowanego jakościowo podejścia zwanego Six Sigma zostało podsumowane przez Kwaka i Anbari'ego [106]. Metoda ciągłego rozwoju procesu testowania oprogramowania, bazująca na Six Sigma została opisana przez Galinac'a i Car'a [54]. Stosowanie dobrych praktyk¹ w wytwarzaniu oprogramowania komercyjnego było analizowane przez wielu autorów. Na przykład Dinakar analizował przypadki, gdy dobre praktyki były ignorowane [37],

¹ Mariusz Bednarek w swojej książce "Doskonalenie systemów zarządzania" (2007) pisze: "Dobrymi praktykami określamy działanie przynoszące konkretne i pozytywne rezultaty oraz zawierające w sobie pewien potencjał innowacji. Jest ono trwałe i powtarzalne oraz możliwe do zastosowania w podobnych warunkach w innym miejscu lub przez inne podmioty"

podczas gdy Fitzgerald et al. badał te projekty, gdzie praktyki były stosowane [48]. Wykorzystanie wzorców projektowych w zapewnianiu jakości badał Riehle [178]. Bubeck et al. badali projekt komercyjny, gdzie eksperymentowano z całą paletą dobrych praktyk celem podniesienia jakości [24]. Oryginalne metody zapewniania jakości w projektach wytwarzania oprogramowania zostały zaproponowane przez Chena et al. [28] i Grieskampa et al. [64]. Podobnie nowatorskie, holistyczne podejście do zapewniania jakości oprogramowania zostało opisane przez Grutznera et al. [65].

Jednakowoż, w momencie podjęcia tematu, według najlepszej wiedzy autora, nie było badań nad faktycznym zastosowaniem metodyki zapewniania jakości *wykorzystującej predykcję defektów* w wielkoskalowym, komercyjnym wytwarzaniu oprogramowania. Stanowiło to *główną motywację* dla autora do podjęcia tematu pracy. Dodatkowo motywacją były: potrzeba poprawy jakości w projektach dostarczających oprogramowanie krytyczne dla produkcji pojazdów oraz bardzo mała liczba znanych publikacji opisujących użycie predykcji defektów w *komercyjnych* projektach wytwarzania oprogramowania. Tutaj autor wymienić może prace Khoshgoftaar'a et al. (1995, 1997); Khoshgoftaar'a i Seliya (2004, 2005); Ostranda i Weyuker (2002); Ostranda et al. (2005); Turhana et al. (2009) [94, 93, 95, 96, 155, 156, 206, 207]. Jednakże tylko Ostrand et al. (2010), Li et al. (2006), Tosun et al. (2010) opisali zbiór przypadków właściwego użycia predykcji defektów i przedstawili korzyści dla przemysłu jakie z niej wynikają [158, 113, 204].

Fakt, że badaniami będącymi przedmiotem niniejszej pracy, udało się zainteresować dział wytwarzania oprogramowania należący do jednej z wiodących, światowych firm, był dodatkowym, ale bardzo ważnym czynnikiem motywującym. Od momentu podjęcia prac, firma ta była nieocenionym partnerem w badaniach nad zastosowaniem predykcji defektów oprogramowania w projektach komercyjnych.

1.2. Cel rozprawy

Ogólnym celem pracy jest zaproponowanie nowej metodyki zapewniania jakości dla komercyjnych projektów wytwarzania oprogramowania, wykorzystującej jako jeden z kluczowych elementów predykcję defektów oraz wdrożenie tej metodyki w komercyjnym środowisku wytwarzania oprogramowania i zbadanie jej wpływu na końcową jakość produktu (oprogramowania).

Cele szczegółowe pracy to:

Cel 1: Określenie, jakie elementy wytwarzania oprogramowania (np. metody pracy, narzędzia, wymieniane dane) muszą zostać zachowane, aby wdrożenie predykcji defektów w komercyjnych projektach rozwoju oprogramowania zakończyło się powodzeniem.

- Cel 2: Ocena wybranych projektów rozwoju oprogramowania prowadzonych w przemyśle pod kątem możliwości zastosowania predykcji defektów oprogramowania.
- Cel 3: Rozpoznanie wąskich gardeł, które mogą wpłynąć w negatywny sposób na predykcję.
- Cel 4: Propozycja zmian, które mogą zostać wprowadzone w komercyjnych projektach rozwoju oprogramowania, celem łatwej implementacji predykcji defektów w przyszłości.
- Cel 5: Przeprowadzenie symulacji tego, w jaki sposób zmieniłyby się koszty zapewniania jakości w wybranym komercyjnym projekcie rozwoju oprogramowania, gdyby zastosować predykcję defektów oprogramowania i w pełni skorzystać z jej wyników.
- Cel 6: Wdrożenie predykcji defektów jako istotnego elementu procesu zapewniania jakości w komercyjnym projekcie rozwoju oprogramowania i zaproponowanie metodyki zapewniania jakości oprogramowania wykorzystującej predykcję defektów oprogramowania.
- Cel 7: Zaproponowanie sposobu zbadania, w jaki sposób metodyka wykorzystująca predykcję defektów wpłynęła na jakość wytwarzanego przemysłowo oprogramowania.
- Cel 8: W trakcie powyższych prac – zbadanie, jakie wybrane rozwiązania z obszaru uczenia maszynowego (klasyfikatory, algorytmy wyboru cech, algorytmy przeciwdziałania nieźrównoważeniu klas itp.) osiągają najlepsze rezultaty w szczególnym środowisku, jakim są komercyjne projekty wytwarzania oprogramowania.
- Cel 9: Przeprowadzenie modyfikacji procesu predykcji – zbadanie, w jaki sposób na wyniki predykcji defektów oprogramowania wpłynie zastosowanie mało popularnych w środowisku komercyjnym metryk opartych o tzw. zapachy kodu źródłowego.

1.3. Układ rozprawy

Układ rozprawy jest następujący. W rozdziale 2 zostały pokrótce przybliżone czytelnikowi najważniejsze aspekty dziedziny, jaką jest predykcja defektów oprogramowania, wykorzystująca proces uczenia maszynowego. Przedstawiony został typowy proces predykcji defektów w oprogramowaniu, a także – w punkcie 2.1.1 – najczęściej stosowane w odniesieniu do niego metody badawcze. Jako, że w literaturze problemu znaleźć można różne metody ewaluacji modeli predykcji, zostały one przedstawione w punkcie 2.2. Punkty 2.3 i 2.4 poświęcone zostały metrykom i modelom opisanym w analizowanych źródłach. Punkt 2.5 opisuje dokładność predykcji. W niektórych badaniach, zanim odpowiedni model zostanie zbudowany, wykorzystuje się wstępną obróbkę danych. Pozwala to na zwiększenie wydajności procesu. W punkcie 2.6 opisane zostały techniki takiej wstępnej obróbki. Ponieważ najnowsze badania w tej tematyce skupiały się na analizie predykcji międzyprojektowej, zdecydowano się na ich analizę w punkcie 2.7. W następnych dwóch punktach przybliżone zostały znane zastosowania predykcji defektów oraz omówione inne tematy, które pojawiły się w trakcie opisywanych w niniejszej pracy badań.

W rozdziale 3 rozważono wybrane komercyjne projekty rozwoju oprogramowania pod kątem wdrożenia ewentualnej predykcji defektów. Był to wczesny etap badań będących przedmiotem niniejszej rozprawy, zatem należało przyjąć pewne założenia co do przyszłego procesu predykcji defektów oprogramowania. Założenia te zostały przedstawione w punkcie 3.1.1. Po przeanalizowaniu wybranych projektów pod kątem powyższych założeń, wyniki analizy zostały przedstawione w punkcie 3.2. Znając potencjalne „wąskie gardła”, na końcu rozdziału zaproponowano ulepszenia, jakie można wprowadzić w przemyśle zajmującym się rozwojem oprogramowania, aby móc bez przeszkód implementować predykcję defektów oprogramowania.

Brak wąskich gardeł nie jest wystarczający, aby technikę predykcji defektów zacząć stosować w przemyśle wytwarzającym oprogramowanie – potrzebne jest także uzasadnienie biznesowe. O tym traktuje rozdział 4, poruszający kwestię ekonomii wytwarzania oprogramowania. Przedstawiono tam przebieg i wyniki badania, jak mogłyby zmienić się rzeczywiste (faktyczne) koszty zapewniania jakości w wybranym komercyjnym projekcie rozwoju oprogramowania, gdyby – dzięki zastosowaniu modelu predykcji – wyeliminowano defekty we wczesnej fazie projektu. W celu przeprowadzenia badania zbudowano model predykcji i wykonano jego ewaluację, w obu przypadkach wykorzystując rzeczywiste dane z projektu komercyjnego – co opisano w punkcie 4.3. Następnie wykonano symulację wydatków na zapewnianie jakości, gdyby planowanie działań związanych z zapewnianiem jakości w pełni wykorzystywało informacje, jakie niesie model predykcyjny. Proponowaną metodę badawczą opisano w punkcie 4.4, zaś sam eksperyment wraz z wynikami – w punkcie 4.5.

Wytypowanie odpowiedniego projektu i udowodnienie efektywności kosztowej umożliwiło uzyskanie zielonego światła od zarządu firmy do rozpoczęcia eksperymentu z zastosowaniem predykcji defektów *in vivo*, o czym traktuje rozdział 5. W punkcie 5.3.2 zaproponowano nową metodykę, wykorzystującą predykcję defektów. Dla porównania, metodykę zapewniania jakości stosowaną do tej pory przedstawiono w punkcie 5.3.1. W zaproponowanej metodyce ważną rolę pełnią dobre praktyki zapewniania jakości; przybliżono je w punkcie 5.4. Kolejnym ważnym elementem metodyki jest, oczywiście, model predykcji – proces jego budowy szczegółowo opisano w punkcie 5.5. Obserwacje dokonane podczas pracy z nową, zaproponowaną metodyką wykorzystującą predykcję defektów oprogramowania w rzeczywistym, komercyjnym projekcie wytwarzania oprogramowania, zgromadzono w punkcie 5.6.3. Zaś rezultaty eksperymentu z zastosowaniem powyższej metodyki, przedstawiono wraz z wnioskami w punkcie 5.6.4.

W rozdziale 6 opisano, w jaki sposób na wyniki predykcji defektów oprogramowania wpłynie zastosowanie mało popularnych w środowisku komercyjnym metryk opartych o tzw. zapachy kodu źródłowego.

2. Predykcja defektów oprogramowania

Większość badań związanych z tematyką predykcji defektów oprogramowania wykorzystuje algorytmy uczenia maszynowego [10, 19, 39, 70, 109, 134, 147]. Pierwszym krokiem w budowie modelu predykcji jest wygenerowanie instancji z archiwów przechowujących dane opisujące oprogramowanie (takich jak systemy kontroli wersji, aplikacje służące śledzeniu defektów, archiwa maili itd). Każda instancja może reprezentować system, komponent (bądź paczkę) oprogramowania, kod źródłowy pliku, klasę, funkcję (metodę) i/lub zmiany w kodzie, w zależności od pożądanej szczegółowości wyników predykcji. Instancja posiada kilka metryk (lub właściwości) zebranych z archiwum oprogramowania i jest opatrzona dwoma etykietami, odpowiednio: „pozbawiona defektów” albo „zawiera defekty” – lub – opisana za pomocą wielu etykiet (tzw. predykcja wieloetykietowa).

Po wygenerowaniu instancji, które zawierają metryki i etykiety, można zastosować techniki wstępnej obróbki danych, wykorzystywane przy uczeniu maszynowym. Techniki te to między innymi: wybór odpowiednich kryteriów, normalizacja danych, a także redukcja szumów [99, 134, 150, 190, 219]. Wstępna obróbka danych jest opcjonalna, dlatego też techniki te nie są wykorzystywane we wszystkich badaniach związanych z predykcją defektów oprogramowania [39, 109].

Dysponując zbiorem uczącym powstałym poprzez zebranie instancji z archiwów oprogramowania, można zbudować model predykcyjny. Model ów pozwala przewidzieć, czy nowa instancja będzie zawierać defekt, czy też nie (predykcja defektów). Podobnie jak w przypadku zbioru uczącego, podatność na defekty jest opisana dwoma rodzajami etykiet jak wyżej bądź – w przypadku predykcji wieloetykietowej – odpowiednio większą ich liczbą.

2.1. Historia badań nad predykcją defektów oprogramowania

Pierwszy kwestię przewidywania obszarów defektogennych w oprogramowaniu analizował Akiyama w roku 1971 [2]. Akiyama stworzył prosty model wykorzystujący liczbę linii kodu źródłowego (LOC – ang. *Lines of Code*), postulując, że LOC przedstawia złożoność oprogramowania. Niestety LOC była metryką zbyt prostą, aby w pełni opisać złożoność całego systemu. W konsekwencji McCabe i Halstead niezależnie (odpowiednio w latach 1976 i 1977 [131, 68]) zaproponowali pomiar tzw. złożoności cyklomatycznej. Metryki McCabe’a i Halstead’a były bardzo popularne w latach siedemdziesiątych i na początku lat

osiemdziesiątych [45] ubiegłego stulecia (dziś wiemy, że większość badań empirycznych nie potwierdza założeń i definicji złożoności Halstead’a – więcej na ten temat czytelnik znajdzie w punkcie 2.3.1).

Należy jednak odnotować, że modele z tego okresu nie były modelami czysto predykcyjnymi dla defektów w oprogramowaniu. Służyły one raczej do opisanie związku między metryką a liczbą defektów [45]. Pierwsze próby faktycznej predykcji podjęli Shen et al. budując model regresji liniowej i testując go na nowo napisanym kodzie [185]. Jednakże, Munson et al. stwierdzili, że technika ta w tamtym czasie nie była precyzyjna i zaproponowali algorytm, który klasyfikował moduły na dwie kategorie: wysokiego i niskiego ryzyka [145]. Co ciekawe, w systemie, na którym był testowany, algorytm uzyskał dokładność na poziomie 92%.

Jednakże propozycja Munsona wciąż posiadała istotne ograniczenia, jak na przykład brak wsparcia dla systemów zorientowanych obiektowo. Chidamber i Kemerer zaproponowali w związku z tym w 1994 roku zestaw zorientowanych obiektowo metryk [31], które zostały wykorzystane przez Basilego w celu przewidzenia potencjalnych defektów w systemach obiektowych [12].

W pierwszej dekadzie XXI wieku wciąż istniało kilka istotnych wyzwań dla predykcji defektów. Po pierwsze, modele predykcyjne starano się wykorzystywać w zapewnianiu jakości dopiero tuż przed oddaniem nowego oprogramowania do użycia (innymi słowy – przed wdrożeniem na środowisko produkcyjne). Jednakże proces zapewniania jakości wykorzystujący predykcję defektów stałby się praktyczniejszy i tańszy (patrz wspomniana wcześniej reguła Boehm’a), gdyby istniała możliwość przewidywania obszarów defektogennych po każdej zmianie kodu źródłowego. Aby to umożliwić, Mockus et. al. stworzyli w 2000 roku model dostarczający informacji o defektogennym kodzie przy każdym przyroście kodu źródłowego [138]. Obecnie takie podejście nazywane jest just-in time (JIT – dokładnie na czas) a algorytmy opierające się na JIT wciąż zyskują na popularności [52, 91, 98].

Drugim wyzwaniem stojącym przed techniką predykcji defektów oprogramowania jest fakt, że stworzenie modelu predykcji dla projektów *nowych* lub takich, które *dysponują małą liczbą danych historycznych*, było bardzo utrudnione lub praktycznie niemożliwe. W miarę zyskiwania na popularności praktyk stosowania metryk kodu źródłowego w predykcji, ograniczenie powyższe stało się jednym z największych problemów w badaniach nad predykcją defektów [227]. Aby rozwiązać ten problem, badacze zaproponowali tzw. predykcję międzyprojektową [117, 150, 207, 212]. Zimmermann et al. oraz He et al. przeprowadzili badania związane z dokładnością algorytmów przewidujących dla predykcji międzyprojektowej [72, 227]. O tym, czym jest predykcja międzyprojektowa, dowie się czytelnik z punktu 2.7 niniejszej rozprawy.

Trzecie wyzwanie wiązało się z problemem „Czy systemy przewidujące prawdopodobieństwo wystąpienia defektów w oprogramowaniu są w jakikolwiek sposób przydatne dla rozwoju przemysłu?”. W rezultacie przeprowadzono badania, które sprawdziły ten aspekt, a także zaproponowano praktyczne rozwiązania tego problemu [43, 111, 174]. Żaden z badaczy jednakowoż nie zaproponował – według najlepszej wiedzy autora dysertacji – rozwiązania, które można by stosować jako stały element metodyki zapewniania jakości oprogramowania. **Niniejsza rozprawa opisuje prace nad wypełnieniem tej luki.**

Przeprowadzono także badania, które analizowały trendy w rozwoju technologii informatycznych. Wykorzystując internetowe serwisy wspomagające wytwarzanie oprogramowania, badacze tacy jak Zimmermanna et al. [225], Pingzera et al. [164] oraz Taba'e et al. [199] byli w stanie zaproponować wykorzystanie nowych rodzajów metryk w predykcji defektów. Ewentualne problemy z wykorzystaniem informacji wrażliwych (np. danych osobowych) przy wykorzystaniu ogólnie dostępnych repozytoriów zostały przedstawione przez Petersa et al. [163].

2.1.1. Rodzaje badań nad predykcją defektów

Wiele z prowadzonych w ostatniej dekadzie badań skupiało się na zaproponowaniu nowych metryk służących do budowania danego modelu. Proponowane były na przykład metryki projektowe [198], metryki zmian [143] czy metryki eksploracyjne [148], ale najszerzej badane w kontekście predykcji defektów były metryki kodu źródłowego [67, 82] oraz metryki procesowe [171, 120, 87]. Te związane z kodem źródłowym, jak sama nazwa wskazuje, opisują określone cechy kodu źródłowego. Metryki procesowe pozyskiwane są z takich źródeł jak systemy kontroli wersji oprogramowania czy systemy, które zarządzają procesem rozwoju tegoż. Tym samym pozwalają opisać wiele aspektów rozwoju oprogramowania, takich jak zmiany w kodzie źródłowym, liczbę autorów (programistów) danego fragmentu kodu, relacje między programistami itd. Przydatność metryk procesowych została udowodniona w wielu badaniach [70, 109, 147, 171].

Jak zostało przedstawione wcześniej, większość badań związanych z przewidywaniem występowania defektów jest oparta na uczeniu maszynowym. Modele predykcji, budowane za pomocą algorytmów uczenia maszynowego, mogą określać zarówno podatność kodu źródłowego na defekty (klasyfikacja) lub liczbę defektów w kodzie programu (regresja). W celu zwiększenia wydajności predykcji, wciąż testuje się nowe techniki w zakresie uczenia maszynowego, jak na przykład częściowo nadzorowane uczenie [112, 208], czy algorytm BugCache. Algorytm ten wykorzystuje informacje o lokalizacji historycznych defektów oraz listę obszarów kodu źródłowego najbardziej podatnych na defekty [101].

Badania obejmowały także próby zwiększania dokładności (czyli granulacji wyników) predykcji. Początkowo modele predykcji dążyły do identyfikacji obszarów defektogennych

na poziomie całego systemu, jego komponentów, plików i klas. Kolejne prace skutkowały zwiększeniem dokładności wyników predykcji do poziomu metody czy nawet pojedynczej zmiany w kodzie [71, 98]. Dalszy rozwój badań w tym obszarze jest szczególnie istotny dla twórców programów – pozwoli bowiem dostarczyć producentom oprogramowania informacji o dokładnej lokalizacji defektogennych części kodu.

Opracowywanie sposobów wstępnej obróbki danych dla modeli predykcji jest również niezwykle istotną kategorią badań zakresie predykcji defektów. Zanim zbudowany zostanie model predykcji, zastosowane mogą być takie techniki jak wybór cech [190, 147], normalizacja [134, 150] czy redukcja szumów [99]. Użycie technik wstępnej obróbki danych wpływa na zwiększenie jakości predykcji [99, 190, 219]. Badacze zaproponowali także metody, dzięki którym modele predykcji defektów można przenosić między projektami wytwarzania oprogramowania (tzw. cross-project).

Większość badań, które zostały opisane powyżej, były przeprowadzone i oceniane w ściśle określonym otoczeniu (kontekście, projekcie). Często modele predykcji były budowane oraz testowane w tym samym projekcie wytwarzania oprogramowania. Niestety, w przypadku nowego projektu, czyli dopiero co napisanego nowego programu, który nie posiada bogatej historii naprawy defektów (bądź nie ma jej wcale), bardzo trudne jest stworzenie odpowiedniego rozwiązania predykcji przyszłego występowania defektów. W takim przypadku badacze proponują metody takie jak kompensacja metryk [212], filtr NN (Nearest Neighbour – najbliższy sąsiad [207], TNB (Transfer Naive Bayes) [117] i TCA+ [150]. Metody te adaptują modele predykcji w nowym projekcie poprzez wybór podobnych wzorców w kodzie, transfer wybranych danych lub stworzenie nowego modelu predykcji uwzględniającego powyższe, specyficzne okoliczności [150, 117, 207, 212].

Interesującym tematem badań w zakresie predykcji defektów w podejściu międzyprojektowym (cross-project) jest analiza wykonalności. Wielu badaczy zakłada, że predykcja cross-project jest trudna do osiągnięcia i jedynie nieliczne techniki predykcji w takich kontekstach zyskują zadowalające wyniki [227]. Zatem znalezienie efektywnej techniki predykcji defektów cross-project wciąż stanowi atrakcyjny obszar badań. Obiecujące wyniki daje wykorzystanie algorytmów wykorzystujących drzewa decyzyjne. Jednakże, drzewa decyzyjne były badane jedynie z wykorzystaniem ograniczonych zbiorów danych i nie przeanalizowano ich w odpowiednio szerokim zakresie [227, 72].

2.1.2. Predykcja defektów w przemyśle – aktualny stan wiedzy

Pomimo tego, że w ciągu ostatnich kilkudziesięciu lat napisano olbrzymią liczbę prac naukowych o predykcji defektów, to studium przypadków komercyjnych skupiające się na wdrożeniu, głównych przeszkodach czy zyskach wynikających z użycia predykcji defektów w środowisku komercyjnym jest zjawiskiem niezwykle rzadkim. Przeglądy systematyczne

aktualnego stanu wiedzy na temat predykcji defektów zostały opublikowane w 1999 przez Fenstona i Neila [45] w 2009 przez Catala i Dirí [25] oraz – w najbardziej obszernej formie – przez Halla et al. w 2012 roku [67]. Prace te udowadniają, że tylko kilka opublikowanych badań bazowało na realnych, komercyjnych aplikacjach w zakresie predykcji defektów.

Pierwsza publikacja związana z tym tematem pojawiła się w roku 1997 i została napisana przez Khoshgoftaar et al. [93]. Było to studium przypadków w zakresie modelowania jakościowego w dużym systemie telekomunikacyjnym. Autorzy wykorzystali sieć neuronową do zbudowania realnego systemu predykcji przyszłych defektów zarządzanego przez firmę telekomunikacyjną, ale ostatecznie rezultaty nie zostały wykorzystane do wsparcia procesów zapewniania jakości firmy. Dwie inne publikacje Khoshgoftaara i Seliya z 2004 i 2005 kontynuowały poprzedni pomysł i skupiły się co prawda na analizie danych pochodzących z projektów komercyjnych, ale nie na wykorzystaniu predykcji w praktyce [95, 96]. Podobne podejście można zauważyć w większości prac wykorzystujących dane komercyjne – przykłady można znaleźć w różnych publikacjach Ostranda [155, 158], Tosuna [204] i Turhana [206, 207]. Jednostkowe zastosowanie informacji uzyskanych za pomocą predykcji defektów można znaleźć w pracach takich jak [218], [198] i [104]. Według najlepszej wiedzy autora, jedyne kompletne badania opisujące wprowadzenie predykcji defektów w środowisku komercyjnym, zostały przedstawione przez Ostranda [156], Li et al. [113] i Tosuna [205].

W tym miejscu warto także nadmienić o wkładzie polskich badaczy – Jureczko i Madeyski wykorzystali dane zebrane z pięciu komercyjnych projektów rozwoju oprogramowania do zbudowania modelu predykcji defektów [86]. Badacze ci zaproponowali na potrzeby predykcji defektów nową metrykę procesową, analizując szereg projektów, w tym komercyjnych [120]. Niestety, według najlepszej wiedzy autora, techniki predykcji defektów nie zostały wykorzystane w metodykach zapewniania jakości w komercyjnym wytwarzaniu oprogramowania.

2.2. Miary stosowane do oceny predykcji defektów

Literatura przedmiotu podaje różne miary stosowane w celu oceny efektywności predykcji defektów.

2.2.1. Miary stosowane do oceny klasyfikacji

Wyniki predykcji – elementy oprogramowania sklasyfikowane podczas procesu predykcji jako podatne na defekty (defektogenne) lub niepodatne na defekty (nie-defektogenne) – mogą być porównane z faktycznymi danymi opisującymi rozkład elementów oprogramowania podatnych na defekty i zostać użyte do zbudowania tak zwanej macierzy pomyłek (tabela 2.1)

– powszechnie wykorzystywanego narzędzia do określenia efektywności algorytmów dających wyniki kategoryczne [67].

Tabela 2.1: Macierz pomyłek dla predykcji defektów oprogramowania

Elementy oprogramowania	Sklasyfikowane jako defektogenne	Sklasyfikowane jako niedefektogenne
Faktycznie defektogenne	Prawdziwie pozytywne (TP)	Fałszywie negatywne (FN)
Faktycznie niedefektogenne	Fałszywie pozytywne (FP)	Prawdziwie negatywne (TN)

W celu zmierzenia wyników predykcji defektów oprogramowania dostarczanych przez klasyfikatory (modele, które klasyfikują), istotne jest wzięcie pod uwagę następujących rodzajów wyników:

- Prawdziwie pozytywne (*true positives*, TP): Instancje faktycznie defektogenne sklasyfikowane jako defektogenne;
- Fałszywie pozytywne (*false positives*, FP): Instancje faktycznie niedefektogenne, sklasyfikowane jako defektogenne;
- Prawdziwie negatywne (*true negatives*, TN): Instancje faktycznie niedefektogenne, sklasyfikowane jako niedefektogenne;
- Fałszywie negatywne (*false negatives*, FN): Instancje faktycznie defektogenne, sklasyfikowane jako niedefektogenne.

Na podstawie tych wyników możemy zdefiniować następujące miary, które są najczęściej stosowane w literaturze dotyczącej predykcji defektów.

1. Wynik fałszywie dodatni (ang. *false positive rate*, *FPR*):

$$FPR = \frac{FP}{TN + FP}. \quad (2.1)$$

FPR znany jest także jako prawdopodobieństwo wystąpienia fałszywych alarmów (FP) [134]. FPR określa jak często instancje pozbawione defektów są błędnie prognozowane jako defektogenne.

2. Dokładność (ang. *accuracy*, *ACC*):

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}. \quad (2.2)$$

Dokładność uwzględnia zarówno rezultaty prawdziwie pozytywne jak i prawdziwie negatywne, stawiając je nad pozostałymi rezultatami. Innymi słowy, dokładność przedstawia odsetek poprawnie sklasyfikowanych instancji. Jednakże, dokładność nie jest odpowiednią metodą do mierzenia predykcji defektów oprogramowania, z powodu występowania tamże zjawiska niebalansowania klas (opisanego szerzej w punkcie 2.9.1).

Dlatego też dokładność nie jest rekomendowana do tego typu badań, ponieważ mimo wykazywanej wysokiej dokładności, model nie musi poprawnie identyfikować tego, na czym nam zależy – instancji defektogennych [172].

3. *Precyzja* (ang. *precision*, *Prec*):

$$Prec = \frac{TP}{TP + FP}. \quad (2.3)$$

Precyzja zwana jest również pozytywną wartością predykcyjną (ang. *Positive Predictive Value* – PPV) [134]; jej wynikiem jest stosunek elementów oprogramowania sklasyfikowanych jako defektogenne, które faktycznie były defektogenne (wyniki prawdziwie pozytywne), do sumy wszystkich elementów sklasyfikowanych jako defektogenne (fałszywie oraz prawdziwie).

4. *Czułość* (ang. *sensitivity* lub *recall*, *Rec*):

$$Rec = \frac{TP}{TP + FN}. \quad (2.4)$$

Czułość określa się także jako prawdopodobieństwo wykrycia (ang. *probability of detection* – PD) lub współczynnik TPR (ang. *True Positive Rate*) [134]. Miara ta uwzględnia wszystkie przypadki sklasyfikowania modułów oprogramowania faktycznie defektogennych. Wynikiem *Czułości* jest stosunek elementów oprogramowania sklasyfikowanych jako defektogenne, które faktycznie były defektogenne (wyniki prawdziwie pozytywne), do sumy wszystkich sklasyfikowanych elementów, które w rzeczywistości są defektogenne. Miara ta jest szczególnie przydatna w zastosowaniach komercyjnych klasyfikatorów i została użyta w rozdziale 4 niniejszej pracy, traktującym o efektywności kosztowej predykcji defektów oprogramowania.

5. *Swoistość* (ang. *specifity* lub *selectivity*):

$$Specifity = \frac{TN}{TN + FP}. \quad (2.5)$$

Swoistość określa się także jako współczynnik TNR (ang. *True Negative Rate*) [41]. Miara ta uwzględnia wszystkie przypadki sklasyfikowania modułów oprogramowania faktycznie niedefektogennych. Wynikiem *Czułości* jest stosunek elementów oprogramowania sklasyfikowanych jako niedefektogenne, które faktycznie **nie** były defektogenne (wyniki prawdziwie negatywne), do sumy wszystkich sklasyfikowanych elementów, które w rzeczywistości są niedefektogenne. Miara ta jest niejako uzupełnieniem *czułości* – opisuje ona zdolność modelu predykcji do prawidłowego sklasyfikowania tych elementów, które nie są defektogenne.

6. *Miara F* (ang. *F-measure*):

$$F = \frac{2 \times (Prec \times Rec)}{Prec + Rec}. \quad (2.6)$$

Miara F to średnia harmoniczna precyzji i czułości [200]. Jest to popularna miara, wykorzystywana w wielu badaniach związanych nie tylko z predykcją defektów oprogramowania [99, 109, 150, 171, 219], ale jest szeroko stosowana do oceny klasyfikatorów, w tym najnowszych badaniach dotyczących zastosowania uczenia maszynowego w diagnostyce COVID-19 [211]. Jednakowoż, w swojej wielokrotnie cytowanej pracy David Powers zwrócił uwagę, że miara F może być nieodpowiednia dla danych wykazujących niezrównoważenie klas, ponieważ nie uwzględnia wyników prawdziwie negatywnych; podczas gdy miary wykorzystujące korelację (takie jak współczynnik korelacji Matthews – patrz niżej) pozostają symetryczne, oceniając oba kierunki predykcji, tj. klasyfikator przewidujący rzeczywistą klasyfikację i rzeczywista klasyfikacja „przewidująca” predykcję klasyfikatora [167]. Dodatkowo, Chicco i Jurman w swoich badaniach wprost udowodnili, że miara F przenosi mniej informacji o poprawności klasyfikacji binarnej niż współczynnik korelacji Matthews [29, 30].

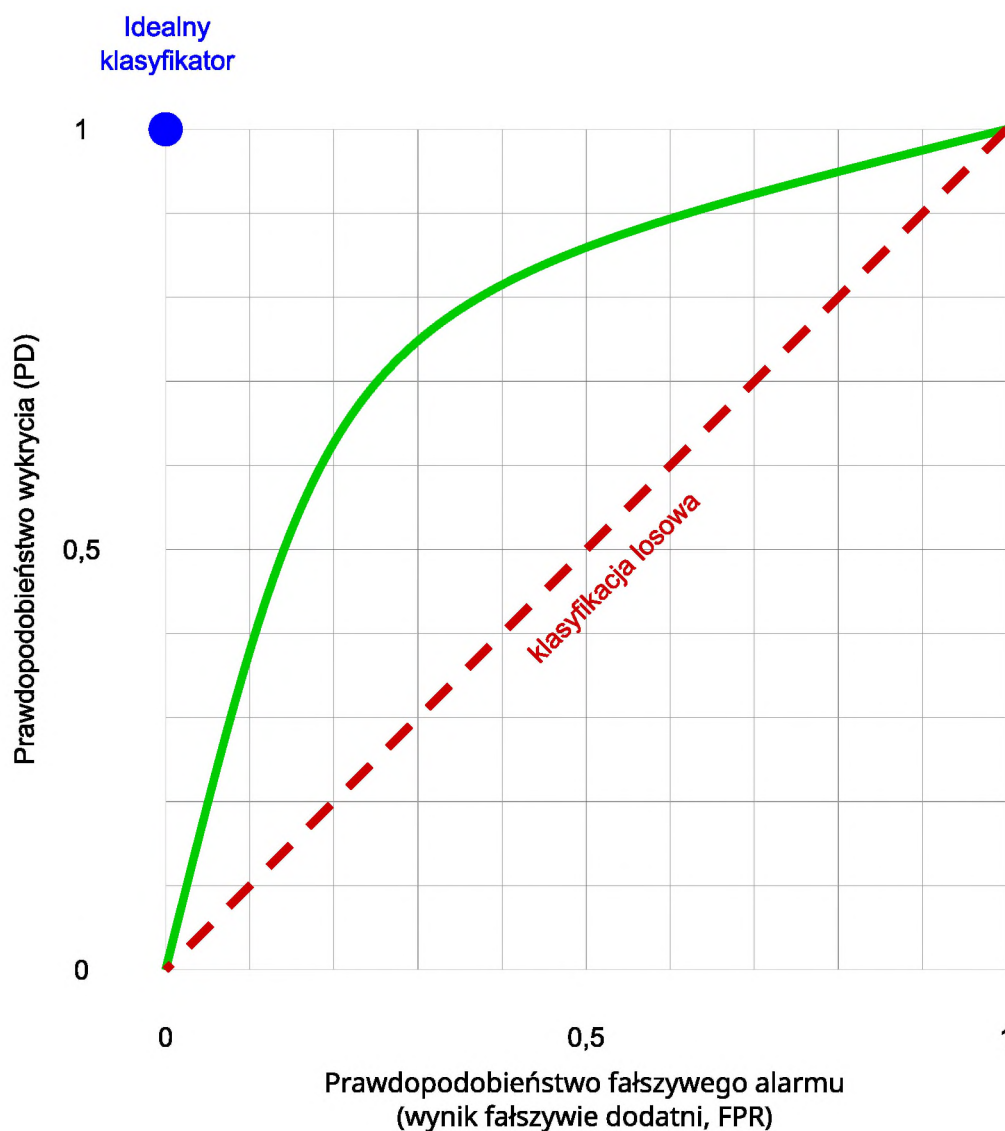
7. *Współczynnik korelacji Matthews* (ang. *Matthews Correlation Coefficient, MCC*), mimo, że zaproponowany przez Matthews do oceny diagnostyki nowotworów już w 1975 roku [130], został użyty w predykcji defektów przez Shepperd’a et al dopiero w 2014 [186]. Mierzy rozbieżność (korelację) pomiędzy zaobserwowaną a przewidzianą klasyfikacją bazując na **wszystkich czterech** kwadrantach macierzy pomyłek, pozwalając opisać całą macierz za pomocą jednej liczby:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \quad (2.7)$$

Miarę tę można stosować nawet wtedy, gdy występuje silne niezbalansowanie klas. Współczynnik zwraca wartość z przedziału od -1 do +1. Wartość +1 oznacza predykcję idealną, 0 nie lepsze niż losowe zgadywanie, a -1 oznacza całkowitą niezgodność między predykcją a obserwacją. Współczynnik Korelacji Matthews, jako pozbawiony wad charakterystycznych dla np. Miary F, uważany jest za jedną z najlepszych miar dla klasyfikacji binarnej [167, 29, 30].

8. AUC (ang. *area under curve*) mierzy pole obszaru zakreślonego krzywą ROC (ang. *receiver operating characteristic* - krzywa charakterystyczno-operacyjna odbiornika¹). Krzywą ROC wykreślają wcześniej wymienione miary prawdopodobieństwa wykrycia (PD) i prawdopodobieństwa fałszywych alarmów (wyniku fałszywie dodatniego, FPR).

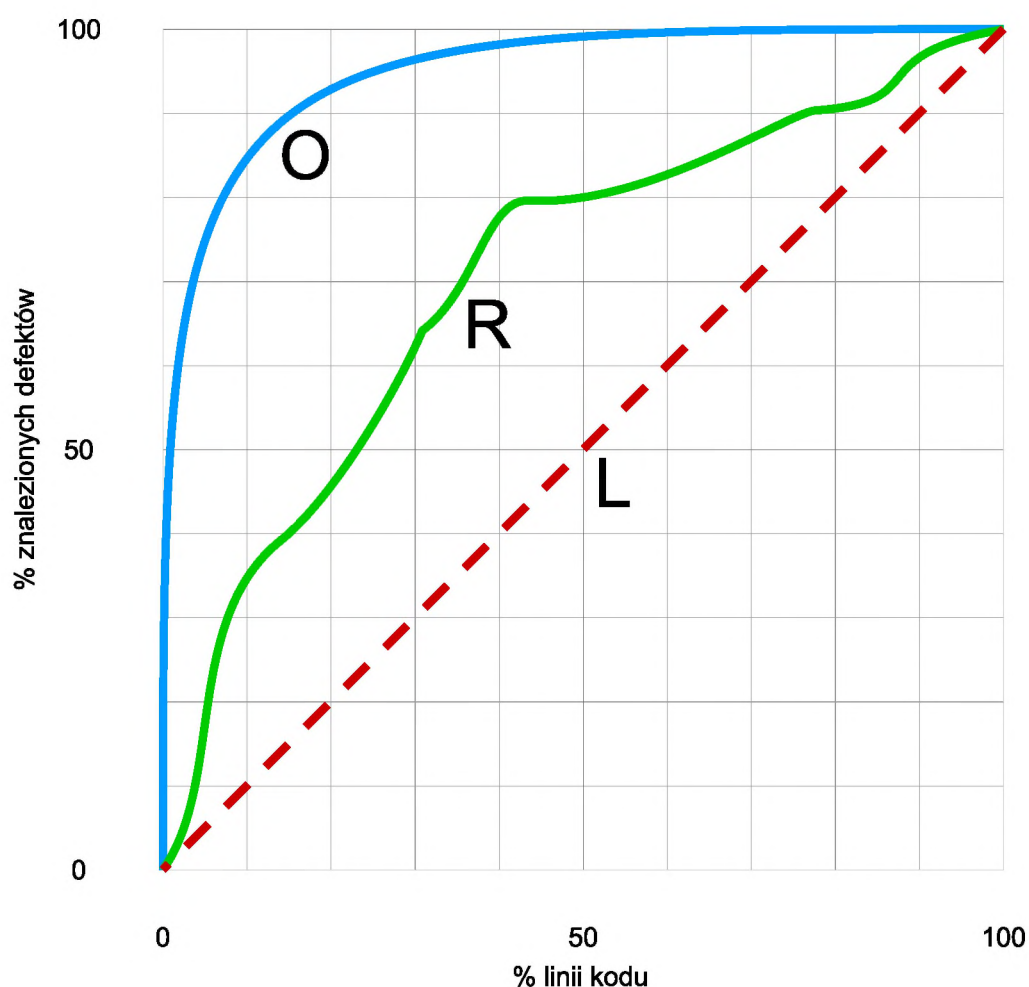
¹ Ma to związek z pierwotnym zastosowaniem krzywej ROC, wykorzystywanej do oceny sprawności pierwszych radarów.



Rysunek 2.1: Przykład krzywej ROC.

Rysunek 2.1 przedstawia typową krzywą ROC. Wartości FPR i PD są zależne od progu predykcji każdej sklasyfikowanej instancji. W miarę jak model predykcji staje się lepszy, krzywa ma tendencję do zbliżania się do punktu $(FPR, PD) = (0, 1)$. Innymi słowy, idealny model predykcji defektów powinien mieć wartość $AUC = 1$. Dla klasyfikacji czysto losowej krzywa będzie przypominać prostą linię przebiegającą od punktu (0,0) do punktu (1,1) [134, 172]. AUC o wartości 0,5 określa się jako predykcja losowa [172]. Miary takie jak precyzja czy czułość mogą się różnić w zależności od przyjętego progu predykcji. Miara AUC określa efektywność predykcji bez względu na wartości progowe, dzięki czemu stanowi stabilną miarę, którą można wykorzystywać do porównywania różnych modeli predykcji defektów oprogramowania.

9. AUCEC – powierzchnia pod krzywą efektywności kosztowej (ang. *area under cost-effectiveness curve*) jest kolejną metodą pomiaru predykcji defektów oprogramowania, wykorzystywaną przy założeniu, że w konsekwencji przeprowadzenia predykcji defektów pewna liczba linii kodu (LOC) zostanie sprawdzona przez testerów bądź programistów. Efektywność kosztowa (opłacalność) predykcji defektów była badana m.in. przez Arisholma et al [7]. Ilustruje ona jak wiele defektów można znaleźć w n% LOC badanego lub testowanego kodu. Innymi słowy, jeżeli dany model pomaga znaleźć więcej defektów przy mniejszym nakładzie pracy na testy w porównaniu do innych algorytmów, to można wtedy powiedzieć, że jego efektywność kosztowa jest wyższa.



Rysunek 2.2: Przykładowe krzywe opłacalności, opisujące modele predykcji: optymalny (O), rzeczywisty (R) i losowy (L).

Rysunek 2.2 przedstawia przykładowe krzywe opłacalności przytaczane przez badaczy [173]. Oś X określa procent przetestowanych linii kodu, podczas gdy oś Y przedstawia procent znalezionych defektów [173]. Wykres przedstawia trzy łączące się ze sobą krzywe. Załóżmy, że krzywe O, R i L określają kolejno optymalny, rzeczywisty i losowy model predykcji. Jeżeli weźmie się pod uwagę obszar pod krzywą (AUCEC), to wtedy można będzie zauważyć, że model optymalny posiada największą wartość AUCEC spośród wszystkich przedstawionych. W przypadku krzywej opisującej model losowy, AUCEC wynosi połowę całkowitego obszaru wykresu ($AUCEC = 0,5$). Wysoka wartość AUCEC dla modelu optymalnego oznacza, że można znaleźć znacznie więcej defektów przy przetestowaniu mniejszej liczby LOC niż w przypadku gorszych modeli.

2.2.2. Miary regresji

Do pomiarów modeli predykcji defektów wykorzystujących regresję wykorzystuje się często miary opisujące korelację między faktyczną a przewidywaną liczbą defektów w badanym kodzie [10, 148, 189, 226]. Reprezentatywne miary to: korelacja Spearmana, korelacja Pearsona, miara R^2 i ich pochodne [10, 148, 189, 226]. Miary te są wykorzystywane w badaniach zależności między wybranymi metrykami a liczbą defektów [226].

2.2.3. Odpowiedni wybór miar

Miara F jest często wykorzystywana w badaniach nad predykcją defektów [99, 109, 150, 163, 172, 188]. Porównywanie (ocena) modeli za pomocą miar precyzji i czułości nie jest proste, ponieważ poszczególne modele mogą wykazywać się wysoką precyzją ale niską czułością i *vice versa*. Jako że miara F jest średnią harmoniczną precyzji i czułości, a także pozwala ocenić model za pomocą pojedynczej wartości, może to tłumaczyć dużą popularność tej miary.

Jednakże wartość miary F może różnić się w zależności od tego, jaki przyjęty został próg prawdopodobieństwa predykcji danego fragmentu kodu źródłowego. Jeżeli model przewiduje, że dana instancja zawiera defekty, oznacza to, że określone przez model prawdopodobieństwo wystąpienia defektów osiągnęło przyjęty wcześniej próg. Lessmann et al. zwrócili uwagę na to, że we wnioskach przedstawionych w literaturze związanej z tym tematem pomijano aspekt progów (kiedy już możemy uznać instancję na potencjalnie defektogenną, a kiedy nie?) co prowadziło do niejednorodnych rezultatów [110].

W celu poradzenia sobie z ograniczeniami miary F, badacze wykorzystują inne miary – takie jak AUC i AUCEC, które nie są zależne od progów [59, 110, 172, 195]. Rahmanna et al. przekonują w swoich pracach, że AUCEC jest miarą dobrą do ewaluacji modeli predykcji z praktycznego punktu widzenia [171, 172, 173].

Z kolei autorzy tacy jak Powers czy Chicco et al. twierdzą, że optymalną miarą klasyfikacji, w tym predykcji defektów oprogramowania, jest współczynnik korelacji Matthews, jako jedyny, który uwzględnia wszystkie elementy macierzy pomyłek, jednocześnie dając jedną wartość opisującą efektywność modelu predykcyjnego [167, 29, 30]..

2.3. Metryki wykorzystywane w predykcji defektów

Metryki grają kluczową rolę w procesie predykcji defektów oprogramowania. Metryki te można podzielić na dwie kategorie: metryki kodu i metryki procesowe. Metryki kodu są zbierane bezpośrednio z kodu źródłowego analizowanego programu, podczas gdy źródłem metryk procesowych są zbiory informacji o technicznej historii rozwijanego oprogramowania, takie jak repozytoria kodu, systemy kontroli wersji czy systemy śledzenia defektów.

2.3.1. Metryki kodu

Metryki kodu, zwane także metrykami produktowymi, określają złożoność kodu źródłowego. U podstaw stosowania metryk kodu leży założenie, że im bardziej złożony jest kod źródłowy, tym bardziej podatniejszy jest na defekty. Tym samym, w celu określenia poziomu złożoności kodu źródłowego, w literaturze przedmiotu zaproponowano wiele metryk kodu.

Metryki rozmiaru określają „objętość, długość, ilość i ogólną wielkość produktów oprogramowania” [32]. Typowym przedstawicielem metryki rozmiaru jest liczba linii kodu (LOC). Według najlepszej wiedzy autora model zaproponowany przez Akiyame był pierwszym badaniem wiążącym LOC z występowaniem defektów w oprogramowaniu [2]. Później LOC było wykorzystane jeszcze wielokrotnie do predykcji defektów [39, 71, 109, 134, 188, 195]. Jedną z szerzej stosowanych metryk rozmiaru jest *liczba wykonywalnych linii kodu*, która różni się od LOC tym, że zlicza jedynie te linie kodu, które realizują operacje programu. Jeśli występuje duża różnica pomiędzy metryką liczby linii kodu oraz metryką liczby wykonywalnych linii kodu może to oznaczać, że w kodzie źródłowym występuje dużo linii rozpoczynających się znakiem oznaczającym komentarz. Linie te nie muszą mieć wartości informacyjnej, a być jedynie prowizorycznie „wyłączonymi” instrukcjami programu. Zatem porównanie obu metryk może służyć ocenie kodu źródłowego pod kątem łatwości jego utrzymania.

Maurice Howard Halstead w 1977 roku zaproponował szereg metryk opisujących rozmiar kodu, wykorzystującą liczbę operatorów i operandów [68] w kodzie. Metryki Halsteada to np. długość programu, rozmiar słownika programu a nawet szacowany trud czy wysiłek wymagany do napisania programu. Sharma i Spinellis w swoich badaniach z 2020 wskazują na znikomą popularność tych metryk wśród programistów [184], ponieważ metryki te są od wielu lat krytykowane przez badaczy za to, że reguły obliczeniowe stosowane w metrykach Halsteada

są arbitralne i wykorzystują „magiczne liczby” bez dostarczenia wystarczających podstaw teoretycznych. Shepperd i Ince jako przykład przedstawiają tu metrykę czasu programowania Halsteada T [187]:

$$T = \frac{E}{f \times s}, \quad (2.8)$$

gdzie E jest określone jako wysiłek programistyczny, f jako stosunek sekund do minut = 60, a s to liczba Strounda, czyli liczba podejmowanych przez programistę decyzji na sekundę (sic!), zależna od danego programisty i typowo przyjmująca wartości od 5 do 25. Ponadto, metryki te wykorzystują tokeny oraz liczbę operatorów i operandów, które są uważane za zbyt uproszczone i prymitywne, ponieważ nie wychwytyją kontroli, danych i struktury mierzonego modułu programu. Wreszcie, Shepperd i Ince zwrócili uwagę, że metryki Halsteada zostały opracowane w erze przetwarzania wsadowego, gdzie programy komputerowe miały znacznie mniejsze rozmiary niż dzisiaj - często wynoszącą jedynie kilkaset linii kodu [187].

Mimo krytyki, Metryki Halsteada wciąż są spotykane w publikacjach, w tym wykorzystywane w badaniach nad predykcją defektów [134, 195, 207]. Co więcej, metryka rozmiaru Halsteada (ang. *Halstead Volume metric*, V) obliczana wzorem:

$$V = N \times \log_2 \eta, \quad (2.9)$$

gdzie N to suma wszystkich występujących operatorów i operandów, a η to liczba rodzajów operatorów i operandów występujących w badanym kodzie, jest wykorzystywana jako element składowy *współczynnika utrzymywalności kodu* obliczanego przez narzędzia firmy Microsoft.

Thomas J. McCabe Senior w 1976 zaproponował metrykę *złożoności cyklomatycznej* (ang. *cyclomatic complexity*, CC) w celu określenia złożoności oprogramowania [131]. Złożoność cyklomatyczna wynika z liczby węzłów, gałęzi i połączonych komponentów w grafach przepływu sterowania kodu źródłowego. Metryka ta reprezentuje jak skomplikowane są ścieżki sterowania. Jako że metryka McCabe’a określa złożoność struktury kodu źródłowego, to różni się zasadniczo od innych metryk rozmiarowych, w tym metryki Halsteada, które mierzą rozmiar i liczbę elementów kodu źródłowego. Ohlsson i Alberg jako jedni z pierwszych użyli metryki McCabe’a do predykcji defektów w telekomunikacji [153]. Wielu innych badaczy również wykorzystało metrykę złożoności cyklomatycznej McCabe’a do zbudowania własnych modeli predykcji [110, 134, 143, 150, 195, 207]. Metryka ta także jest krytykowana – Sharma i Spinellis zwracają uwagę, że złożoność cyklomatyczna nie reprezentuje poprawnie złożoności fragmentu kodu (na przykład w obecności instrukcji switch-case) [184]. W konsekwencji, platforma SonarQube, szeroko stosowana do mierzenia i śledzenia metryk jakości kodu oraz długu technicznego, wprowadziła niedawno nową metrykę, tj. *złożoność poznawczą* (ang. *cognitive complexity*), jako alternatywę dla metryki złożoności cyklomatycznej [53].

Firma Microsoft w swoich narzędziach programistycznych stosuje metrykę kodu zwaną *współczynnikiem utrzymywalności kodu* (ang. *maintainability index, Mi*), która wskazuje względną łatwość utrzymania kodu. Metryka ta wykorzystuje zarówno złożoność cyklomatyczną McCabe’a jak i metrykę rozmiaru Halestada i jest obliczana według wzoru [137]:

$$Mi = \frac{171 - 5,2 \times \ln(V) - 0,23 \times CC - 16,2 \times \ln(LOC) \times 100}{171} \quad (2.10)$$

Im wyższa wartość metryki Mi , tym powinno się oczekiwać kodu łatwiejszego w pielęgnacji. W narzędziach Microsoft obliczona wartość metryki jest normalizowana do zakresu od 0 do 100. Wyświetlana wartość od 20 do 100 oznacza, że kod jest łatwy w utrzymaniu. Wartości od 10 do 19 wskazują, że kod jest umiarkowanie łatwy w utrzymaniu. Na niską utrzymywalność kodu wskazują wartości pomiędzy 0 a 9.

Jako że programowanie obiektowe jest szczególnie popularne, pojawiły się metryki kodu *scrict* dla obiektowych języków programowania. W dużej mierze zostały one zaproponowane, aby usprawnić procesy wytwarzania oprogramowania obiektowego. Przykładowymi metrykami dla programów zorientowanych obiektowo (ang. *Object Oriented Metrics; OO*) jest grupa metryk Chidamera i Kemerera (CK) [31]. Zostały one skonstruowane na podstawie cech charakterystycznych języków zorientowanych obiektowo takich jak dziedziczenie, sprzężenie czy kohezja [31]. Basili et al. swoim badaniem potwierdzili, że jest możliwe stworzenie modelu predykcji wystąpienia defektów przy wykorzystaniu metryk CK [12]. Później w wielu badaniach wykorzystywano metryki CK do zbudowania własnych modeli predykcji [10, 39, 90, 99, 109, 159, 219].

Brak spójności metod (ang. *Lack of Cohesion of Methods, LCOM* lub krócej *Lack of Cohesion*) metryka pokazująca, czy klasa realizuje jedną funkcjonalność (niska wartość metryki), czy też wykonuje ich kilka (wysoka wartość metryki), co określa się jako niską spójność badanego elementu kodu.

Sprzężenie klas (ang. *class coupling, CC*) jest relatywnie często używaną metryką z grupy CK, która mierzy sprzężenie pomiędzy unikalnymi klasami poprzez parametry, zmienne lokalne, zwracane typy, wywołania metod, klasy bazowe czy implementacje interfejsów. Dobra praktyka projektowania oprogramowania wymaga, aby typy i metody miały wysoką spójność i niskie sprzężenie. Wysokie sprzężenie klasy wskazuje, że została ona napisana tak, że trudno będzie ją utrzymywać oraz ponownie użyć w kolejnych wersjach oprogramowania z powodu wielu współzależności od innych elementów programu.

Głębokość drzewa dziedziczenia (ang. *depth of inheritance tree, DIT*) jest kolejną popularną metryką kodu z grupy CK. Podaje ona liczbę klas języka obiektowego, które dziedziczą po sobie, aż do klasy bazowej. Głębokość drzewa dziedziczenia jest podobna do sprzężenia klas

w tym, że zmiana w klasie bazowej może wpłynąć na każdą z jej odziedziczonych klas. Im wyższa jest ta liczba, tym głębsze jest dziedziczenie i tym większe ryzyko, że modyfikacja klasy bazowej spowoduje nieprzewidziane konsekwencje i defekt oprogramowania. Zwykle pożądaną jest jak najniższa wartość tej metryki.

Oprócz metryk CK, zaproponowano szereg innych metryk zorientowanych obiektowo i bazujących na rozmiarze kodu źródłowego [40]. Metryki rozmiaru w przypadku programowania obiektowego zliczają np. liczbę klas, metod itp. Także i w tym przypadku wykorzystano metryki zorientowane obiektowo do budowy modeli predykcji [38, 99, 109, 159, 219, 223, 226].

2.3.2. Metryki procesowe

W tym punkcie opisane zostaną popularne metryki drugiego rodzaju – metryki procesowe.

Spiętrzenie zmian w kodzie (ang. *relative code change churn*) – Nagapann i Ball [147] zaproponowali osiem metryk, które mierzą zmiany, które zostały wprowadzone w kodzie źródłowym [147]. Przykładowo, metryka nazwana „M1” jest obliczana poprzez podzielenie zmienionych linii kodu (łączna liczba usuniętych i dodanych linii kodu w porównaniu do bazowej wersji programu) przez łączną liczbę LOC [147]. Pozostałe zaproponowane metryki biorą pod uwagę różne, znormalizowane zmiany, takie jak liczbę usuniętych LOC podzieloną przez wszystkie LOC, pliki modyfikowane (łączna liczba plików w komponencie, których struktura została zmieniona) podzielone przez łączną liczbę plików i tak dalej [147]. W studium przypadku przeprowadzonym przez Nagapann’a i Ball’a, opisane tam metryki okazały się być dobrym czynnikiem prognostycznym, mogącym wyjaśnić zależność między zagęszczeniem zmian w kodzie źródłowym a podatnością na defekty [147].

Metryka modyfikacji (lub metryka zmian) jest wykorzystywana do obliczania zakresu zmian kodu źródłowego, zapisanych w historii wybranego systemu kontroli wersji. Przykładowo, można wyliczyć liczbę korekt/napraw defektów/refaktoryzacji pliku zawierającego kod oraz liczbę osób odpowiedzialnych za edycję tegoż pliku. Moser et al. [143] wyodrębnili 18 metryk zmian z repozytorium kodu narzędzia Eclipse po to, aby przeprowadzić analizę porównawczą między metrykami kodu oraz metrykami zmian. Niektóre z metryk Mosera et al. [143] uwzględniają także dodane i usunięte linie kodu, podobnie jak to ma miejsce w metryce zliczającej zmiany w kodzie (*relative code change churn*) [147]. Jednakże metryki te nie uwzględniają zależności między łączną liczbą LOC i liczbą plików, lecz operują na średniej i maksymalnej wartości zmian w kodzie. Metryki Moser’a et al. rejestrują także maksymalną i średnią liczbę zmienionych zestawów (plików wysłanych do repozytorium kodu jednorazowo, tj. w jednym „commicie”) oraz wprowadzają tzw. *metrykę wieku* – wiek plików liczony w tygodniach oraz wiek ważony znormalizowany poprzez uwzględnienie LOC) [143]. Moser

et al. przekonywali, że metryki modyfikacji są lepszymi predyktorami defektogenności, niż metryki kodu.

Entropia zmian w kodzie – W swojej pracy z 2009 roku [70], Ahmed E. Hassan wykorzystał koncept entropii informacji Claude E. Shannona [183] do zaobserwowania, jak złożone mogą być zmiany w kodzie proponując *metrykę złożoności historycznej* (HCM – *History Complexity Metric*). Aby potwierdzić poprawność swoich założeń, Hassan zbudował szereg modeli statystycznej regresji liniowej, wykorzystujących metrykę HCM, ale też – osobno – metryki zmian oraz liczbę modyfikacji kodu i liczbę defektów w sześciu projektach otwarto-źródłowych [70]. Ocena wyników wykazała, że modele predykcji bazujące na HCM uzyskały znacznie lepsze wyniki, niż te wykorzystujące metryki zmian. Pomysł na wykorzystanie konceptu entropii informacji w celu mierzenia złożoności zmian jest ciekawy, jednakże porównanie modelu wykorzystującego HCM do modelu wykorzystującego jedynie dwie metryki zmian nie jest wyczerpującym badaniem. Ponadto, ewaluacja modeli została przeprowadzona na stosunkowo wysokim poziomie – całego systemu informatycznego, a nie np. poszczególnych plików.

Śpiętrzenie metryk kodu oraz entropia kodu – D'Ambros et al. porównali metryki wykorzystywane w badaniach nad predykcją defektów. W swoich badaniach nie napotkali oni metryk, które bezpośrednio określałyby entropię kodu oraz wskazywały, za pomocą ilu metryk można ów kod opisać (czyli tzw. spiętrzenie metryk). Dlatego też D'Ambros et al. zaproponowali nowe metryki: entropii kodu (oznaczoną jako *HH* – *historical entrophy*) oraz spiętrzenia metryk (*CHU* – *code metrics churn*) [34]. W porównaniu do metryk zmian bazujących na LOC (zmianie liczby linii kodu), CHU oblicza zmiany w dwutygodniowej bazie przechowującej kombinacje szeregu różnych metryk obiektowych. Ponieważ CHU oblicza zmiany w odstępie dwutygodniowym [34], jest ona w stanie dostarczyć o wiele precyzyjniejszych informacji na temat zakresu zmian niż metryki, które szacują liczbę zmian między bazową i nową korektą [147]. D'Ambros et al. zaproponowali także warianty (modyfikacje) metryk CHU i HH – porównując je, badacze doszli do wniosku, że użycie wariantów WCHU (*weighted churn*) i LDHH (*linear decay historical entrophy*) poskutkowało uzyskaniem wysoce efektywnej predykcji defektów [34].

Ograniczeniem metryk jest olbrzymie zapotrzebowanie na dane i moc obliczeniową, co wynika z tego, że metoda ta wymaga regularnych analiz zmian kodu źródłowego w systemie kontroli wersji [34]. *Popularność*: Bachelii et al. zaproponowali tak zwaną metrykę popularności, bazując na analizie archiwów mailowych producentów oprogramowania. Głównym założeniem tej metody jest to, że najszerszej omawiane przez e-mail artefakty w oprogramowaniu są najbardziej podatne na defekty. Większość z metryk popularności opiera się na zliczeniu, ile razy konkretna klasa jest omawiana w poczcie elektronicznej [10]. Metryki wydobywające informacje z archiwów mailowych, są niewątpliwie ciekawostką, jednak sami

autorzy ocenili, że metoda ta nie jest lepsza od innych metryk bazujących na kodzie czy procesach [10].

Autorstwo: Bird et al. zaproponowali cztery metryki bazujące na autorstwie danego modułu oprogramowania [19]. Badania te rozpoczęły się od pytania "jak bardzo to, kto jest autorem kodu, wpływa na jakość modułu?". Autorstwo modułu określane jest poprzez poziom wkładu własnego w moduł a „mniejszy” i „większy” współautorzy określani są na podstawie tego, czy ich wkład jest mniejszy czy większy niż 5%. Stąd zaproponowano cztery metryki autorstwa modułu: MINOR (Liczba autorów z mniejszym wkładem w moduł), MAJOR (liczba autorów z wkładem większym w moduł), TOTAL (łączna liczba autorów) i OWNERSHIP (procent autorów z największym wkładem w moduł, z liczby wszystkich autorów). Bird et al. postulowali, że im wyższy wskaźnik OWNERSHIP, tym mniejsza podatność na defekty [19].

Rahman et al. zbadali zależność między defektami a liczbą autorów kodu oraz doświadczeniem programistów [170]. Badania te wykazały, iż kontrola jakości powinna skupiać się na tych plikach w kodzie źródłowym, których autorami byli mniej doświadczeni developerzy.

Metryka mikro-interakcji: Lee et al. zaproponowali metrykę mikro-interakcji (MIM – Micro interaction metric), wydobytej z narzędzia Mylyn, który pobiera informacje o interakcji programisty ze środowiskiem Eclipse [109]. Głównym założeniem MIM jest to, że defekty mogą być wywołane błędnymi działaniami programistów. Przykładowo, zaobserwowana większa ilość czasu przeznaczona na edycję danego kodu źródłowego, może iść w parze z podatnością tego kodu na defekty. Lee et al. wydobyli 56 metryk tego rodzaju z Mylyn, a następnie porównali ich wyniki z metrykami kodu i procesowymi [109]. W badaniach tych metryki MIM osiągnęły lepsze wyniki niż metryki kodu i procesowe, zarówno przy użyciu klasyfikacji jak i regresji jako metody predykcji defektów. Jednakże MIM jest mocno zależna od rozszerzenia (pluginu) Eclipse’a jakim jest Mylyn. Dlatego też zastosowanie tego rodzaju metryk może być bardzo utrudnione w kontekstach, które nie wykorzystują narzędzi takich jak Mylyn [109].

2.3.3. Inne metryki

Nie licząc metryk kodu i procesowych, naukowcy zaproponowali nowe metryki bazujące na posiadanej wiedzy, takie jak miernik sieci [132, 164, 226] i metryka antywzorca projektowego [199].

Meneely et al. zebrali metryki programistyczne z programistycznych portali społecznościowych, które przedstawiają strukturę współpracy między programistami w ramach wspólnych repozytoriów kodu źródłowego [132]. Bazując na danych z programistycznych portali, badania te wykazały, że defekty w oprogramowaniu są mocno zależne od topologii sieci współpracy programistów [132].

Pinzger et al. także skonstruowali sieć ilustrującą zależności między modułami kodu źródłowego a tworzącymi go programistami [164]. Sieć ta reprezentuje w jaki sposób każdy z programistów wpłynął na każdy z modułów. Nazwana została ona siecią udziałów (ang. *contribution network*) i – według badaczy – jest ona w stanie w znacznym stopniu przewidzieć defekty w oprogramowaniu po jego wypuszczeniu na rynek [164].

Zimmermann et al. skonstruowali grafy wewnętrznej zależności kodu binarnego, a następnie przeprowadzili analizę sieci bazującej na tych grafach. Na tej podstawie zbudowali modele predykcji i porównali je z modelami bazującymi na metrykach kodu i procesowych [226]. Według nich, ocena sieci jest w stanie wskazać znacznie większą liczbę fragmentów kodu binarnego podatnych na defekty niż metryki procesowe czy metryki kodu.

Taba et al. [199] zaproponowali cztery metryki antywzorców projektowych. Antywzorce to w zasadzie słabo zaprojektowane oprogramowanie, więc można założyć większe prawdopodobieństwo wystąpienia defektów w kodzie źródłowym wraz z zaobserwowaniem zwiększenia się wartości metryk antywzorców.

Przeprowadziwszy ocenę metryki w dwóch projektach open-source, predykcja defektów wykorzystująca metryki antywzorców projektowych wykazała się wysoką wartością miary F [199].

2.3.4. Wybór odpowiedniego rodzaju metryk

Jako że metryki kodu, na przykład rozmiarową (LOC), złożoności cyklomatycznej McCabe’a czy Halsteada wykorzystuje się już od lat 70. XX wieku, to liczba badań przy wykorzystaniu tego rodzaju metryk jest znacznie większa niż w przypadku metryk procesowych. Większość modeli wykorzystujących metryki procesowe została opisana w latach dwutysięcznych, kiedy repozytoria oprogramowania wykorzystujące systemy kontroli wersji, takie jak SVN czy GIT, stały się popularne. Nieustającym tematem dyskusji w świecie inżynierii oprogramowania jest pytanie, czy użycie metryk kodu jest najbardziej efektywne w kontekście predykcji oprogramowania, czy też metryki procesowe są od nich lepsze. Na przykład, Menzies et al. przekonywali, że metryki kodu są wysoce użyteczne do budowy efektywnych modeli predykcji [134], tymczasem przeprowadzone przez Rahman’a et al. badania zarówno metryk kodu jak procesowych wskazywały na to, że metryki kodu są mniej skuteczne niż metryki procesowe [171].

2.4. Modele predykcji defektów

Większość modeli predykcji defektów bazuje na uczeniu maszynowym. W zależności od tego, co mają szacować (np. podatność na defekty, liczbę defektów itp.), modele bazujące na uczeniu maszynowym dzielone są na dwa zasadnicze typy: modele klasyfikacji i modele

regresji. Jako że proponuje się coraz to nowsze metody uczenia maszynowego, bada się również przydatność tychże metod w budowaniu lepszych modeli predykcji [112, 116].

Nie licząc modeli uczenia maszynowego, można doszukać się badań, gdzie proponuje się odmienne podejście do predykcji defektów, takie jak np. wykorzystanie algorytmu defectCache [101, 43, 111, 173].

Klasyfikacja i regresja charakteryzują się podobnym procesem predykcji, ponieważ bazują one na procesie uczenia maszynowego. Różnicuje je to, co tak właściwie mają określać. Modele klasyfikacyjne zwykle definiują podatność na defekty [134, 143, 227], podczas gdy algorytmy regresji szacują liczbę defektów [147, 146, 226]. Odpowiedzią na pytanie „który model powinien być wykorzystany przez zespoły zajmujące się zapewnianiem jakości?”, jest zależna od tego, jakie jest jego przeznaczenie.

Regresja logistyczna jest najczęściej wykorzystywana w badaniach tego zjawiska. Drzewa decyzyjne i naiwny klasyfikator Bayesa równie często są wykorzystywane w badaniach. W kontekście uczenia się maszynowego w modelach regresyjnych, regresja liniowa i ujemna regresja dwumianowa były najczęściej wykorzystywane w literaturze [19, 34, 90, 109, 157, 189, 214].

W ramach nowych metod uczenia maszynowego zaproponowano między innymi aktywne uczenie i uczenie częściowo nadzorowane. Lu i Cukic [116], zaproponowali model przewidujący wystąpienie defektów bazujący na uczeniu aktywnym, gdzie próbki instancji są wybierane, a następnie klasyfikowane przez wyrocznie (ludzkich profesjonalistów) pod kątem tego, czy dane instancje nadają się do zestawu treningowego (uczącego).

Li et al. zaproponowali metody CoFest i ACoFest [116]. CoFest to metoda próbkowania bazująca na algorytmie częściowo nadzorowanego uczenia się poprzez wielokrotną ewaluację procesu predykcji wykorzystującej algorytmy lasu losowego. ACoFest to rozszerzona wersja CoFest wykorzystująca uczenie aktywne [116].

Kim et al. zaproponowali narzędzie BugCache, które nadaje wysoki priorytet obszarom kodu, które wcześniej zostały zapamiętane jako defektogenne. Podczas ewaluacji modelu na siedmiu projektach open-source, 10% plików zawierało od 73% do 95% wszystkich defektów [101]. BugCache udostępnia informacje o lokalizacji, takie jak chwilowa i przestrzenna lokalizacja (jeżeli defekt w module został wprowadzony niedawno lub jeżeli dany moduł został zmieniony w innych instancjach oprogramowania – np. w innej wersji, to wtedy prawdopodobieństwo że pojawią się w nich defekty, jest wyższe [101]).

2.5. Dokładność predykcji

Opisywane w literaturze modele predykcji wystąpienia defektów posiadają różne poziomy dokładności, takie jak: podsystem [46], [70, 103], komponent/paczka [110, 147, 226],

plik/klasa [134, 126, 150, 157, 227], metoda [59, 71] i – finalnie – atomowe zmiany w kodzie [98].

Badania przeprowadzone przez Hata et al. [71] wykorzystywały model predykcji z dokładnością na poziomie metody (składowej klasy kodu obiektowego). Autorzy wykazali, że algorytmy opierające się na tej metodzie charakteryzują się większą efektywnością kosztową niż inne metody z niższą dokładnością (na poziomie plików czy paczek).

Kim et al. zaproponowali model predykcji defektów nazwany klasyfikacją zmian (change classification) [98]. W założeniu rozwiązanie to miało być bezpośrednim wsparciem dla programistów, jako że (w warunkach laboratoryjnych) model ten może dostarczyć natychmiastowe wyniki predykcji za każdym razem, gdy programista prześle zmiany w kodzie źródłowym do systemu kontroli wersji (dokładność predykcji na poziomie pojedynczej zmiany). Jednak wykorzystanie tego rozwiązania w praktyce może być bardzo trudne, gdyż w procesie budowania modelu, aby był on efektywny, wykorzystano informacje o setkach (przeciętnie 500-700) dobrze udokumentowanych zmian w kodzie wieloletnich projektów otwarto-źródłowych [98].

Aby wykorzystać model klasyfikacji zmian w praktyce, musimy wziąć pod uwagę efektywność kosztową [172], [173]. Jedną z metod na zwiększenie efektywności kosztowej modelu predykcji jest szacowanie możliwości wystąpienia defektu na najniższych poziomach oprogramowania [71]. W związku z tym, badacze powinni poświęcić więcej uwagi zarówno na predykcję defektów, jak i na ocenę (klasyfikację) zmian na poziomie pojedynczych linii kodu.

2.6. Wstępna obróbka zbiorów danych

Wstępna obróbka danych jest szeroko stosowana w uczeniu maszynowym. Ponieważ większość badań nad predykcją defektów bazuje na tym podejściu, powstał szereg opracowań, które w dużym stopniu skupiały się na wstępnej obróbce danych [99, 134, 150, 190, 219]. W zależności od badań, techniki wstępnej obróbki danych różnią się, ponieważ prace badawcze przeprowadza się przy wykorzystaniu różnych metryk, modeli i podmiotów.

2.6.1. Normalizacja

Normalizacja to obecnie standardowa technika zmierzająca do uzyskania optymalnego, w danych okolicznościach, stopnia uporządkowania danych wejściowych, co pozwala na zwiększenie wydajności modeli klasyfikacyjnych [69, 62].

Menzies et al. proponują wykorzystanie filtra logarytmicznego (ang. *log-filter*) w celu znormalizowania wartości dla metryk o gwałtownym przyroście [134]. Inne eksperymenty prowadzone na tych samych obiektach badań, które wykorzystywał Menzies również wykorzystywały filtr logarytmiczny [134, 207, 208].

Nam et al. zauważyli, że wyniki predykcji międzyprojektowej (patrz rozdział 2.7) są zależne od wykorzystanych technik normalizacji [150]. Zdefiniowali więc oni zasady określające dobór odpowiednich metod normalizacji, takich jak normalizacja minimalna i maksymalna, z-score i wariacje z-score, co pozwala na poprawę wyników w predykcji międzyprojektowej.

2.6.2. Wybór cech

Shivaji et al. zwrócili uwagę na to, że słabe wyniki modeli predykcji mogą być powodowane zbyt dużą liczbą metryk wykorzystywanych do zbudowania modelu [190]. Z tego powodu Shivaji et al. zaproponowali technikę wyboru cech, która może poprawić wydajność klasyfikacji zmian [190]. Turhan et al. także zastosowali tak zwaną selekcję podzbioru przy wykorzystaniu informacji zdobytych przed przeprowadzeniem procesu predykcji wystąpienia defektów [207].

Jako że zbiory danych zawierające informacje wykorzystywane w predykcji defektów mogą cierpieć z powodu współliniowości, badacze często stosują analizę głównych składowych (ang. *PCA - Principal Component Analysis*) w celu zredukowania wymiaru i/lub przekształcenia danych w zbiór cech niewspółliniowych [34, 147, 226].

Gdy zestaw typów danych jest relatywnie mały, proces wyboru cech może bazować na różnych złożonych algorytmach znajdujących wszelkie możliwe kombinacje dostępnych rodzajów danych. Dla każdej kombinacji wskazanej przez taki algorytm tworzony jest model uczenia maszynowego i przeprowadzana jest jego walidacja. Następnie określana jest wydajność modelu – np. jeśli użyto klasyfikatora, określany jest błąd klasyfikacji. Po przetestowaniu wszystkich możliwych kombinacji dostępnych typów (cech) danych wybierany jest podzbiór typów o najmniejszym błędzie klasyfikacji.

Jeśli konieczne jest wykorzystanie zbioru danych o dużej liczbie cech (typów danych), wykorzystanie złożonych algorytmów może być drogie. Pendharkar [162] szacuje, że gdyby algorytm tego typu został zastosowany na zbiorze zawierającym 21 typów danych, to potrzeba by było około 120 lat obliczeń na komputerze bazującym na procesorze RISC 900 MHz, aby wykonać proces wyboru cech.

2.6.3. Naprawa powiązań między zmianami w kodzie a zarejestrowanymi defektami

Ponieważ dane o defektach są zwykle pobierane automatycznie z systemów kontroli wersji oraz śledzenia defektów [100, 193], to często mogą być one obarczone błędami [18]. W celu redukcji tego typu szumów Wu et al. zaproponowali narzędzie ReLink, które automatycznie odnawia powiązania (linki) pomiędzy logami zmian repozytoriów systemów kontroli wersji a identyfikatorami defektów [219]. Kim et al. zaproponował algorytm wykrycia i eliminacji szumów nazwany Closest List Noise Identification (CLNI) [99].

2.7. Predykcja międzyprojektowa

Zdarza się, że projekty wytwarzania oprogramowania, szczególnie te nowopowstałe, nie zawierają wystarczającej liczby danych uczących. Najczęstszym „ratunkiem” w takiej sytuacji pozostaje predykcja międzyprojektowa (inaczej kross-projektowa, mieszana), czyli zbudowanie modelu przy użyciu danych z innego projektu („źródłowego”), a następnie użycie tegoż modelu w projekcie z niewystarczającą ilością danych („docelowego”). Nie jest to łatwe zadanie – przykładowo, Zimmermann et al. przeprowadzili 622 próby predykcji międzyprojektowej i okazało się, że rzeczywiście działało jedynie 3,4% [227]. W celu poprawienia predykcji międzyprojektowej, naukowcy skupili się na badaniach bazujących na uczeniu transferowym i wykonalności predykcji międzyprojektowej.

2.7.1. Uczenie transferowe

Uczenie transferowe to jedna z najaktywniej badanych obecnie sfer w uczeniu maszynowym [160]. Model tradycyjny zakłada, że stosunek danych walidacyjnych do uczących jest taki sam. Innymi słowy, jeżeli ów podział danych zostanie zmieniony, to wtedy cały model musi zostać przebudowany i dostosowany do nowych danych. Jednakże zdobywanie nowych informacji i ich oznakowywanie (nadawanie etykiet) jest procesem kosztownym. Dlatego też niektórzy badacze zajmujący się uczeniem maszynowym skupili się na uczeniu transferowym, umożliwiającym zastosowanie modeli predykcyjnych ze środowisk zawierających wystarczającą liczbę danych uczących w środowiskach z mniejszą ich liczbą [173]. Społeczność inżynierów oprogramowania zaadaptowała idee i techniki uczenia transferowego do predykcji międzyprojektowej.

Kompensacja metryk: Watanabe et al. zastosowali technikę kompensacji metryk w celu dostosowania danych do nowego kontekstu przy wykorzystaniu informacji pobranych z kontekstu (np. projektu) źródłowego [212]. Główną ideą kompensacji metrycznej jest znormalizowanie każdej wartości metryki docelowej używając średnich wartości metryk źródłowych [212]. Posiadając zbiory danych z dwóch projektów, Watanabe et al. przeprowadzili szereg predykcji międzyprojektowych przedstawiając w swojej pracy wyniki opisane takimi wielkościami jak precyzja i czułość [212]. Wyliczono średnią wartość miary F dla predykcji kross-projektowych zarówno z wykorzystaniem, jak i bez wykorzystania kompensacji metryk. Średnia wartość miary F dla predykcji międzyprojektowej korzystającej z kompensacji metryk (0.67) przewyższała wynik dla predykcji bez kompensacji (0.58).

Wąskim gardłem rozwiązania proponowanego przez Watanabe et al. jest niedostateczna ewaluacja zaproponowanego przez nich podejścia. Przeprowadzono jedynie dwie predykcje międzyprojektowe. Zabrakło także badań statystycznych, które mogłyby potwierdzić prawdziwość stawianych przez autorów tez.

Filtr NN: Turhan et al. zastosowali tak zwany filtr najbliższego sąsiada (ang. *nearest neighbour, NN*) w celu zwiększenia wydajności predykcji międzyprojektowych [207]. Głównym założeniem filtru NN jest wykorzystanie instancji źródłowych podobnych do tych znajdujących się w projekcie docelowym w celu zbudowania modelu predykcji. Innymi słowy, można zbudować model korzystający z wybranych instancji klas danych projektu źródłowego, których charakterystyka jest podobna do charakterystyki klas danych znajdujących się w instancjach danych projektu docelowego. W takim wypadku dany model może osiągnąć lepsze wyniki predykcji niż model zbudowany z wykorzystaniem tylko instancji z projektu źródłowego [207]. Na przykład, filtr NN może wybrać 10 instancji danych źródłowych jako najbliższych sąsiadów dla każdej przygotowywanej instancji danych docelowych.

Aby ocenić wyniki predykcji międzyprojektowej, wykorzystującej filtr NN, Turhan et al. przeprowadzili eksperymenty przy wykorzystaniu dziesięciu zbiorów danych podchodzących z organizacji NASA i SOFTLAB [207]. Przeprowadzili oni także test Manna-Whitneya celem potwierdzenia rezultatów. Jak obliczono, średnia wartość miary F dla modeli predykcji międzyprojektowych wykorzystujących filtr NN to 0.36. Jest to wynik lepszy niż w przypadku modeli pozbawionych filtra (0.26).

Transfer naiwnego klasyfikatora Bayesa. W celu polepszenia wyników predykcji międzyprojektowej, Ma et al. zaproponowali technikę transferu naiwnego klasyfikatora Bayesa (ang. *Transfer Naive Bayes, TNB*) [117]. Głównym założeniem TNB jest wyliczenie nowego prawdopodobieństwa w modelu naiwnego klasyfikatora Bayesa poprzez wykorzystanie wartości ważonej instancji źródłowej (wziętej z projektu źródłowego). Waga instancji źródłowej obliczana jest na podstawie podobieństwa między nią a instancją docelową.

Podobieństwo instancji oblicza się na podstawie wartości minimalnej i maksymalnej każdego elementu walidacyjnego zbioru danych. Innymi słowy, podobieństwo jest wyliczane przez liczbę cech instancji źródłowej, których wartości znajdują się pomiędzy minimalną i maksymalną wartością odpowiadających im cech docelowych. Co interesujące, wartość wagi jest wyliczana z wykorzystaniem równań prawa powszechnego ciężenia [117]. Im większa waga instancji uczącej, tym większe jest jej podobieństwo do instancji ze zbioru walidacyjnego. Ostatecznie model TNB jest obliczany z nowymi prawdopodobieństwami przy użyciu wag instancji źródłowych.

Transfer naiwnego klasyfikatora Bayesa doprowadził do lepszych wyników predykcji (0.39) niż filtr NN (0.35), wyrażonych w uśrednionej wartości miary F. Nie bez znaczenia pozostaje jednak fakt, że lepsze wyniki predykcji międzyprojektowej, i to bez użycia NN czy TNB, uzyskali Turhan et al. [207] i Ma et al. [117]. Powodu takiego stanu rzeczy można upatrywać w tym, że Ma et al. nie skorzystali z wyboru cech w swoich badaniach [117].

Największym ograniczeniem badań Ma et al. wydaje się być fakt, że badacze nie przedstawili raportów zawierających rezultaty z wykorzystania predykcji

wewnątrzprojektowej, przez co nie można zweryfikować, czy efektywność predykcji międzyprojektowej wykorzystującej TNB jest porównywalna z efektywnością predykcji wewnątrzprojektowej [117].

Rozrzucony transfer analizy komponentów. Nam et al. zaproponowali dla predykcji międzyprojektowej algorytm, który nazwali TCA+ [150]. TCA+ to rozszerzona wersja transferu analizy komponentów (ang. *Transfer Component Analysis, TCA*), algorytmu uczenia maszynowego zaproponowanego przez Pan'a et al., który służy znalezieniu wspólnej przestrzeni, gdzie dystrybucja źródłowych i docelowych zbiorów danych jest do siebie podobna [160]. Stosując TCA dla predykcji międzyprojektowej Nam et al. [150] zauważyli, że wyniki różnią się w zależności od tego jaki typ normalizacji został zastosowany we wstępnym przetwarzaniu danych. Z tego powodu Nam et al. zaproponowali algorytm TCA+, dodający mechanizm decyzyjny, pozwalający na wybranie odpowiedniego typu normalizacji dla TCA. W badaniach wykorzystano opcje normalizacji takie jak min-max, z-score i wariacje z-score [150]. Wyniki predykcji międzyprojektowej wyrażone miarą F z wykorzystaniem TCA+ dały podobne rezultaty (0.46) jak predykcja wewnątrzprojektowa (0.46).

Międzyprojektowa predykcja defektów bazująca na uczeniu transferowym. Wyżej wymienione podejścia do predykcji międzyprojektowej łączą wykorzystanie zbiorów danych jednego projektu do zbudowania modeli predykcji dla projektu powstającego, lub istniejącego, jednak nie posiadającego danych historycznych. Jednakże każda z przedstawionych powyżej metod możliwa była tylko dla zbiorów danych z takim samym zestawem cech. Jeśli w tym kontekście przyjrzymy się badaniom Nam et al., to liczba projektów wykorzystanych w ich badaniach wynosi 8, a całkowita liczba predykcji tam przeprowadzonych to 26. Jeżeli uwzględnimy wszystkie potencjalne kombinacje międzyprojektowe, to rezultat powinien wynosić $(8 \times (8 - 1) = 56)$. Nie było to jednak możliwe do wykonania, z uwagi na różne zbiory cech danych. W badaniach nad TCA+ zestaw cech trzech zbiorów danych miał rozmiar 26, dla pozostałych pięciu zbiorów wynosił natomiast 61. Dlatego też predykcja międzyprojektowa możliwa była tylko na zbiorach danych z takim samym zestawem cech, czyli sześciu predykcji międzyprojektowych ($3 \times (3 - 1)$) oraz 20 predykcji międzyprojektowych ($5 \times (5 - 1)$). Uzyskanie predykcji międzyprojektowej dla zbiorów danych różniących się zestawami cech pozostaje kwestią do rozwiązania.

2.7.2. Wykonalność predykcji międzyprojektowej

W swoim studium wykonalności predykcji międzyprojektowej, Zimmermann et al. [227] wykorzystali algorytm drzewa decyzyjnego i takie cechy projektu, jak wykorzystane języki programowania i liczba programistów. Wykazali, że modele predykcji wykorzystujące drzewa decyzyjne wykazują się wysoką efektywnością w ramach jednego projektu, natomiast ich efektywność znacząco spada, gdy wykorzystywane są do predykcji międzyprojektowej, nawet

jeśli zbiory danych charakteryzują się tymi samymi zbiorami cech. Drzewa decyzyjne w studium wykonalności predykcji międzyprojektowej zostały także wykorzystane przez He et al. Struktura tych algorytmów została zbudowana na podstawie różnic w charakterystycznych cechach dystrybucyjnych zbiorów źródłowych i docelowych takich jak: średnia, mediana, wariancja, niesymetryczność i tak dalej [72]. Walidacja wykorzystanych tam algorytmów została przeprowadzona z użyciem najlepszych wyników predykcji dla różnych danych uczących, tak więc wyniki te nie do końca znajdują odzwierciedlenie w ewentualnym praktycznym zastosowaniu tego sposobu przewidywania defektogenności oprogramowania.

2.8. Wykorzystanie predykcji defektów

Jednym z podstawowych celów modeli predykcji defektów jest efektywna alokacja środków przeznaczonych na analizę i testowanie oprogramowania [134], [173]. Jednakże liczba studiów przypadku wykorzystujących te modele w przemyśle jest relatywnie niewielka [43, 111]. Rahmana et al. jako jeden z powodów takiego stanu rzeczy wskazują w swoich badaniach efektywność kosztową [171, 172, 173]. Studium przypadku przeprowadzone w Google przez Lewisa et al. [111] porównało użycie do oznaczania obszarów potencjalnie defektogennych wspomniane wcześniej narzędzie BugCache [101] z algorytmem zaproponowanym przez Rahmana [173], bazującym na liczbie zamkniętych defektów w przeszłości. Badanie to wykazało, że programiści preferują użycie algorytmu Rahmana, jednocześnie nie potrafiąc przełożyć tego na faktyczne korzyści [111].

W innym badaniu Rahman wykazał [174], że predykcja defektów może być pomocna w nadawaniu priorytetów ostrzeżeniom zgłaszanym przez statyczne analizatory kodu (tzw. „wyszukiwarki defektów”) takie jak FindBug [152].

Kolejne możliwe wykorzystanie to zastosowanie wyników predykcji defektów celem priorytetyzacji lub selekcji przypadków testowych. Przykładowo, w testach regresji², wykonanie wszystkich przypadków testowych jest procesem niezwykle kosztownym. Dlatego też zaproponowano szereg metod priorytetyzujących i wybierających przypadki testowe, właśnie z użyciem predykcji defektów [220]. Ponieważ wyniki predykcji defektów są w stanie wskazać defektogenne obszary oprogramowania [101, 173, 226], to bez wątpienia można wykorzystać tę wiedzę do wybrania, spośród wszystkich dostępnych przypadków testowych, tych odpowiednich do testów regresji.

² Testami regresji nazywamy ponowne wykonywanie testów w celu sprawdzenia, że wcześniej przetestowane oprogramowanie nadal działa (odpowiada wymaganiom) po wprowadzeniu zmian.

2.9. Inne istotne kwestie

Nie licząc aspektów predykcji defektów oprogramowania przedstawionych wcześniej, wciąż przed badaczami pozostaje wiele interesujących i istotnych tematów z tego obszaru. Jednym z nich jest tak zwane niezbalansowanie klas, innym – ochrona danych zawierających defekty [163], kolejnym zaś analiza porównawcza pomiędzy modelami predykcji a statycznymi „wyszukiwarkami defektów”, czyli narzędziami analizy statycznej kodu [174].

2.9.1. Niezbalansowanie klas

Zdarza się, że dane uczące są sklasyfikowane nierównomiernie – na przykład, średni udział instancji sklasyfikowanych jako defektogenne w zbiorach danych PROMISE, wykorzystanych przez Petersa et al., to 18% [163]. Przyjmijmy, że posługujemy się modelem, który określa wszystkie instancje jako pozbawione defektów – wtedy *dokładność* modelu (patrz punkt 2.2.1) będzie na poziomie 0.82 pomimo faktu, że żadna instancja zawierająca defekty nie została poprawnie sklasyfikowana. Praca ze zbiorami danych, w którym występuje zjawisko niezbalansowania klas wymaga od badaczy zastosowania szczególnych środków przeciwdziałających negatywnym skutkom tego zjawiska. Tak też było w przypadku badań będących przedmiotem niniejszej pracy – przedsięwzięte środki przeciwdziałające zjawisku niezbalansowania klas zostały opisane w punktach 4.3.2 oraz 5.5.3.

2.9.2. Bezpieczeństwo danych wykorzystywanych do predykcji

Warunkiem przyśpieszenia badań w zakresie predykcji międzyprojektowej jest istnienie ogólnodostępnych, licznych zbiorów danych o defektach. Jednakże firmy tworzące oprogramowanie niechętnie dzielą się takimi danymi z powodu „poufności danych” [163]. Dlatego też badania nad międzyprojektową predykcją defektów są przeprowadzane często na oprogramowaniu otwarto-źródłowym lub na bardzo ograniczonych ilościowo zbiorach danych z zamkniętych systemów komercyjnych [117, 150, 172, 207]. Przykładowo, eksperymenty przeprowadzone przez Zimmermanna et al. dla predykcji międzyprojektowej są niemożliwe do powtórzenia, ponieważ firma Microsoft nie udostępnia swoich zbiorów danych o defektach [227].

Jedną z propozycji rozwiązania tego problemu przedstawili Peters et al. Zaproponowali oni narzędzie MORPH, które modyfikuje zbiory danych o defektach po to, aby rozwiązać problem prywatności tychże danych [163]. MORPH losowo zamienia rekordy danych przy jednoczesnym utrzymaniu warunków ich sklasyfikowania [163]. W ten sposób, jak przekonują autorzy, narzędzie to pozwala zanonimizować oryginalne zbiory danych, a jednocześnie zachować dobre wyniki predykcji defektów bazującej na nowo powstałych danych [163].

2.9.3. Porównanie predykcji defektów i statycznych analizatorów kodu

W porównaniu do modeli predykcji defektów, narzędzia statycznej analizy kodu, w szczególności te, które pełnią funkcję „wyszukiwarki defektów”, wyszukują defekty poprzez wykorzystanie „semantycznych abstrakcji kodu źródłowego” [174]. Rahman et al. porównali techniki predykcji defektów i statyczne wyszukiwarki defektów biorąc pod uwagę efektywność kosztową wykorzystania obu podejść [174]. Badacze odkryli, że oba podejścia mogą się uzupełniać, gdyż mogą odnajdywać różne defekty [174]. Co więcej, ostrzeżenia analizatorów statycznych spriorytetyzowane przez modele predykcji defektów mogą prowadzić do lepszych rezultatów niż priorytetyzacja wbudowana natywnie w „wyszukiwarki defektów” [174]. Prace tego typu pokazują, jak różne kierunki badań mogą być zbieżne w dążeniu do zapewniania wysokiej jakości oprogramowania.

2.10. Wyzwania

Badania nad predykcją defektów oprogramowania wciąż mają przed sobą szereg wyzwań. Pomimo tego, że znaleźć można wiele wartościowych analiz, wciąż jednak zastosowanie tych technik w praktyce nie jest zadaniem łatwym z podanych poniżej powodów:

- Większość z prowadzonych badań została przeprowadzona z wykorzystaniem oprogramowania typu open-source. Wziąwszy pod uwagę fakt, że predykcja międzyprojektowa wciąż nie jest wystarczająco efektywna, modele predykcji zbudowane za pomocą danych pozyskanych z projektów open-source w znakomitej większości nie nadają się do zastosowań w komercyjnych projektach rozwoju oprogramowania. Zbiory danych opisujące oprogramowanie komercyjne nie są publicznie dostępne, ze względu na obawy przedsiębiorstw przed upublicznieniem danych, zatem nie znana jest ich efektywność w przypadku predykcji międzyprojektowej, a skuteczność tego rodzaju predykcji wciąż pozostaje kluczowym aspektem w przypadku zastosowań w komercyjnych projektach rozwoju oprogramowania [163].
- Predykcja międzyprojektowa wciąż pozostaje wyjątkowo skomplikowaną kwestią z dwóch powodów:

Różne zestawy cech Obecnie publicznie dostępnych jest wiele zbiorów danych. Jednakże wiele z nich nie może być wykorzystanych ze względu na to, że zbiory zebrane z różnych domen dysponują różnymi zestawami cech – np. reprezentują wartości innych rodzajów metryk (patrz rozdział 2.3). Modele predykcji bazujące na uczeniu maszynowym nie mogą być budowane na zbiorach danych o różnym zestawie cech.

Wykonalność Badania nad wykonalnością predykcji wciąż są na wczesnym etapie rozwoju. Odnalezienie jednolitego podejścia do kwestii sprawdzania wykonalności

predykcji defektów będzie niezwykle pomocne dla praktycznych zastosowań predykcji defektów, w szczególności w projektach wytwarzania oprogramowania komercyjnego.

- Ponieważ projekty oprogramowania stale rosną (rośnie liczba linii kodu programów), granulacja predykcji³ na poziomie pojedynczych plików może okazać się niewystarczająca, aby zapewnić zadowalającą efektywność kosztową wdrożenia predykcji. Stąd konieczne są dalsze badania w obszarze zwiększania dokładności predykcji na poziomie pojedynczych linii kodu czy każdej zmiany w kodzie źródłowym.
- Modele predykcji znane do tej pory nie zawsze gwarantowały dobre rezultaty. Wraz z rozwojem repozytoriów oprogramowania pojawiają się nowe typy procesów, dostarczające informacji, które nie zostały jeszcze wykorzystane do predykcji defektów. Daje to nowe pole do eksperymentów i być może znalezienia „uniwersalnego modelu predykcji defektów”.

³ Czyli na jakie elementy dzielimy oprogramowanie, aby sklasyfikować te elementy jako defektogenne bądź niedefektogenne.

3. Komercyjne projekty wytwarzania oprogramowania i ich słabe strony z perspektywy tematu badań

3.1. Pozyskiwanie informacji opisujących projekty komercyjnego wytwarzania oprogramowania

Pierwszym etapem badań opisanych w niniejszej pracy było zebranie informacji opisujących pięć komercyjnych projektów oprogramowania, zaproponowanych przez firmę-partnera jako kandydatów do wdrożenia predykcji. Zostały one oznakowane jako Projekt B, Projekt P, Projekt T, Projekt V i Projekt M. Charakterystyka każdego z nich bazuje na ośmiu pytaniach:

1. Czy defekty zostały zareportowane w jakimś konkretnym narzędziu (narzędziach)?
2. Od kiedy defekty są raportowane?
3. Kiedy jest zakładany koniec obecnego projektu?
4. Czy wykonywane są testy regresji?
5. Czy wykonywane są testy nowych funkcjonalności?
6. Jakie jest pokrycie kodu źródłowego przez testy jednostkowe¹?
7. Czy został sporządzony plan testów i przypadki testowe?
8. Jak wiele osób pracuje nad tą aplikacją i jakie są ich role?

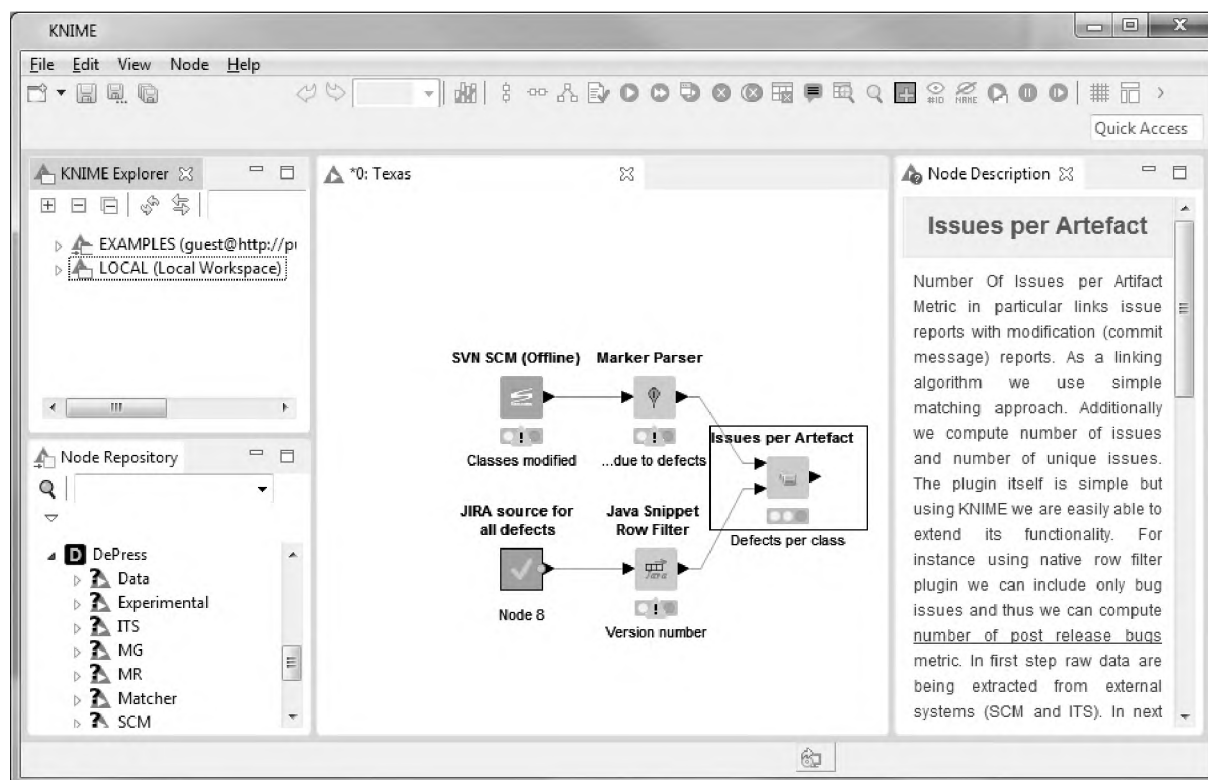
Charakterystyki projektów zostały zebrane w Dodatku A.

3.1.1. Narzędzie DePress i założona forma implementacji predykcji defektów

Na tym etapie badań wzięto pod uwagę dwa warianty przeprowadzenia predykcji defektów. W pierwszym z nich informacje opisujące pierwszą wersję aplikacji (oznaczoną jako n), takie jak defekty zarejestrowane (podczas testów) w systemie śledzenia defektów oraz metryki opisujące ów obciążony defektami kod źródłowy, zostały wykorzystane jako dane uczące.

¹ Testowanie jednostkowe to metoda testowania oprogramowania, za pomocą której poszczególne jednostki kodu źródłowego są testowane – w izolacji od pozostałych jednostek – w celu określenia, czy odpowiadają stawianym im wymaganiom.

Wersja $n+1$ została wykorzystana do predykcji defektów. Zarejestrowane defekty, znalezione w wersji $n+1$ użyto do walidacji predykcji. Następnie metryki defektogennego kodu i defekty z wersji $n+1$ wykorzystano do przygotowania danych uczących wykorzystanych do predykcji dla wersji $n+2$ i tak dalej. W drugim podejściu informacje opisujące pierwszą wersję aplikacji (oznaczoną jako n) zostały wykorzystane do procesu uczenia maszynowego (zbudowania modelu), ale ów model – w formie niezmienniczej – był wykorzystywany do predykcji defektów każdej kolejnej wersji aplikacji ($n+1$, $n+2$, *etc.*).



Rysunek 3.1: Okno platformy KNIME wraz z narzędziem DePress (widocznym w lewym dolnym rogu), wraz z przykładowym diagramem przepływu danych.

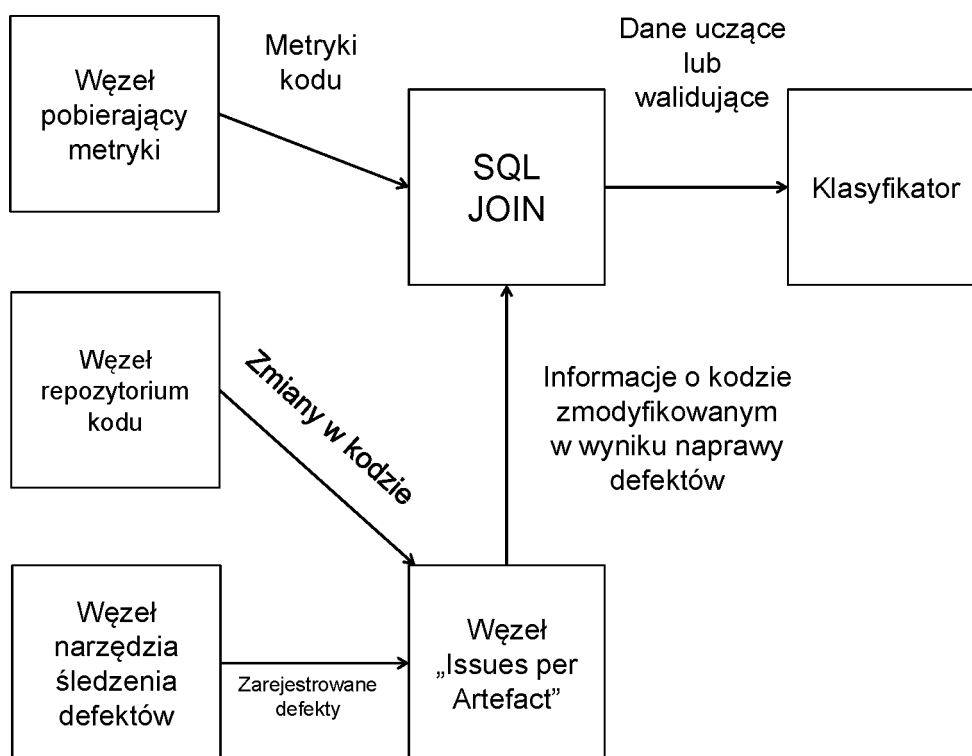
Pierwsze podejście jest bardziej dokładne, ale i również droższe – wymaga zbierania i obróbki danych, jak i budowania modelu predykcji, dla każdej kolejnej wersji aplikacji, co angażuje czas i zasoby. Podejście to może przyjąć różne formy (przykładowo, można wykorzystać uczenie transferowe w celu poradzenia sobie z tak zwanym dryfem koncepcji²). W podejściu tym oczekiwano możliwie dużego zbliżenia się do metody Just In Time, czyli predykcji defektów „w czasie rzeczywistym”, sygnalizującej potencjalną defektogenność kodu źródłowego tuż po tym, jak ów kod zostanie wysłany do repozytorium [161, 166]. Drugie podejście – które, historycznie, jest klasycznym podejściem do predykcji defektów – jest tańsze, ale również mniej dokładne, ponieważ dla kolejnych wersji aplikacji wykorzystuje wciąż ten

² Dryf koncepcji w uczeniu maszynowym i eksploracji danych odnosi się do zmiany w czasie relacji pomiędzy danymi wejściowymi i wyjściowymi w problemie bazowym.

sam model predykcyjny, który powstał z wykorzystaniem kodu źródłowego z „pierwotnej” wersji n , którego cechy mogą zasadniczo różnić się od np. kodu źródłowego z wersji $n+20$.

W obu przypadkach podjęto decyzję o wykorzystaniu narzędzia DePress. Narzędzie to ujrzało światło dzienne w 2014 roku i zostało opisane w publikacji Madeyskiego i Majchrzaka [121]. Pozwala ono na wykorzystanie szerokiego zakresu technik eksploracji danych, w tym predykcji defektów w różnych projektach IT, niezależnie od technologii i wykorzystanego w nich języka. Narzędzie zbudowano w oparciu o platformę KNIME, co czyni DePress oprogramowaniem przyjaznym dla użytkownika i jednocześnie bazuje na wolnej licencji GNU General Public License, zatem może być dowolnie wykorzystywane w celach komercyjnych [105]. Platformę KNIME wraz z narzędziem DePress i przykładowym diagramem przepływu danych przedstawia rys. 3.1.

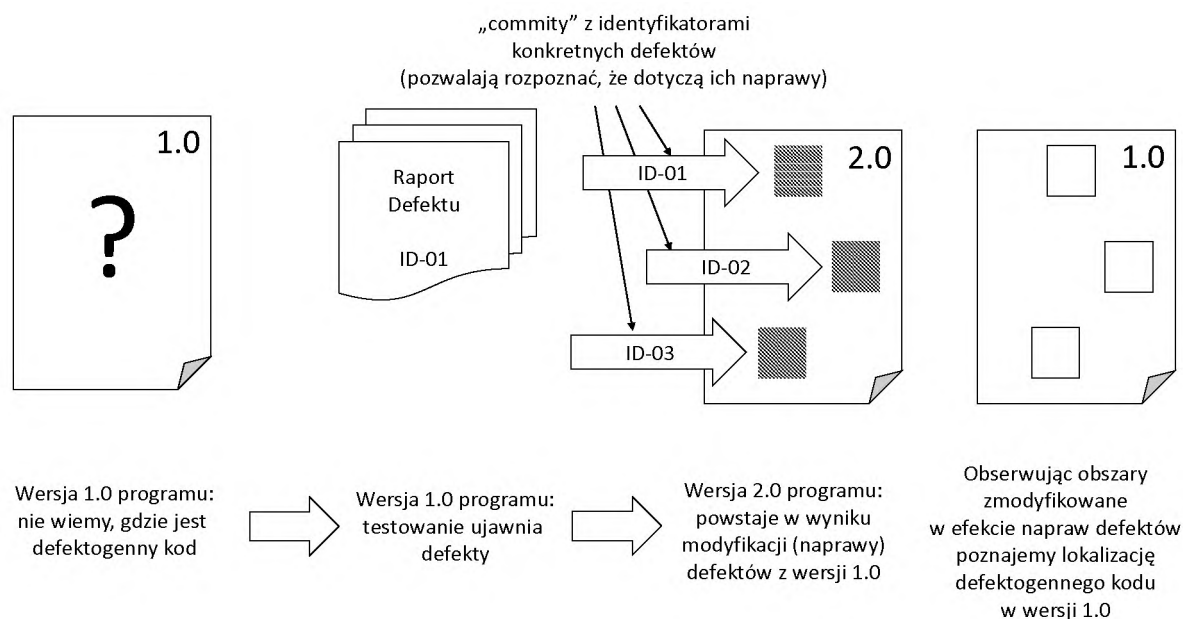
Możliwe zastosowania DePress to analiza różnorodnych cech oprogramowania, produktów, procesów zapewniania jakości, integracji danych z różnych źródeł i narzędzi przeznaczonych do analizy danych – wliczając w to predykcję defektów w oprogramowaniu.



Rysunek 3.2: Uproszczony diagram przepływu danych, użyty w narzędziu DePress

Diagram przepływu danych w narzędziu DePress wykorzystywany w niniejszych badaniach został schematycznie przedstawiony na rys. 3.2. Jest to rozszerzona wersja diagramu z rys. 3.1. Węzeł diagramu odpowiedzialny za interpretację danych pobranych z repozytoriów kodu został użyty do zebrania danych o zmianach w kodzie źródłowym w formacie XML

(na przykład poprzez wygenerowanie przez klienta SVN tzw. „logu zmian w kodzie”). Drugim źródłem danych był plik, który został wygenerowany przez oprogramowanie śledzące defekty i zaimportowany do DePress przez „węzeł narzędzia śledzenia defektów”. Kolejny plik wejściowy zawiera metryki kodu, zebranych przy użyciu dodatkowych narzędzi takich jak wtyczka Eclipse Metrics 2 dla środowiska programistycznego Eclipse czy Microsoft SourceMonitor dla środowiska programistycznego VisualStudio (więcej informacji można znaleźć w [114]).



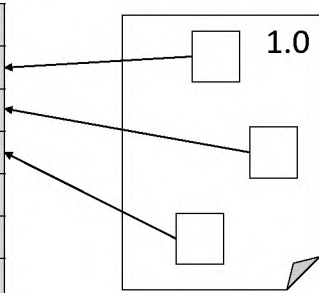
Rysunek 3.3: Separowanie różnego rodzaju modyfikacji kodu (pojedynczych napraw defektów, implementacji nowych funkcjonalności, itp.) poprzez wysyłanie ich do repozytorium kodu w osobnych „commitach” pozwala zaobserwować, które części kodu w pierwotnej wersji aplikacji były odpowiedzialne za później wykryte defekty.

W używanym tu diagramie przepływu danych (przedstawionym na rys. 3.2) metryki kodu dla konkretnych wersji oprogramowania (n) zostały połączone z listą modułów kodu (klasy, metody), zmodyfikowanych z powodu defektów. Połączenie takie jest możliwe w większości przypadków, kiedy programista doda unikatowy identyfikator defektu do komentarza w momencie kiedy kod naprawy defektu jest wysyłany do repozytorium, tak jak zilustrowano to na rysunku 3.3. W następnym kroku, metryki opisujące wybrane moduły kodu są dodawane do danych wykorzystywanych w uczeniu i walidacji modelu (rysunek 3.4).

3.1.2. Wymagania stawiane projektom

Wymagania, które muszą zostać spełnione, aby tylko umożliwić predykcję z wykorzystaniem opisywanego w punkcie 3.1.1 podejścia (wymagania 1, 3 i 4) oraz te,

Metryki kodu										Defektogenne?
5	0	2,128	201	82	2883	1989751	2	145		1
4	0,8	0,267	128	12	2836	1118438	2	257		1
5	0	0,926	180	27	2815	3303991	4	234		1
2	0	0,133	26	19	2806	1375033	2	254		0
1	0,57	0	67	7	2759	2728155	2	288		0
1	0	0	6	0	2751	1920078	2	298		0



Rysunek 3.4: Rozróżnienie obszarów defektogennych i niedefektogennych (por. rysunek 3.3) umożliwia przygotowanie danych uczących – można przyjąć, że na powyższym przykładzie jeden wiersz w tabeli reprezentuje określony obszar kodu źródłowego, np. plik, klasę.

których spełnienie gwarantuje wysoką efektywność predykcji (wymagania 2 i 5 do 8) zostały przedstawione poniżej.

Wymaganie 1: Zestaw plików zawierających informacje o metrykach kodu i o defektach muszą być dostępne.

Wymaganie 2: Zunifikowana metoda oznaczeń wersji oprogramowania w repozytorium kodu musi być dostępna.

Wymaganie 3: Każdy raport o defekcie musi znaleźć się w jednym wpisie w systemie śledzenia defektów.

Wymaganie 4: Każdy wpis o defekcie musi być zarejestrowany pod unikatowym identyfikatorem.

Wymaganie 5: Informacje o tym, w której wersji każdy defekt się pojawił muszą być dostępne.

Wymaganie 6: Kod naprawczy defektu musi być wysłany do repozytorium w pojedynczej paczce (tzw. „commicie”).

Wymaganie 7: Kod naprawczy defektu musi być wysłany do repozytorium z odpowiadającym mu identyfikatorem defektu.

Wymaganie 8: Żadne inne zmiany w kodzie nie mogą być wysłane w tej samej paczce co kod naprawczy pojedynczego defektu.

Tym samym został stworzony zestaw wymagań odnoszący się do pracy z repozytorium kodu i z systemem śledzenia defektów. Następnym krokiem w badaniach było sprawdzenie, czy wymagania zostały spełnione w badanych projektach.

3.2. Wyniki analizy

Projekty rozwoju oprogramowania wybrane do niniejszych badań zostały sprawdzone pod kątem wymagań przedstawionych w punkcie 3.1.2. Poczynione obserwacje mogą zostać

podzielone na dwa obszary: związane z repozytorium kodu i związane z systemem śledzenia defektów.

3.2.1. Elementy związane ze śledzeniem defektów

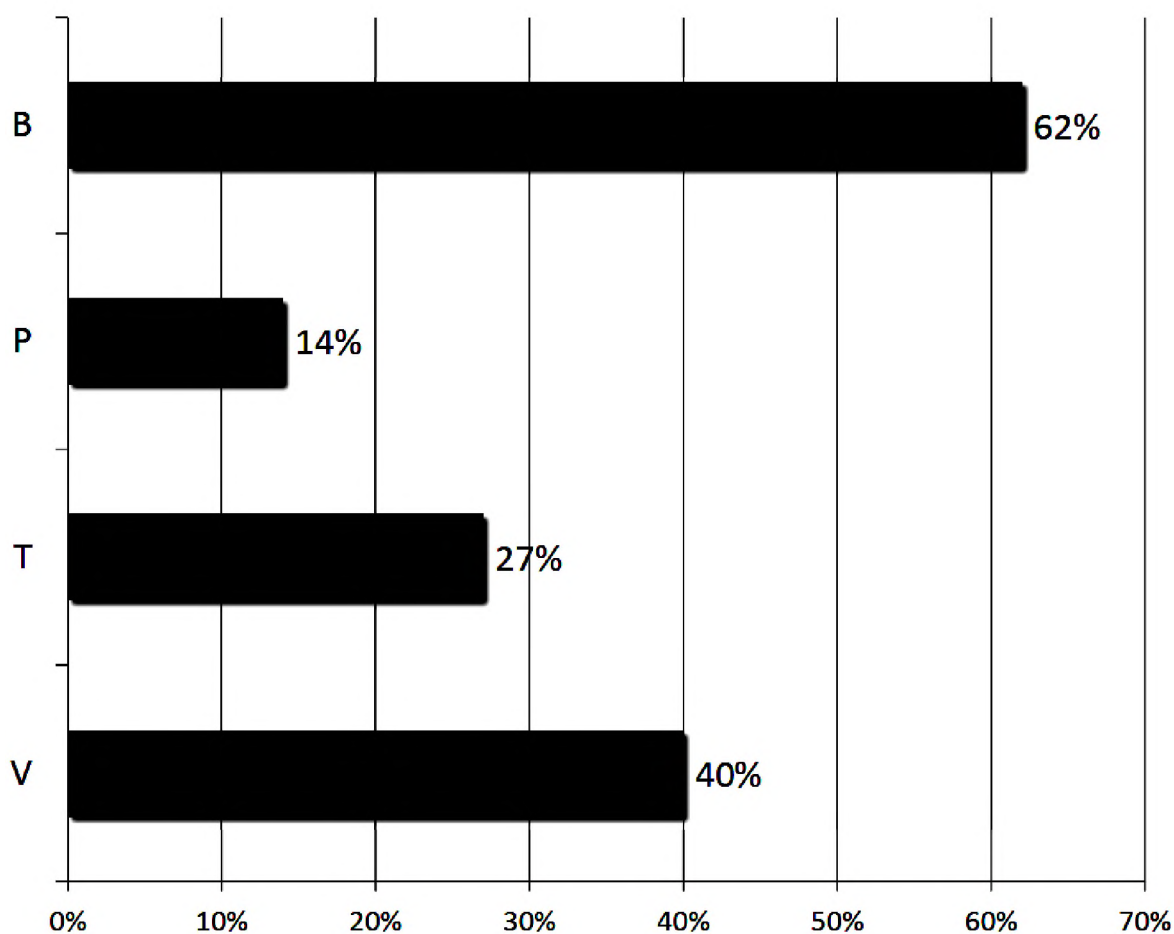
Z wymagań przedstawionych w punkcie 3.1.2, wymagania od 1 do 5 mogą być skojarzone z obszarem należącym do systemu śledzenia defektów. *Wymaganie 1* zostało spełnione we wszystkich analizowanych projektach oprogramowania. W większości przypadków zdobycie plików zawierających informacje o defektach było proste i ograniczało się tylko do włączenia odpowiedniej funkcji „eksportuj plik” w systemie śledzenia defektów. Jednakże podczas badań pojawiło się kilka pomniejszych problemów związanych z plikami danych. Nie miały one jednak wpływu na proces predykcji. Pierwszy z nich był ściśle powiązany z oprogramowaniem JIRA, wykorzystywanym w niektórych z badanych środowisk. Ustawienia globalne JIRA do wszystkich powiązanych projektów były stałe i pozwalały na wydobycie jedynie 1000 rekordów w jednym pliku. Ten mały problem miał wpływ na czas potrzebny na przygotowanie danych wejściowych dla predykcji defektów w projektach zawierających kilka tysięcy zarejestrowanych defektów. Aby przygotować dane wejściowe, zestawy defektów musiały zostać podzielone ręcznie na części zawierające maksymalnie 1000 rekordów. Wtedy mogły one być eksportowane, aczkolwiek po tej operacji musiały one zostać ze sobą połączone. Jest to wprawdzie prosta rzecz, ale zabrała sporo czasu i wpłynęła na ramy czasowe predykcji.

Podobny problem dało się zauważyć w projektach wykorzystujących kilka różnych systemów śledzenia defektów. W tym przypadku, w celu przygotowania ostatecznego zbioru danych, dane z różnych systemów kontroli wersji musiały zostać pobrane – często do więcej niż jednego pliku, a potem za pomocą narzędzia DePress sprowadzone do standardowego formatu i połączone. Ekstremalny przykład takiej sytuacji pojawił się w projekcie M, gdzie duża część defektów została zgromadzona jako prosta konwersacja mailowa między testerami i programistami. Każda z nich musiała być zebrana do jednego pliku, załadowana do narzędzia DePress, sformatowana i w końcu powiązana z jednym zestawem danych.

Użycie zunifikowanego systemu nazewnictwa w obu źródłach danych – w systemie śledzenia defektów i w repozytorium kodu jest istotne: jeżeli to założenie opisane w punkcie 3.1.2 jako *Wymaganie 2* nie jest spełnione, to wtedy powiązanie danych opisujących kod z odpowiedniej wersji aplikacji (wydania) z informacją o defekcie jest praktycznie niemożliwe. Należy wziąć pod uwagę, że w regularnym cyklu życia oprogramowania w danej firmie raport o defekcie może być zarejestrowany w dowolnej chwili dla przynajmniej trzech różnych wersji aplikacji. Przykładowo. jeżeli użytkownik końcowy zauważy błąd, to powiązany raport będzie dotyczył wersji n oprogramowania (jeżeli założymy, że tylko jedna wersja danego oprogramowania jest wykorzystywana przez użytkownika końcowego, co samo

w sobie jest mało prawdopodobne). W tym samym czasie testerzy testują wersję $n + 1$, a programiści tworzą wersję $n + 2$ – również dla tych dwóch wersji mogą być tworzone raporty.

Taki przypadek jest analogiczny do sytuacji, w której różne wersje oprogramowania są testowane równolegle (tj. w tym samym czasie). Dla czterech projektów: B, P, T oraz V, diagramy przedstawiające najczęściej pojawiające się numery wersji w systemach śledzenia defektów są przedstawione w Dodatku B. Projekt M nie został tam ujęty, ponieważ z powodu wczesnego etapu prac rozwojowych tylko wersja 1.0.0.0 została wykorzystana do implementacji ewentualnych zmian czy napraw defektów. W związku z tym w systemie śledzenia defektów projektu M nie istnieje żaden inny numer wersji jak tylko 1.0.0.0. Jak



Rysunek 3.5: Procentowy udział raportów o defektach bez informacji o wersji, z podziałem na projekty

przedstawiono na wzmiankowanych diagramach, nazewnictwo wersji nie jest jednolite w niektórych przypadkach (rys. B.1, B.2, B.3). Problem staje się jeszcze bardziej złożony, jeżeli zestawione zostanie ze sobą nazewnictwo w systemach śledzenia defektów z tym znajdującym się w repozytoriach kodu. W projektach P i V defekty były zarejestrowane przy użyciu innego sposobu numeracji wersji niż ten, który został zaadaptowany w repozytorium. W związku z tym, informacje o defektach nie mogą być automatycznie powiązane z danymi powiązanymi

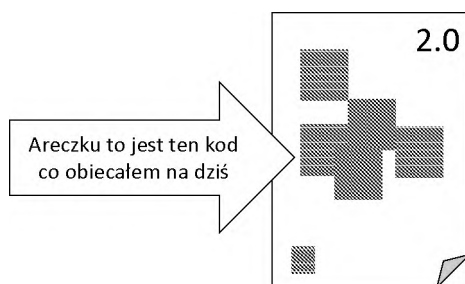
z kodem, takimi jak metryki czy dziennik (log) zmian. Ręczne połączenie jest czasochłonne i może mieć wpływ na ramy czasowe procesu.

Z drugiej strony jednak, niezwykle wręcz zgodność pojawiła się w zakresie *Wymagania 3*. W każdym badanym projekcie każdy zaraportowany defekt był śledzony za pomocą jednego wpisu w systemie do tego służącym. To w naturalny sposób doprowadziło do spełnienia *Wymagania 4* – gdzie dla każdego oprogramowania śledzącego defekty, takiego jak JIRA, HP Quality Center czy IBM ClearQuest, unikatowy identyfikator został przypisany do każdego nowego wpisu.

Wymaganie 5 jest ściśle powiązane z problemem numeracji. Jak łatwo można zauważyć na diagramach przedstawionych w Dodatku B, w większości zarejestrowanych defektów nie ma informacji o tym w jakiej wersji (wydaniu) oprogramowania się one pojawiły. Dane procentowe opisujące to zjawisko są przedstawione na rys. 3.5. W tym przypadku proponowane rozwiązanie predykcji przedstawione w punkcie 3.1.1 pozwala na wykorzystanie wersji nie zawierających informacji o numerze. Należy jednak oczekiwać spadku jakości – przykładowo, jeżeli defekty zostały znalezione w części kodu źródłowego, która następnie została refaktoryzowana³, to doprowadzi to do sytuacji, w której metryki kodu opisujące ten właśnie zmieniony fragment zostaną oznaczone w zbiorze danym wykorzystanym do nauki jako obarczone defektem.

3.2.2. Elementy związane ze śledzeniem zmian w kodzie

Wymagania 1, 2, 6 i 7 oraz 8 z punktu 3.1.2 zaliczyć można do obszaru repozytorium kodu źródłowego. Jak zostało przedstawione wcześniej, *Wymaganie 1* zostało spełnione we wszystkich badanych projektach. Problemy wywołane przez różną konwencję w zakresie nazewnictwa wykorzystywanego w systemach śledzenia defektów i repozytoriach, zostały opisane w punkcie 3.2.1.



Rysunek 3.6: Zła praktyka korzystania z systemu kontroli wersji: wiele zmian w jednym „commicie” wraz z komentarzem nie niosącym użytecznej informacji – por. rysunek 3.7.

³ Refaktoryzacja kodu źródłowego to proces, w wyniku którego następuje (syntaktyczna) zmiana kodu, ale bez (semantycznej) zmiany jego funkcjonalności. Najczęstszym powodem refaktoryzacji kodu jest potrzeba podniesienia jego przejrzystości.

W trakcie badań można było zauważyć, że jeżeli najlepsze praktyki w zakresie nazewnictwa nie są obligatoryjne dla projektów, to wtedy *Wymagania 6, 7 i 8* nie są spełnione. Zamiast wysyłać do repozytorium każdą zmianę kodu oddzielnie, niezależnie, z odpowiednim komentarzem i przy wykorzystaniu jednego „commitu” (rys. 3.3), powszechnie stosowaną praktyką jest wysyłanie różnych zmian w jednej „paczce” w ramach jednego, zbiorczego „commitu” (ilustruje to rys. 3.6). Takie podejście może wpływać na rzetelność danych wejściowych predykcji defektów w różnym stopniu, w zależności od sytuacji:

- programista wysłał do repozytorium kod naprawczy defektu w postaci pojedynczej „paczki kodu”, ale nie dodał odpowiedniego komentarza z unikatowym identyfikatorem defektu (nadanego przez system śledzenia defektów);
- programista wysłał do repozytorium kod naprawczy defektu z komentarzem zawierającym identyfikator, ale paczka zawiera inne małe zmiany, nie powiązane z defektem, jak na przykład refaktoryzację lokalnego kodu (tzw. „ukryta refaktoryzacja”);
- programista wysłał kod naprawczy, a także niewielkie dodatkowe zmiany, ale nie dodał identyfikatora defektu podczas wysłania go do repozytorium;
- programista wysłał różne zmiany w kodzie, powiązane z różnymi zadaniami (naprawa defektów, nowe funkcjonalności, refaktoryzacja) w pojedynczej paczce, załączając (lub nie), odpowiedni komentarz.

Ostatnia z wymienionych praktyk sprawia, że dane opisujące zmiany w kodzie są bezużyteczne dla wybranego podejścia do predykcji defektów w wybranych projektach.

W czterech z pięciu badanych projektów oprogramowania, gdzie rozwijano dojrzałe systemy z wieloma wcześniejszymi wersjami wypuszczanymi przez długi okres czasu, odpowiednie praktyki pracy z systemami kontroli wersji były zachowane. Skutkowało to wysoką jakością danych opisujących zmiany w kodzie wynikające z naprawy defektów. Wyjątkiem był mały i relatywnie nowy Projekt M, gdzie, zwłaszcza w początkowych fazach projektu, duża liczba „commitów” zawierała jednocześnie (w tej samej „paczce”) wiele modyfikacji różnego rodzaju, w szczególności nowe funkcjonalności, refaktoryzacje i naprawy defektów. Wiele „commitów” zawierało kilka tuzinów zmian w kodzie różnego rodzaju. Co więcej, zmiany te nie były opisane w komentarzach do „commitów”. Komentarze je opisujące, znajdujące się w repozytorium Projektu M, były zbyt ogólne. Większość z nich opisywała jedynie główną funkcjonalność przesyłaną w „commicie”. Kod naprawczy dostarczany wraz z nimi często nie był w ogóle opisany. Nawet w przypadkach, w których zarówno modyfikacje jak i naprawy defektów były opisane w komentarzach, miały one tendencję do bycia niejednoznacznymi, jak na przykład: „Kilka poprawek w bazie danych”, „Poprawki w interfejsie sieciowym”. Żaden z tych komentarzy nie zawierał odniesienia do raportu o defekcie, dlatego też nie można określić jakie defekty zostały naprawione w której paczce. W takiej sytuacji, gdy brakuje bezpośredniego połączenia między defektem a zmianami w

3.3.1. Elementy związane ze śledzeniem defektów

W badanych projektach, profesjonalne oprogramowanie do śledzenia defektów, jak JIRA czy Microsoft Team Foundation Server (niedawno przemianowany na Microsoft Azure DevOps), nadaje unikatowe identyfikatory dla każdego zarejestrowanego defektu. Członkowie zespołu powinni rejestrować każdy znaleziony defekt jako pojedynczy wpis w systemie śledzenia defektów. Taka praktyka zapewni, że każdy zarejestrowany defekt otrzyma unikatowy identyfikator przypisany przez system śledzący. Powinien być on wykorzystany później, podczas naprawy defektów. Przegląd defektów zgromadzonych w systemie śledzenia również powinien być pożądaną praktyką, aby uniknąć powielania defektów. Odnaleziony zarejestrowany defekt, który jest duplikatem zarejestrowanego innego defektu, podcina w trakcie przeglądu pod „oryginał” ze statusem 'duplicate'. Dzięki takiemu oznaczeniu, na etapie zbierania danych do procesu uczenia maszynowego pomija się owe zdublowane defekty.

Celem uzupełnienia brakujących informacji o wersji z raportów o defektach (jak zostało to przedstawione na rys. 3.5), zostały zebrane dodatkowe informacje, pozwalające znaleźć zależność między numerem wersji, a czasem, w którym ta konkretna wersja aplikacji została zainstalowana w środowisku testowym (w przypadku, gdy defekt został znaleziony przez testera) bądź produkcyjnym (gdy defekt znalazł użytkownik końcowy). Dane te zostały później wykorzystane przez odpowiedni diagram przepływu DePress w celu uzupełnienia brakujących informacji w danych dostarczonych przez system śledzenia defektów.

3.3.2. Elementy związane z repozytorium kodu

Żadne inne modyfikacje nie powinny być wysyłane wraz z kodem naprawczym do repozytorium – wysoka kultura pracy powinna być gwarantem tego, że kod będzie wysyłany oddzielnie do repozytorium po każdej zmianie wraz z odpowiednim jej opisem (ilustruje to rys. 3.3) – jeżeli jest to naprawa określonego defektu, to komentarz (opis) powinien zawierać identyfikator defektu przypisany przez system śledzenia defektów.

W celu zapewniania takiego podejścia, oprócz wykorzystania identyfikatorów defektów zapisanych w repozytorium do połączenia konkretnych zmian w kodzie z odpowiadającymi im defektami, użyto specjalnych węzłów DePress wykonujących semantyczną lub składniową analizę dziennika (logu) zmian kodu w celu połączenia zmian w kodzie z raportami o defektach zebranymi w systemie śledzenia. Analiza składniowa wyszukuje predefiniowane schematy znajdujące się w komentarzach „commitów” celem rozpoznania kodu naprawczego. Analiza semantyczna z kolei wykorzystuje relację czasową bazując na założeniu, że status raportu o defekcie jest zmieniany na „naprawiony” zaraz po wysłaniu odpowiedniego kodu do repozytorium. Te metody wyszukiwania zależności między zmianą kodu a określonym defektem były wykorzystane z dobrymi wynikami, a następnie opisane przez Śliwerskiego, Zimmermana i Zellara [193].

3.4. Podsumowanie tego etapu badań

Jak wynika ze wstępnej analizy projektów rozwoju oprogramowania, które zostały wytypowane przez firmę-partnera badań do wdrożenia predykcji defektów oprogramowania, odpowiednia kultura pracy jest niezbędna przy tworzeniu oprogramowania i musi być przestrzegana od początku. Nawet najmniejsze odstępstwo od najlepszych praktyk – jak na przykład rezygnacja z wprowadzenia unikatowych identyfikatorów defektów w komentarzach „commitów”, początkowo wydająca się nieszkodliwą, może skutkować tym, że jakakolwiek innowacyjna technologia, która będzie w przyszłości wdrażana, może okazać się trudną bądź niemożliwą do wdrożenia lub co najmniej ucierpieć z powodu niskiej jakości danych. Wraz z rozwojem przygotowywanych aplikacji, nawet najmniejsze rozbieżności bądź niedopatrzenia mogą zmienić się w poważne problemy związane z jakością bądź danymi związanymi z projektem, jak na przykład przedstawiony wcześniej 62-procentowy brak danych o wersji w raportach defektów (rys. B.1).

Już na początkowym etapie badań będących przedmiotem niniejszej rozprawy ważną rolę w zbieraniu informacji spełniło narzędzie DePress wraz z zestawem interfejsów dedykowanych danym narzędziom programistycznym. W przyszłości, gdy wymagania (czynniki) umożliwiające wdrożenie predykcji defektów w projekcie wytwarzania oprogramowania nie będą mogły zostać spełnione w projekcie od samego jego początku, DePress może wesprzeć konkretne projekty w celu odzyskania zagubionych danych, wykorzystując do tego swoje możliwości w zakresie wykorzystania i łączenia różnych źródeł danych, a także mechanizmów eksploracji danych jak i analizy składniowej i semantycznej.

Opisane w tym rozdziale badania pozwoliły zrealizować następujące cele szczegółowe, nakreślone w punkcie 1.2 niniejszej pracy:

- Cel 1: Określono, jakie elementy wytwarzania oprogramowania muszą zostać zachowane, aby wdrożenie predykcji defektów w komercyjnych projektach rozwoju oprogramowania zakończyło się powodzeniem.
- Cel 2: Oceniono wybrane komercyjne projekty rozwoju oprogramowania pod kątem możliwości zastosowania predykcji defektów oprogramowania.
- Cel 3: Rozpoznano potencjalne wąskie gardła, które mogą wpłynąć w negatywny sposób na predykcję.
- Cel 4: Zaproponowano określone zmiany, które mogą zostać wprowadzone w komercyjnych projektach rozwoju oprogramowania, celem łatwej implementacji predykcji defektów w przyszłości.

4. Analiza efektywności kosztowej pilotażowego zastosowania predykcji defektów w komercyjnym projekcie wytwarzania oprogramowania

4.1. Po co badać efektywność kosztową predykcji defektów?

Jak opisano w rozdziale 2, pierwsze próby wykorzystania uczenia maszynowego do zapewniania jakości w wytwarzaniu oprogramowania pojawiły się już we wczesnych latach dziewięćdziesiątych XX wieku [145]. Od tamtego momentu technika ta uległa znacznemu usprawnieniu, ale nie zyskała ona jak dotąd większej popularności w projektach komercyjnych. Do niedawna wdrażanie predykcji w wybranym projekcie wymagało pochłaniających czas przygotowań, a także specjalnego dodatkowego programu lub programów, które zdobywały potrzebne informacje, a następnie przepuszczały je przez odpowiednie procedury wstępnej obróbki.

Jedną z przyczyn małej popularności wykorzystania uczenia maszynowego w predykcji defektów w projektach komercyjnych jest duża różnorodność danych niezbędnych do wykonania predykcji, jak również wiele różnych źródeł danych, które są spotykane w komercyjnych projektach rozwoju oprogramowania. Przeszkodą jest konieczność zbierania danych z wielu różnych źródeł, w różnych formatach i przy użyciu różnych protokołów dostępu lub programistycznych interfejsów aplikacji (API). Kolejną barierą jest łączenie danych za pomocą różnych algorytmów uczenia maszynowego o różnych charakterystykach. Prognozowanie wad oprogramowania uznano za proces zbyt złożony, zbyt kosztowny i zbyt czasochłonny, a do tej pory nie było znanych rozwiązań pozwalających na połączenie go w jedną, uniwersalną aplikację do przewidywania defektów, która mogłaby być wykorzystywana w różnych projektach.

W momencie podjęcia badań będących przedmiotem niniejszej pracy jedynym znanym autorowi narzędziem wypełniającym powyższą lukę był opisany w punkcie 3.1.1 program DePress. Cechy tego narzędzia czynią go bardzo obiecującym w kontekście komercyjnego stosowania predykcji defektów oprogramowania, z jednym zastrzeżeniem – program ten nie został wcześniej przebadany pod kątem korzyści, jakie może przynieść jego zastosowanie w środowisku komercyjnego wytwarzania oprogramowania. Zatem, aby móc na późniejszym

etapie prac uznać DePress za podstawowe narzędzie do predykcji defektów oprogramowania komercyjnego, należało wprzód opowiedzieć na następujące pytania:

- *Jaki jest najwyższy poziom czułości predykcji możliwy do osiągnięcia przez narzędzie DePress w bazowej konfiguracji przy wykorzystaniu danych z projektu komercyjnego?* Możliwe korzyści różnią się w zależności od potencjalnej efektywności predykcji. Sugeruje to, że na początku należy zweryfikować jaka jest najwyższa wartość *czułości* (patrz punkt 2.2.1) predykcji wykonanej w całości przez oprogramowanie DePress. Jako, że DePress pozwala na duży zakres własnej modyfikacji, ze względu na jego architekturę bazującą na rozszerzeniach (pluginach) oraz na to, że jest to oprogramowanie open-source, niektóre modyfikacje programu DePress mogą być kosztowne i skomplikowane. Dlatego też zdecydowano się na ograniczenie użycia DePress tylko do jego bazowych ustawień.
- *Czy wykorzystanie narzędzia DePress będzie opłacalne dla projektu komercyjnego?* Aby odpowiedzieć na to pytanie, porównano koszty wdrożenia i użycia predykcji bazującej na DePress do potencjalnych korzyści które mogłyby z tego wynikać. Aby to osiągnąć wykorzystano metryki takie jak zwrot z inwestycji (ROI – ang. *Return On Investment*) i stosunek korzyści do kosztów (BCR – ang. *Benefit-Cost Ratio*).
- *Jak efektywny kosztowo jest DePress w predykcji defektów dla projektów oprogramowania komercyjnego gdy osiągnięta zostanie najwyższa wartość czułości predykcji, możliwa do osiągnięcia przez to narzędzie?* Jeżeli odpowiedź na poprzednie pytanie jest pozytywna, to następnym krokiem jest weryfikacja jakie będą korzyści wynikające z najwyższej wartości czułości możliwej do osiągnięcia przez bazowe ustawienie predykcji w DePress.

4.2. Komercyjny projekt wytwarzania oprogramowania

Główną motywacją dla firmy-partnera na tym etapie było sprawdzenie, czy zastosowanie predykcji defektów i narzędzia DePress w celu podniesienia jakości w ich projektach rozwoju oprogramowania będzie efektywne kosztowo.

4.2.1. Charakterystyka wytwarzanego oprogramowania

Ważnym kryterium w zakresie wyboru odpowiedniego projektu do celów badania efektywności kosztowej było sprawdzenie, czy jest on wystarczająco dojrzały, aby móc dostarczyć historycznych informacji, które można by wykorzystać jako źródło danych uczących. Bazując na wynikach badań nad projektami wskazanymi przez firmę-partnera, opisanych w rozdziale 3, ostatecznie wybranym został Projekt T, w ramach którego tworzy się i utrzymuje aplikację nazwaną umownie Xates, jednocześnie dostarczając informacji koniecznych do osiągnięcia najwyższej możliwej wartości *czułości* predykcji, takich jak:

- informacje o znalezionych defektach,

- zarejestrowane zmiany w kodzie będące wynikiem napraw defektów,
- metryki kodu.

W analizowanych projektach możliwe jest widoczne oddzielenie tych części kodu, która były zastosowane tylko do naprawy defektów. Jest to możliwe dzięki zastosowaniu praktyki wykorzystanej przez programistów, opisanej w rozdziale 3: każda modyfikacja kodu źródłowego jest wysłana do repozytorium wraz z odpowiednim komentarzem. W przypadku modyfikacji będącej rezultatem naprawy danego defektu, unikatowy identyfikator defektu jest włączony do komentarza. Co więcej, takie działanie jest przykładem właściwego użycia systemu kontroli wersji. Informacja o numerze kolejnej wersji oprogramowania jest dostępna, a nazewnictwo każdej z nich jest ustandaryzowane.

Z perspektywy organizacyjnej, Xates jest narzędziem zarządzania dokumentami, które wspiera proces certyfikowania, zwany Vehicle Type Approval (VTA), dla komponentów i pojazdów produkowanych przez firmę-partnera badań. Certyfikat VTA potwierdza, że produkty spełnią standardy wydajności. Kluczowymi użytkownikami narzędzia Xates są kierownicy i inżynierowie, a także przedstawiciele branży motoryzacyjnej i europejskie firmy handlowe. Proces certyfikacyjny VTA jest istotny dla użytkownika, ponieważ dostępność produktów firmy-partnera jest od tego procesu bezpośrednio zależna. Powoduje to, że niezawodność, rozwój i pielęgnacja oprogramowania Xates jest niezwykle istotna. Defekty w aplikacji mogą opóźnić pracę inżynierów certyfikacji, co jest nie do zaakceptowania, ponieważ takie opóźnienia mogą wpłynąć na czas przeznaczony na proces certyfikowania i co więcej, mieć negatywny wpływ na to, kiedy produkt firmy będzie dostępny dla klientów.

Z perspektywy technicznej, Xates to aplikacja bazująca na języku Java, napisana przy użyciu platformy Java Enterprise Edition [10]. Główne środowisko programistyczne wykorzystane w procesie produkcji to Eclipse Integrated Development Environment [203], a Apache Maven [201] jest narzędziem wykorzystanym do procesu ciągłej integracji¹. Do zbudowania całej aplikacji musi być skompilowanych domyślnie 13 różnych projektów Maven. Do celach badawczych udostępnionych zostało 21 kolejnych wydań (wersji) oprogramowania, zaczynając od Xates 4.0.0, a kończąc na wersji 6.0.0. Kod źródłowy był magazynowany w repozytorium zarządzanym przez system kontroli wersji Apache Subversion (SVN) [202].

W rozważanym projekcie można zaobserwować trzy *główne* fazy cyklu oprogramowania (fazy projektu):

1. Pisanie kodu.
2. Testowanie.
3. Faza utrzymania, „po wydaniu”.

¹ Ciągła integracja to praktyka stosowana w rozwoju oprogramowania, polegająca na częstej, regularnej integracji bieżących zmian w kodzie z całą aplikacją, poprzez regularne budowanie (kompilację) aplikacji.

4.3. Budowanie modelu predykcji

4.3.1. Wykorzystane zbiory danych

Na jakość predykcji wpływać będą w szczególności dwa czynniki powiązane z wykorzystanymi zbiorami danych:

- rozmiar zbiorów,
- liczba defektów znaleziona w wydaniu źródła zbiorów.

Rozmiar zbiorów jest istotny, ponieważ w uczeniu maszynowym efektywność procesu jest w większości przypadków zależna od rozmiaru danych: im jest on większy, tym większa efektywność mechanizmu uczenia.

Tabela 4.1: Metryki kodu (opisane szerzej w punkcie 2.3.1) opisujące klasy języka Java zbierane przez narzędzie Eclipse Metrics 2.

Nazwa metryki	Opis
NumberOfOverriddenMethods	Całkowita liczba metod w klasie, które zostały nadpisane przez klasę dziedziczącą
NumberOfAttributes	Całkowita liczba atrybutów klasy
NumberOfChildren	Całkowita liczba klas potomnych
NumberOfMethods	Całkowita liczba metod w klasie
DepthOfInheritanceTree	Głębokość drzewa dziedziczenia
LackOfCohesionOfMethods	Spójność kodu klasy obliczana metodą Hendersona-Sellersa [74]
NumberOfStaticMethods	Całkowita liczba metod statycznych w klasie
SpecializationIndex	Metryka określająca specjalizację kodu, zaproponowana przez Hendersona-Sellersa [73], będąca złożeniem trzech innych metryk zbieranych przez narzędzie: $\frac{NumberOfOverriddenMethods \times DepthOfInheritanceTree}{NumberOfMethods}$
WeightedMethodsPerClass	Suma złożoności cyklomatycznych zmierzonych dla wszystkich metod w klasie
NumberOfStaticAttributes	Całkowita liczba statycznych atrybutów klasy

Liczba defektów ma bezpośredni związek z możliwością wystąpienia problemu niebalansowania klas – występującym, gdy łączna liczba instancji danych z jednej klasy (np. „defektogennej”) jest istotnie mniejsza niż łączna liczba instancji innej klasy („poprawnej”). Jeżeli dla dużej aplikacji w jej konkretnym wydaniu wykryta zostanie relatywnie niewielka

liczba defektów (co jest typowe dla wielu projektów [83]), to można oczekiwać wysokiego niezbalansowania klas.

Biorąc pod uwagę powyższe czynniki, przeanalizowano każdą dostępną wersję oprogramowania pod względem jej rozmiaru i liczby zarejestrowanych defektów. Jako wyznacznik rozmiaru wybrano liczbę oddzielnych modułów (tzn. klas Java). W celu zdobycia liczby modułów dla każdej wersji zebrano metryki kodu. Zebranie metryk musiało być przeprowadzone podczas procesu kompilacji aplikacji (w tym przypadku poprzez narzędzie Maven). W celu zmniejszenia wpływu na pracę zespołu projektowego zdecydowano się na skopiowanie całego kodu źródłowego z repozytorium na dysk jednej ze stacji roboczych i skompilowanie każdej wersji lokalnie, w środowisku programistycznym Eclipse. W celu zebrania metryk podczas kompilacji użyto rozszerzenia środowiska pod nazwą Eclipse Metrics 2. Eclipse Metrics 2 to darmowe rozszerzenie dystrybuowane na licencji Common Public License pracujące jako wtyczka (plugin) do Eclipse IDE [21]. Było stworzone jako kontynuacja Eclipse Metrics – oryginalnego pluginu Eclipse do zbierania metryk [181]. Eclipse Metrics 2 pozwala na zebranie różnych rodzajów metryk (zob. tabela 4.1) i eksportuje je jako plik XML. Plik ten może być potem sparsowany przez węzeł DePress zwany Eclipse Metrics (podobnie jak samo narzędzie). Najlepszym źródłem do uzyskania informacji o liczbie defektów wykrytych w poszczególnych wersjach jest narzędzie służące do śledzenia defektów. W przypadku projektu Xates wykorzystano do tego celu oprogramowanie JIRA stworzone przez firmę Atlassian. Oznacza to, że w badaniu brano pod uwagę wszystkie defekty zarejestrowane w narzędziu śledzenia defektów JIRA (oraz pozostawiające ślady zmian kodu – w wyniku usuwania defektów – w repozytorium kodu źródłowego SVN), niezależnie od ich pochodzenia. Źródłem defektów w przypadku tego projektu mogły być testy integracyjne oraz testy funkcjonalne i нефункционалне. Defekty w skryptach testowych nie były brane pod uwagę – nie były one rejestrowane w JIRA jako wady oprogramowania Xates, nie mogło być też żadnych śladów w kodzie źródłowym aplikacji Xates po naprawie defektów w skryptach. Podobnie jak Eclipse Metrics, JIRA pozwala również na eksport danych o defektach do postaci XML, który może być następnie przetwarzany przez węzeł DePress o nazwie Jira Offline.

Po przetransferowaniu do DePress, rozmiar i dane o defektach mogą być zebrane w kilku prostych krokach. Wyniki dla każdej wersji Xates są zaprezentowane w tabeli 4.2.

Po przeanalizowaniu danych przedstawionych w tabeli 4.2, zdecydowano się na wykorzystanie zbiorów danych bazujących na wydaniu 4.0.0 zarówno do procesu nauczania jak i procesu walidacji klasyfikatora. Największa liczba znalezionych defektów była porównana z liczbą modułów dostępnych do analizy, co uczyniło wersję 4.0.0 najlepiej przystosowaną do powyższych celów. Do dalszych badań klasyfikowały się też wersje 5.0.0 i 6.0.0. Inne wydania posiadały zdecydowanie mniejszą liczbę zarejestrowanych defektów przy podobnej liczbie modułów zapisanych w repozytorium kodu w wyniku czego stosunek liczby defektów

Tabela 4.2: Liczba defektów i modułów (plików źródłowych) dla każdej z wersji oprogramowania

Numer wersji	Defekty	Moduły	Średnia liczba zarejestrowanych defektów na moduł kodu
4.0.0	837	744	1,125
4.0.1	19	685	0,03
4.0.2	23	673	0,03
4.1.0	45	597	0,08
4.1.1	17	546	0,03
4.2.0	54	608	0,09
4.3.0	10	660	0,02
4.3.1	8	608	0,01
4.4.0	29	616	0,05
4.5.0	11	747	0,01
5.0.0	270	744	0,36
5.1.0	15	593	0,03
5.2.0	23	606	0,04
5.4.0	35	582	0,06
5.5.0	42	582	0,07
5.6.2	10	612	0,02
5.6.3	1	612	0,002
6.0.0	462	688	0,67

do liczby modułów wynosił poniżej 0,1². Jako, że każdy z zapisanych modułów, bazując na zarejestrowanych defektach, może zostać sklasyfikowany jedynie jako „0” lub „1”, to dla wydań innych niż 4.0.0, 5.0.0 oraz 6.0.0 zaobserwowane zostanie znacznie silniejsze (niż w wydaniach „głównych”) niebalansowanie klas, które w skrajnym przypadku może skutkować tym, że w zbiorze uczącym znajdzie się 306 modułów oznaczonych „0” i tylko jeden oznaczony „1”. Z tego względu zrezygnowano z dalszych badań na wydaniach nie będących „głównymi” (tj. 4.0.0, 5.0.0 oraz 6.0.0).

Zbiory danych wykorzystane do obu celów – nauczania i walidacji – zostały zbudowane przez podzielenie danych z wersji 4.0.0 przez użycie wbudowanej w KNIME funkcji próbkowania warstwowego (*Stratified Sampling*), dzięki czemu, po podziale danych, w obu

² Wynikało to z metodyki pracy w projekcie, która zakładała wprowadzanie znacznie mniejszych modyfikacji kodu źródłowego w „pomniejszych” wydaniach, tym samym mniejszy nakład pracy na testowanie, a co za tym idzie – mniejszą liczbę zarejestrowanych defektów.

zbiorach: uczącym i walidującym zachowano ten sam stosunek między modułami kodu sklasyfikowanymi w różny sposób.

4.3.2. Przeciwdziałanie niezbalansowaniu klas



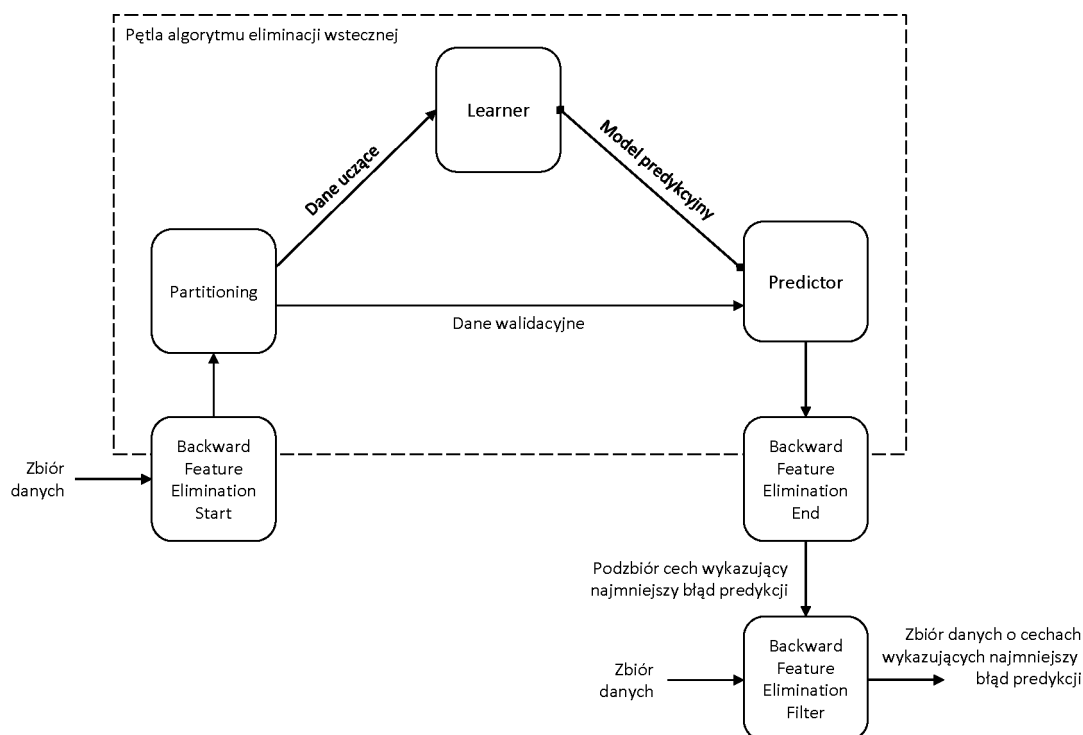
Rysunek 4.1: Diagram przepływu narzędzia DePress przeciwdziałający zjawisku niezbalansowania klas

W badaniach wykorzystano dwie wartości „0” i „1” aby oddzielić moduły podatne na defekty, czyli te, w których pojawił się co najmniej jeden defekt („1”), od tych pozbawionych defektów („0”). Jak wspomniano w punkcie 4.3.1, do badań wybrano te wersje (wydania) aplikacji, gdzie spodziewano się najmniejszego niezbalansowania klas. Jednak w przypadku, gdyby takowe niezbalansowanie – nawet nieznaczne – wystąpiło, zdecydowano się na rozwiązanie polegające na losowym usunięciu niektórych instancji z klasy większościowej.

Podążając za pierwotnym założeniem, wykorzystano mechanizm wbudowany w DePress, konstruując diagram przepływu danych tak, jak przedstawiono to na rys. 4.1. Najpierw zbiór danych jest dzielony na dwie części poprzez klasyfikację wierszy dwóch zbiorów w zależności od wartości „1” lub „0” przy wykorzystaniu węzła *Row Splitter* (rozdzielacz wierszy). Następnie, instancje klasy większościowej zostają zredukowane do osiągnięcia dokładnie tej samej liczby instancji klasy mniejszościowej, poprzez próbkowanie losowe klasy większościowej za pomocą węzła *Row Sampling* (próbkowanie wierszy). Na końcu, dwa takie same pod względem rozmiaru zbiory danych zostały połączone przez węzeł *Concatenate* (konkatenacja) w celu utworzenia zbioru danych z taką samą liczbą rekordów obu klas („1” i „0”).

4.3.3. Proces wyboru cech

Jak wspomniano w rozdziale 2, zbiór danych, który ma być wykorzystany do uczenia maszynowego, może być zmodyfikowany w celu minimalizacji błędu klasyfikacji. W celu osiągnięcia założonych celów na tym etapie badań, zdecydowano o użyciu techniki wyboru cech, opisanej w punkcie 2.6.2.



Rysunek 4.2: Diagram przepływu narzędzia KNIME realizujący mechanizm wyboru cech

W tym konkretnym przypadku, celem jest uzyskanie takiego podzbioru typów metryk ze wszystkich możliwych typów metryk zebranych przez narzędzie Eclipse Metrics 2 (tabela 4.2), którego użycie będzie skutkowało najmniejszym błędem predykcji.

Platforma KNIME posiada wbudowaną funkcjonalność wyboru cech bazującą na algorytmie eliminacji wstecznej [105]. Funkcjonalność ta składa się z pętli zbudowanej przy wykorzystaniu dwóch węzłów (rys. 4.2): *Backward Feature Elimination Start* i *Backward Feature Elimination End*. W pętli stworzonej przez te węzły powinien być ustanowiony odpowiedni mechanizm uczenia maszynowego – odpowiednio: węzeł uczący (Learner) i predyktor (Predictor). Do podzielenia wprowadzanych danych na część uczącą i walidacyjną został wykorzystany węzeł dzielący zbiory (Partitioning).

Metoda eliminacji wstecznej wykorzystana w narzędziu KNIME jest przeprowadzana w następujący sposób:

1. W pierwszej iteracji pętla jest wykonana dla wszystkich cech (kolumn): zbiór danych jest dzielony na dwa podzbiory, model jest budowany przez węzeł uczący wykorzystujący pierwszy zbiór, a następnie podlega walidacji w predyktorze, który wykorzystuje drugi zbiór. Sposób dzielenia zbiorów użytkownik ustala sam – powinien on być zgodny ze

sposobem dzielenia zbiorów na uczący i walidacyjny przyjętym we właściwym procesie uczenia maszynowego (tzn. spoza mechanizmu wyboru cech).

2. W każdej kolejnej iteracji jedna z cech (jedna kolumna tabeli) jest wyłączona z powyższego procesu (jedna cecha na iterację, za każdym razem inna), czyli tworzenie modelu i jego walidacja, jest przeprowadzana analogicznie jak w pierwszej iteracji, ale bez jednej cechy (kolumny) z oryginalnego zbioru. Dla każdej iteracji mierzony jest błąd predykcji, wyrażony w odsetku wierszy, dla których rezultat przewidywany i rzeczywisty mają różne wartości.
3. Gdy proces tworzenia modelu i jego walidacji zostanie przeprowadzony dla każdego z podzbiorów „pomniejszonych” o jedną kolumnę względem zbioru oryginalnego, węzeł *Backward Feature Elimination End* odrzuca z „oryginalnego” zbioru kolumnę mającą najmniejszy wpływ na wynik predykcji.
4. Punkty 2 i 3 są powtarzane, dopóki nie pozostanie jedna cecha (kolumna).

Na zakończenie do węzła *Backward Feature Elimination Filter* przekazywana jest informacja o zestawieniu cech (podzbiorze kolumn tabeli) wyłonionym w wyniku powyższego procesu, który skutkował najniższym błędem predykcji. Informacja ta może zostać następnie wykorzystana przez użytkownika KNIME w celu usunięcia z rzeczywistego zbioru danych tych cech (kolumn tabeli), które mogą nic nie wnosić, bądź mieć negatywny wpływ na wyniki predykcji.

4.3.4. Modele predykcji

Jako, że narzędzie DePress bazuje na rozbudowanej platformie do eksploracji danych KNIME, na potrzeby niniejszych badań mogły zostać wykorzystane różne typy predyktorów podatności na defekty. Ponieważ wyniki predykcji są odnoszone na skali nominalnej (tzn. są kategoryczne i przyjmują wartości³ „1” oraz „0”, inaczej: „narażony na defekty” i „nie narażony na defekty”), zdecydowano się na wykorzystanie następujących klasyfikatorów, z uwagi na ich częste wykorzystywanie (z powodzeniem) w predykcji defektów w oprogramowaniu [67, 143, 94, 182] oraz ze względu na dostępność w podstawowej instalacji KNIME:

- naiwny klasyfikator Bayesa,
- probabilistyczna sieć neuronowa⁴,
- drzewo decyzyjne.

Więcej informacji o klasyfikatorach wbudowanych w KNIME można znaleźć w dokumentacji programu [105].

³ Ponieważ jest to skala nominalna, pomimo używania wartości liczbowych 0 i 1 na skali tej nie definiuje się relacji porządku.

⁴ Probabilistyczna sieć neuronowa – rodzaj algorytmu uczenia maszynowego, który jako wynik swojego działania przedstawia nie pojedynczą klasę, ale rozkład prawdopodobieństwa przynależności do każdej z klas; jest to istotna cecha w porównaniu do algorytmu drzewa decyzyjnego, który wskazuje bezpośrednio na konkretną klasę.

4.4. Metoda oceny efektywności kosztowej

W celu określenia efektywności kosztowej zastosowania techniki predykcji defektów w komercyjnym projekcie wytwarzania oprogramowania przy wykorzystaniu narzędzia KNIME i jego rozszerzenia DePress, przyjęto poniższy plan:

1. Stworzenie strategii alokacji działań zapewniania jakości, bazującej na predykcji defektów dostarczonej przez DePress;
2. Analiza rzeczywistych kosztów zapewniania jakości dla wybranej wersji (4.0.0);
3. Przeprowadzenie procesu predykcji defektów z wykorzystaniem algorytmów uczenia maszynowego dostępnego w narzędziu KNIME, wyszczególnionych w punkcie 4.3.4;
4. Zbadanie, jak wysoką efektywność predykcji można osiągnąć obserwując takie miary jak *czułość* oraz *miara F*;
5. Zastosowanie strategii alokacji działań, wykorzystując predykcję charakteryzującą się najwyższą wartością czułości, w celu zasymulowania scenariusza wykorzystania predykcji w zapewnianiu jakości (i kosztów tejże);
6. Porównanie rzeczywistych (bez wykorzystania narzędzia DePress i predykcji) i symulowanych (z wykorzystaniem DePress i predykcji) kosztów zapewniania jakości.

4.4.1. Strategia planowania działań w procesie zapewniania jakości

W badanym przypadku (wersja 4.0.0 oprogramowania) programiści zgodzili się, że wszystkie moduły kodu źródłowego, w których wystąpiły dwa lub więcej zarejestrowane defekty będą uważane za moduły „wysokiego ryzyka”. Moduły te stanowiły 22,4% wszystkich modułów i były odpowiedzialne za 80,36% wszystkich zarejestrowanych defektów. Podobny rozkład defektów w modułach oprogramowania był obserwowany przez różnych autorów [42, 168, 180] i może być interpretowany jako potwierdzenie występowania zasady Pareto w obszarze jakości oprogramowania. Ponadto, w 1976 roku Boehm argumentował, że koszty usuwania defektów są tym droższe, im później usuwane są defekty [20]. Z tej obserwacji, która powszechnie nazywana jest zasadą (czasami błędnie: „prawem”) Boehma [42], wynika kolejna ważna konsekwencja inteligentnej alokacji wysiłków związanych z zapewnianiem jakości: im wcześniej będą miały miejsce działania zapewniania jakości, tym niższy będzie całkowity koszt projektu wytwarzania oprogramowania.

Biorąc pod uwagę powyższe fakty, zaproponowano strategię, która wykorzystuje model predykcji do wskazania jak największej liczby wspomnianych modułów oprogramowania o wysokim ryzyku (w naszym przypadku 22,4%) odpowiedzialnych za większość defektów (w naszym przypadku 80,36%). Model może zatem pomóc w jak największym stopniu zintegrować wysiłki zapewniania jakości z fazą implementacji w cyklu życia wytwarzania oprogramowania, podczas gdy koszt usuwania defektów będzie nadal

stosunkowo niski. Takie podejście powinno w idealnym przypadku obniżyć całkowity koszt usuwania defektów w projekcie i generować oszczędności w całkowitym budżecie projektu [192].

Jeżeli M_{total} oznacza łączną liczbę możliwych do testowania modułów programu, a H_{total} liczbę wykrytych defektów, to można powiedzieć, że „ $0,8H_{total}$ wynika z $0,22M_{total}$ ”.

Jeżeli przez Rec oznaczymy zmierzoną wartość czułości odpowiadającą najwyższej możliwej mierze F predykcji defektów wykonanej przy użyciu narzędzia KNIME z rozszerzeniem DePress z domyślnymi ustawieniami [121], to zachodzi:

$$0 < Rec < 1, \quad (4.1)$$

Wówczas oczekiwana liczba modułów M_i spośród 22% odpowiedzialnych za 80% możliwych do odkrycia defektów powinna wynosić:

$$M_i = 0,22 \times Rec \times M_{total}. \quad (4.2)$$

W związku z tym można oczekiwać, że jeśli mechanizm uczenia maszynowego będzie w stanie wskazać moduły oprogramowania o wysokim ryzyku (22%) osiągając przy tym poziom czułości (Rec), to oczekiwana liczba defektów, których można uniknąć poprzez alokację najlepszych wysiłków w zakresie zapewniania jakości w pierwszej fazie projektu, wyniesie:

$$H'_1 = 0,8 \times Rec \times H_{total}. \quad (4.3)$$

Oczekiwana liczba pozostałych (niewykrytych przez model w fazie pisania kodu) defektów, które będą musiały być naprawione w fazie testowania i utrzymania, wyniesie zatem:

$$H'_{2+3} = H_{total} - H'_1. \quad (4.4)$$

4.4.2. Zwrot z inwestycji

Aby sprawdzić czy wykorzystanie narzędzia DePress będzie opłacalne, wykorzystano współczynnik zwrotu z inwestycji (*Return on Investment; ROI*):

$$ROI = \frac{Zysk - Inwestycja}{Inwestycja}. \quad (4.5)$$

Jeżeli inwestycja nie będzie opłacalna, to wskaźnik ROI będzie negatywny, w przeciwnym przypadku – pozytywny. W ocenie efektywności kosztowej predykcji defektów skupiono się na potencjalnych zyskach, które predykcja defektów może przynieść:

$$Zysk = C_{total} - C'_{total}, \quad (4.6)$$

gdzie C_{total} to rzeczywiste koszty zapewniania jakości bez wykorzystania predykcji defektów, a C'_{total} to symulowane łączne koszty zapewniania jakości w projekcie wykorzystującym predykcję defektów. Co dokładnie składa się na koszty C_{total} oraz C'_{total} zostanie opisane w punkcie 4.5.

Inwestycja jest określana jako łączny koszt wdrożenia predykcji. Dodatkowo, definiujemy *ZwrotNetto* jako zysk pomniejszony o inwestycję:

$$ZwrotNetto = C_{total} - (C'_{total} + Inwestycja) = Zysk - Inwestycja. \quad (4.7)$$

4.4.3. Współczynnik kosztów i korzyści

Aby przeanalizować potencjalne zyski wynikające z użycia predykcji defektów, wykorzystany zostanie stosunek kosztów inwestycji do zysku (*Benefit-Cost Ratio; BCR*):

$$BCR = \frac{Zysk}{Inwestycja}. \quad (4.8)$$

Wartości BCR większe niż 1 oznaczają zysk finansowy wynikający z wykorzystania predykcji bazującej na DePress. Wartości mniejsze niż 1 oznaczają stratę.

4.5. Koszta procesu zapewniania jakości – rzeczywiste i symulowane

4.5.1. Rzeczywiste koszty zapewniania jakości w projekcie

Ponieważ polityka firmy-partnera nie pozwala na publikację rzeczywistych kosztów, do celów tej pracy oznaczono koszty pracy programisty oprogramowania C_d jako:

$$C_d = x. \quad (4.9)$$

W takim przypadku, średni koszt roboczogodziny testerów oprogramowania C_t został, zgodnie z ówczesnymi danymi na temat kosztów robocizny [196], oszacowany jako:

$$C_t = 0,85x. \quad (4.10)$$

Oznacza to, że kiedy tester i programista pracują wspólnie nad naprawą defektów podczas późniejszych faz projektowych (nie w fazie kodowania), to wtedy przeciętny koszt roboczogodziny powinien wynieść:

$$C_{d+t} = \frac{C_d + C_t}{2} = 0,925x. \quad (4.11)$$

Inne koszty, takie jak koszt infrastruktury i sprzętu, pozostaną stałe zarówno dla rzeczywistych jak i alternatywnych (bazujących na predykcji) scenariuszy, dlatego też zostaną w dalszych analizach pominięte.

Czas poświęcony na prace projektowe był rejestrowany przez każdego członka zespołu za pomocą narzędzia JIRA. W wyniku analizy tych danych można było uzyskać średnią wartość całkowitego czasu poświęconego na usunięcie pojedynczego defektu dla każdej fazy projektu (tabela 4.3). Czas spędzony na kontrolę jakości razem z liczbą godzin i liczbą naprawionych defektów podzielony na fazy został przedstawiony w tabeli 4.4.

Zgodnie z danymi przedstawionymi w tabeli 4.4 łączna liczba H_{total} defektów wykrytych w wersji 4.0.0 to

$$H_{total} = \sum_{phase=1}^3 H_{phase} = 190 + 383 + 264 = 837, \quad (4.12)$$

gdzie $phase$ oznacza fazę cyklu życia (1 = implementacja, 2 = testowanie, 3 = utrzymanie), a $H_i, i = 1, 2, 3$ – liczbę defektów wykrytych w fazie i .

Tabela 4.3: Średnie koszty naprawy defektu

Faza projektu	1. Pisanie kodu	2. Testowanie	3. Utrzymanie
Zaangażowani członkowie zespołu	Programista	Programista Tester	Programista Tester
Średni czas naprawy jednego defektu [godziny], T	1	3	3
Zakładany koszt roboczogodziny C_{hour}	C_d	C_{d+t}	C_{d+t}
Koszt naprawy defektu $C_{defect} = T \times C_{hour}$	x	$2,775x$	$2,775x$

Tabela 4.4: Rzeczywiste zasoby wykorzystane do naprawy defektów (QA = zapewnianie jakości)

Faza projektu	1. Pisanie kodu	2. Testowanie	3. Utrzymanie
Liczba wykrytych defektów H_{phase}	190	383	264
Koszt QA na jeden defekt C_{defect}	x	$2,775x$	$2,775x$
Koszt QA na fazę $C_{phase} = H_{phase} \times C_{defect}$	$190x$	$1063x$	$733x$

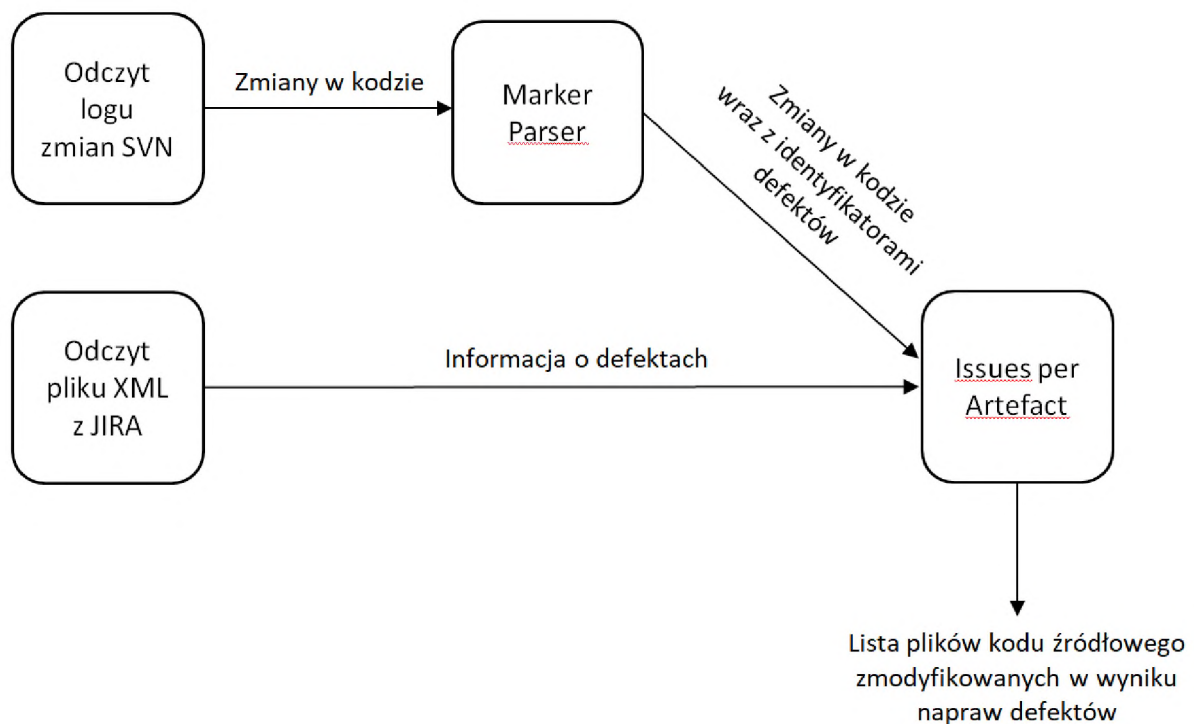
Zatem łączne rzeczywiste koszty zapewniania jakości to:

$$C_{total} = \sum_{phase=1}^3 C_{phase} = 190x + 1063x + 733x = 1985x. \quad (4.13)$$

Stosunek między liczbą defektów znalezionych podczas testów a liczbą tych, które znaleziono po wypuszczeniu oprogramowania można przyjąć jako:

$$\frac{H_2}{H_3} = \frac{383}{264} \approx \frac{3}{2}. \quad (4.14)$$

4.5.2. Konstrukcja modelu i predykcja



Rysunek 4.3: Diagram przepływu narzędzia DePress dostarczający informacji o rozkładzie zmian w kodzie źródłowym, które nastąpiły w wyniku napraw defektów.

Aby stworzyć model predykcji, który będzie w stanie sklasyfikować nowy kod źródłowy, proces uczenia maszynowego musi być przeprowadzony przy wykorzystaniu już skategoryzowanych danych uczących i wziętych ze wcześniejszych wersji programu, gdzie obszar „wysokiego ryzyka” może być z łatwością zidentyfikowany poprzez sprawdzenie, które części kodu uległy zmianom w wyniku naprawy defektów. W przypadku projektu Xates, jak zostało to scharakteryzowane w rozdziale 3, wszystkie zmiany w kodzie wywołane poprawkami zostały oznaczone unikatowymi identyfikatorami defektów, pobranymi z narzędzia śledzenia defektów używanego w projekcie (JIRA). Oba narzędzia – system kontroli wersji jak i aplikacja śledzenia defektów – umożliwiały zdobycie potrzebnych danych przez eksport ich do pliku.

Narzędzie DePress posiada odpowiednie węzły pozwalające na konwersję danych z obu tych programów do jednego, uniwersalnego wewnętrznego formatu. Wszystkie zarejestrowane zmiany kodu źródłowego będące wynikiem poprawek defektów są rozpoznane przez węzeł *Marker Parser* (analizator składni komentarzy „commitów”). Te wpisy są następnie porównywane z danymi z systemu śledzenia defektów przez węzeł *Issues per Artifact* (defekty na artefakt) (rys. 4.3). Efektem wykorzystania tego węzła są statystyki opisujące liczbę zmian będących wynikiem poprawek unikalnych defektów dla każdego z modułów. Dzięki wyżej wspomnianemu porównaniu łatwo jest rozpoznać najbardziej defektogenne obszary kodu, które można sklasyfikować jako obszary „wysokiego ryzyka”. W badanym przypadku (wydanie 4.0.0), jako obszar „wysokiego ryzyka” wskazane zostały wszystkie moduły, które musiały być zmodyfikowane, aby naprawić co najmniej 2 zarejestrowane, unikalne defekty. Przypomnijmy, że moduły te stanowią łącznie 22,4% wszystkich modułów i były odpowiedzialne za 80,36% wszystkich zarejestrowanych defektów.

Metryki kodu zebrane dla każdego modułu były sklasyfikowane zgodnie z zastosowaną metodą: moduły „wysokiego ryzyka” zostały oznaczone jako „1”, a te nie należące do tej grupy jako „0”. Stosunek modułów „1” do tych oznaczonych jako „0” to 1 : 15,52, co wskazuje na problem niezrównoważenia klas.

Skategoryzowane dane zostały podzielone losowo na dwa takie same zbiory podobnie, jak zostało to przedstawione na rysunku 4.2, przez moduł *Partitioning* z użyciem funkcji próbkowania warstwowego (*Stratified Sampling*) opisanej w punkcie 4.3.1 – w obu podzbiorach zachowano ten sam stosunek między modułami kodu odmiennie sklasyfikowanymi. Jeden ze zbiorów został zapisany w celu późniejszej walidacji modeli predykcji, drugi z kolei został wykorzystany do przygotowania modeli predykcji z wykorzystaniem trzech wybranych klasyfikatorów – naiwnego klasyfikatora Bayesa, drzewa decyzyjnego i probabilistycznej sieci neuronowej.

Dla każdego klasyfikatora, dzięki wykorzystaniu narzędzia DePress bazującego na modułowej platformie KNIME, było możliwe wybranie czterech różnych zestawów eksperymentalnych:

- bez doboru cech, z zestawem niezbalansowanych klas,
- bez doboru cech, z zestawem zbalansowanych klas,
- z doбором cech, z zestawem niezbalansowanych klas,
- z doбором cech, z zestawem zbalansowanych klas.

Gdy było to konieczne, wykorzystywano mechanizm wyboru cech przy pomocy funkcjonalności KNIME przedstawionej w punkcie 4.3.3, natomiast zrównoważenie klas zostało osiągnięte przez użycie mechanizmu zaprezentowanego w punkcie 4.3.2.

Tabela 4.5: Wyniki predykcji: Czułość i miara F (w nawiasach) dla wszystkich zestawów eksperymentalnych

Klasyfikator	Bez doboru cech		Z doborem cech	
	Niezbalansowanie klas	Zbalansowanie klas	Niezbalansowanie klas	Zbalansowanie klas
Naiwny Bayes	0,409 (0,237)	0,783 (0,621)	0,318 (0,412)	0.783 (0.766)
Drzewo decyzyjne	0,273 (0,279)	0,682 (0,667)	0,227 (0,357)	0,652 (0,682)
Probabilistyczna sieć neuronowa	0,091 (0,167)	0,783 (0,72)	0,136 (0,24)	0,783 (0,74)

4.5.3. Najwyższa wartość miary F i odpowiadająca jej wartość czułości predykcji

Wykorzystanie podejścia przedstawionego w poprzednim punkcie pozwoliło na przeprowadzenie predykcji i zmierzenie wartości miary F (tabela 4.5) we wszystkich czterech zestawach eksperymentalnych, klasyfikatorach i próbkach. Najlepsze wyniki predykcji (najwyższa wartość miary F wynosząca 0,766) zostały osiągnięte przy użyciu naiwnego klasyfikatora Bayesa, gdy wykorzystane zostały dodatkowe techniki wyboru cech oraz zbalansowania klas. Odpowiadająca temu wynikowi wartość czułości (*recall*) wynosiła:

$$Rec = 0,783. \quad (4.15)$$

4.5.4. Symulacja kosztów predykcji

Do przeprowadzenia symulacji kosztów w przyjętym, alternatywnym scenariuszu wytwarzania oprogramowania, zakładającym, że wykorzystano predykcję defektów za pomocą DePress przyjęto następujące założenia:

- łączna liczba wykrywalnych defektów w wersji 4.0.0, określona wzorem (4.12), jest wartością stałą,
- rozkład defektów w kodzie jest zachowany, tj. w symulowanym scenariuszu defekty występują w tych samych miejscach w kodzie, w jakich zostały zaobserwowane w rzeczywistości,
- średni koszt naprawy jednego defektu przyjmuje wartości opisane w tabeli 4.3,
- pochodząca z predykcji defektów informacja o lokalizacji modułów oprogramowania o „wysokim ryzyku” (charakteryzująca się czułością *Rec*) będzie dostępna w pierwszej fazie projektu,

— stosunek między liczbą defektów znajdujących w fazie testów a liczbą znajdujących w fazie utrzymania, wyrażony równaniem (4.14) jest zachowany.

Biorąc pod uwagę wartość czułości Rec dla najlepszej (charakteryzującej się najwyższą wartością miary F) predykcji uzyskanej dla wersji 4.0.0 w wyniku opracowania modeli predykcyjnych (równanie (4.15)) oraz całkowitą liczbę wykrytych defektów w tej wersji (równanie (4.12)), w oparciu o zaproponowaną strategię (równanie (4.3)) należy oczekiwać, że liczba defektów oprogramowania, które mogą być rozwiązane przez zastosowanie najlepszych praktyk zapewniania jakości w fazie realizacji projektu wynosi:

$$H'_1 = 0,8 \times 0,783 \times 837 = 524. \quad (4.16)$$

Biorąc pod uwagę szacowaną liczbę defektów, które mogą pojawić się w późniejszych fazach projektu (równanie (4.4)) przyjmujemy:

$$H'_{2+3} = 837 - 524 = 313. \quad (4.17)$$

Jak założono, stosunek z równania (4.14)

$$H'_2 = 313 \times 0,6 = 188. \quad (4.18)$$

Analogicznie, szacowana liczba defektów w fazie utrzymania wynosi

$$H'_3 = 313 \times 0,4 = 125. \quad (4.19)$$

Biorąc pod uwagę powyższe wartości, przeprowadzono symulację kosztów zapewniania jakości przy założeniu, że mechanizm uczenia maszynowego będzie w stanie wskazać elementy kodu źródłowego odpowiedzialne za 80% wykrywalnych defektów, z czułością wynikającą ze wzoru (4.15) i że największe wysiłki w zakresie zapewniania jakości będą przeznaczone na fazę programowania po to, aby uniknąć szacowanej liczby defektów (4.16). Rezultaty tej symulacji są przedstawione w tabeli 4.6.

Podsumowując, łączne koszty zapewniania jakości C'_{total} w tym scenariuszu (uwzględniając koszty naprawy defektu C_{defect} dla poszczególnych faz opisane w tabeli 4.3) wyniosą

$$C'_{total} = \sum_{phase=1}^3 C'_{phase} = 524x + 522x + 347x = 1393x, \quad (4.20)$$

gdzie C'_{phase} ($phase = 1, 2, 3$) to odpowiednio koszty zapewniania jakości w poszczególnych fazach: implementacji, testowania oraz utrzymania.

Tabela 4.6: Symulowane koszty zapewnienia jakości przy założeniu, że zastosowano predykcję defektów (QA = zapewnianie jakości).

Faza projektu	1. Pisanie kodu	2. Testowanie	3. Utrzymanie
Liczba naprawionych defektów H'_{phase}	525	188	125
Koszty QA na jeden defekt C_{defect}	x	$2,775x$	$2,775x$
Koszty QA na fazę $C'_{phase} = H'_{phase} \times C_{defect}$	$524x$	$522x$	$347x$

4.5.5. Koszta inwestycji

Koszta wdrożenia predykcji zostały obliczone jako suma kosztów elementarnych, w skład których wchodzi:

- łączne koszty zakupu i instalacji oprogramowania,
- koszty czasowe potrzebne na szkolenie,
- koszty zebrania danych,
- koszty przygotowania predykcji.

Łączne koszty zakupu i instalacji oprogramowania w badanym przypadku powinny być ocenione na zero. W firmie, w której przeprowadzano badania, DePress jest dostępny za darmo poprzez wewnętrzną aplikację instalacyjną. DePress może być zamówiony i zainstalowany na komputerze użytkownika bez ponoszenia żadnych kosztów przez użytkownika bądź projekt. Jeśli narzędzie to nie będzie – z jakiś powodów – niedostępne poprzez wewnętrzną aplikację, można je pobrać darmowo z Internetu.

Koszt czasu potrzebnego na szkolenie – wsparcie predykcji defektów i narzędzia DePress jest dostarczane przez wewnątrz-firmową organizację szkoleniową, której działalność jest opłacana przez projekty rozwoju oprogramowania, niezależnie od tego, czy dane narzędzie i technologia będzie wykorzystana, czy też nie. Zatem koszty szkoleniowe ograniczają się do kosztu roboczo-godzin każdego członka zespołu projektowego (dla uproszczenia przyjmijmy, że będzie to programista) będącego odpowiedzialnym za proces predykcji defektów. Po zmierzeniu czasu potrzebnego na szkolenie jednej osoby (za pomocą narzędzi rejestrujących czas pracy) można stwierdzić, że jeśli osoba odpowiedzialna za przeprowadzenie predykcji defektów ma minimalne doświadczenie w rozwoju oprogramowania, wtedy potrzebuje maksymalnie 4 godzin na szkolenie: 1-2 godzin generalnego wprowadzenia plus 1-2 godzin szkolenia w przygotowaniu predykcji za pomocą narzędzia DePress. Tak krótki czas potrzebny na przyswojenie sobie narzędzia wynika – jak zostało już to wcześniej zaznaczone – z jego łatwego do nauczenia się graficznego interfejsu użytkownika.

Koszta zbierania danych to przede wszystkim koszt roboczogodzin związany z wyeksportowaniem odpowiednich danych ze źródeł danych i wygenerowaniem metryk kodu dla dwóch wybranych wersji. Eksport danych nie zajmie więcej niż jedną roboczogodzinę, a w większości przypadków czynność ta powinna zająć nie więcej niż kilka minut. Co więcej, można wykorzystać węzły bezpośredniego połączenia dostępne w DePress do uzyskania wymaganych danych w czasie rzeczywistym. Dla celów zbierania metryk kodu przy użyciu wtyczki *Eclipse Metrics 2* oprogramowanie musi być kompilowane lokalnie, co obejmuje: pobranie wersji oprogramowania ze zdalnego repozytorium, kompilację aplikacji i zbieranie plików XML. Po zmierzeniu czasu poświęconego na tę czynność okazało się, że zajmuje ona nie więcej niż 2 roboczogodziny. Koszt przygotowania predykcji defektów to koszt roboczogodziny przygotowania diagramu przepływu danych predykcji defektów przy użyciu narzędzia DePress. Ponieważ generyczne diagramy przepływu danych (dla większości technologii wykorzystywanych w firmie) są preinstalowane z narzędziem DePress, oczekuje się jedynie działań korygujących, jeśli takie występują. Wyniki pomiaru czasu wskazują, że przygotowanie poprawnego diagramu przepływu danych DePress nie powinno zająć więcej niż jedną roboczogodzinę.

Podsumowanie kosztów inwestycji przedstawiono w tabeli 4.7.

Tabela 4.7: Koszty inwestycji w predykcję defektów (w projekcie)

Activity	Wymagany czas [godziny]	Koszt [roboczogodziny]
Pozyskanie narzędzia DePress i koszty instalacji	0	0
Nauka korzystania z narzędzia	4	$4x$
Zbieranie danych	3	$3x$
Przeprowadzenie predykcji	1	x
RAZEM (Inwestycja)	8	$8x$

4.6. Wyniki

Czy zastosowanie predykcji defektów oprogramowania przy użyciu narzędzia DePress może przynieść korzyści dla komercyjnego projektu wytwarzania oprogramowania? Symulacja pokazuje, że można oczekiwać wymiernych korzyści. Stosując równanie (4.6) otrzymujemy

$$Zysk = 1985x - 1393x = 592x. \quad (4.21)$$

Na podstawie równania (4.5) możemy przewidzieć, że zwrot z inwestycji wyniesie

$$ROI = \frac{592x - 8x}{8x} = 73. \quad (4.22)$$

Taki wynik ROI nie tylko wskazuje jednoznacznie, że inwestycja się opłaci (ROI większe od zera), sama wartość obliczonego ROI jest bardzo duża i pozwala przypuszczać, że w idealnym scenariuszu (gdy wszystkie teoretyczne założenia, przedstawione w tym rozdziale pracy, zostaną spełnione) zwrot z inwestycji wynosić może nawet 7300%. Proszę zwrócić uwagę, że dla osiągnięcia tak wysokiego ROI zakładamy, że użyjemy względnie taniego (w pozyskaniu, wdrożeniu i utrzymaniu) narzędzia do predykcji defektów, którą to wykorzystywać będziemy w długim horyzoncie czasowym, włączając w to całą fazę utrzymania oprogramowania.

Jak efektywnie kosztowo jest użycie narzędzia DePress w predykcji defektów dla projektów oprogramowania komercyjnego i jaka jest najwyższa wartość czułości predykcji, możliwa do osiągnięcia przez to narzędzie? Jak pokazano w tabeli 4.7, oczekiwane łączne koszty inwestycji obejmujące zastosowanie predykcji defektów w oprogramowaniu bazującej na narzędziu DePress do lepszej alokacji zadań związanych z zapewnianiem jakości to

$$Inwestycja = 8x. \quad (4.23)$$

Stosunek zysku do kosztów (z równania (4.8)) obliczony przy wykorzystaniu równania (4.6) i równania (4.23) to

$$BCR = \frac{592x}{8x} = 74. \quad (4.24)$$

Wartość *ZwrotNetto* (równanie (4.7)) z symulowanej aplikacji predykcji to

$$ZwrotNetto = 592x - 8x = 584x. \quad (4.25)$$

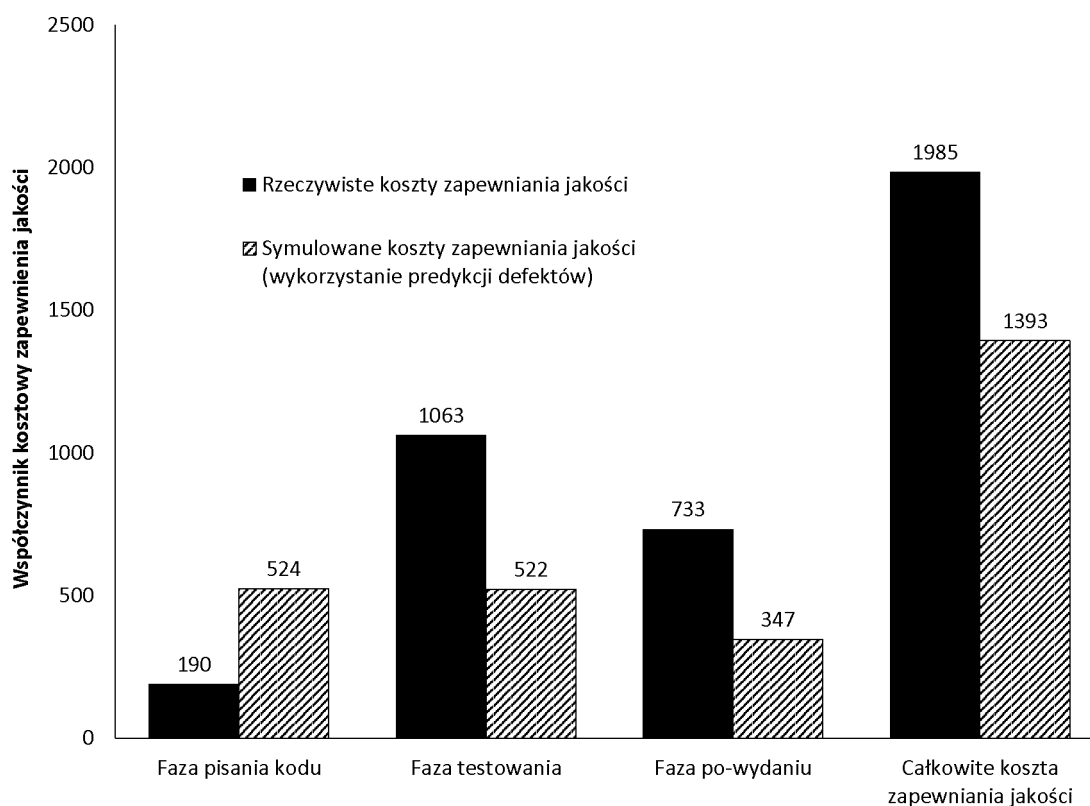
Taka wartość *ZwrotNetto* oraz wskaźnika BCR pokazuje, że można oczekiwać dużych profitów przy wykorzystaniu niedrogiego narzędzia, jakim jest DePress, do wsparcia zapewniania jakości przez predykcję defektów w bardzo długim horyzoncie czasowym.

W przypadku zastosowania strategii predykcji defektów zaproponowanej w punkcie 4.4.1, gdy *czułość* modelu predykcji wyniesie 0,783, takie podejście może skutkować obniżeniem ostatecznych kosztów zapewniania jakości o prawie 30%:

$$1 - \frac{C'_{total}}{C_{total}} = 1 - \frac{1393x}{1985x} = 0.298. \quad (4.26)$$

Taki wynik można osiągnąć dopiero po usunięciu 524 defektów (równanie (4.16)), co stanowi 62.6% wszystkich wykrywalnych defektów, poprzez efektywne wykorzystanie praktyk zapewniania jakości w pierwszej, rozwojowej fazie projektu, na przewidywanych modułach oprogramowania o wysokim ryzyku. Graficzne porównanie kosztów zapewniania jakości –

rzeczywistych i symulowanych, z wprowadzonym przewidywaniem defektów, pokazane jest na rys. 4.4.

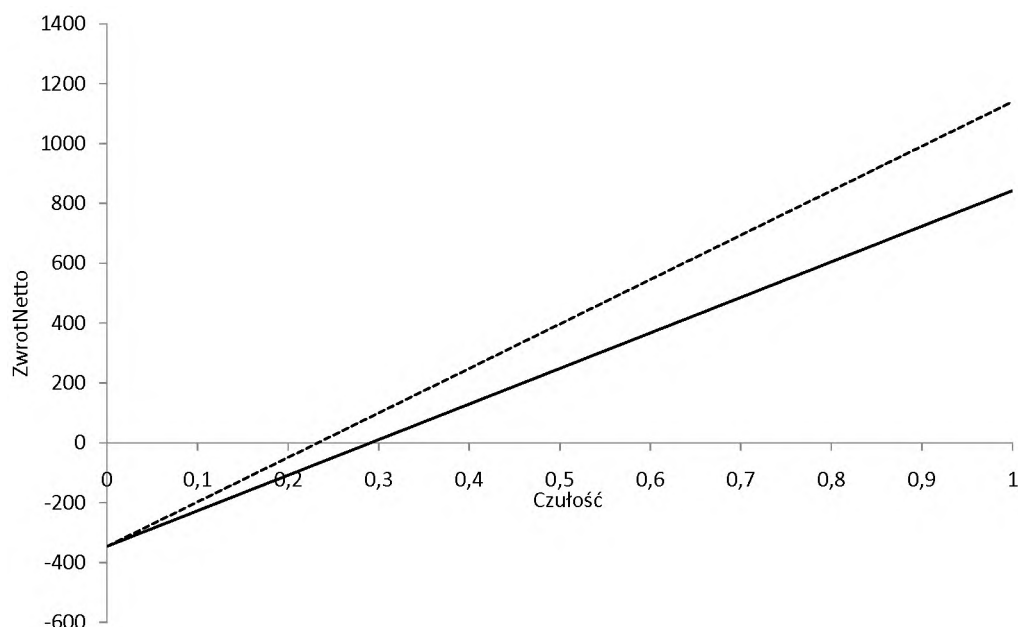


Rysunek 4.4: Koszty zapewnienia jakości w rozważanym projekcie: rzeczywiste w porównaniu z symulowanymi, gdy stosowane jest przewidywanie defektów na podstawie DePress.

4.7. Podsumowanie tego etapu badań

W powyższym badaniu defekty nie zostały podzielone zgodnie z ich dotkliwością (niekrytyczne, krytyczne etc.) i wykorzystują średni, stały czas potrzebny na naprawę każdego z nich. Pominięcie wskaźnika dotkliwości w badaniach nad predykcją defektów jest częstą praktyką [135], aczkolwiek może to być istotne, kiedy symulowane koszty zapewnienia jakości będą porównywane z wartościami rzeczywistymi. W symulacji założono, że taki sam poziom dotkliwości dla każdego defektu odzwierciedla jednakowy, uśredniony koszt (tabela 4.3) naprawy defektu. Aczkolwiek, kiedy wykorzystana jest proponowana alokacja wysiłku w środowisku rzeczywistym, to może zaistnieć sytuacja, w której defekty niewykryte aż do późniejszych faz projektu, to jest fazy testów i fazy utrzymania (po wdrożeniu aplikacji) będą charakteryzować się wyższym współczynnikiem dotkliwości niż defekty, które wykrywa się

i/lub eliminuje w fazie pisania kodu. Taka sytuacja mogłaby negatywnie wpłynąć na łączne, rzeczywiste koszty zapewniania jakości w porównaniu z symulowanymi wartościami.



Rysunek 4.5: Wartość współczynnika ZwrotNetto w zależności od *Czulości* predykcji

W symulacji koszty zapewniania jakości są obliczane bazując na łącznej liczbie defektów które, jak się zakłada, zostaną naprawione w każdej fazie projektowej. Liczba ta jest następnie pomnożona przez średnie koszty naprawy jednego defektu, bazujące na rzeczywistych danych projektu. Następnie symulowane koszty są porównywane z rzeczywistymi. Można dyskutować, czy efektywniejszą formą porównania kosztów zapewniania jakości, jeśli bierze się pod uwagę predykcję defektów oprogramowania, nie byłoby użycie opisanej w punkcie 2.2.1 miary AUCEC. Należy jednak zwrócić uwagę, że oczekiwaną konsekwencją pozytywnego wyniku badań na tym etapie było przekonanie przedstawicieli firmy-partnera o bezpośrednich korzyściach płynących z kontynuowania prac nad predykcją defektów, czyli otrzymanie zgody na faktyczne wdrożenie mechanizmu predykcji defektów jako ważnego elementu procesowego w projekcie wytwarzania komercyjnego oprogramowania. W projekcie – podkreślmy – wysoce nietrywialnym. Było to bezpośrednią motywacją do wykorzystania znanych i powszechnie stosowanych przez przedstawicieli biznesu technik oceny opłacalności, takich jak ROI.

Należy zaznaczyć, że w powyższym badaniu nie uwzględniono kosztów nie śledzonych przez system JIRA. Nie było technicznej możliwości na zdobycie informacji o kosztach działań w zakresie zapewniania jakości podjętych poza zespołem projektowym, np. w takich

przypadkach po wdrożeniu aplikacji do użytku, kiedy zadaniami z obszaru zapewniania jakości zajmował się jedynie dział obsługi klienta, odpowiedzialny za bezpośredni kontakt z użytkownikiem końcowym.

Do celów symulacji wykorzystane zostały te same warunki początkowe, co w rzeczywistym projekcie. Zawierały one tę samą liczbę wykrywalnych defektów jak i podobny stosunek pomiędzy defektami naprawionymi w fazie testów, jak i po wydaniu oprogramowania (równanie (4.4)). W rzeczywistej predykcji, przy wykorzystaniu proponowanej strategii alokacji wysiłków, należy oczekiwać, że wyżej wymienione warunki będą inne – na przykład, jeśli predykcja defektów zostanie zastosowana dla innej wersji oprogramowania lub innego projektu (dane nam się będzie wkrótce przekonać, jak efektywna jest predykcja defektów w innym projekcie wytwarzania oprogramowania).

Efektywność kosztowa w symulowanym scenariuszu jest ściśle powiązana z jakością predykcji opisywaną przez *czułość* (*Rec*). Bazując na zaprezentowanej symulacji, możliwe jest obliczenie wartości *ZwrotNetto* przez wykorzystanie proponowanej strategii alokacji wysiłków zapewniania jakości dla różnych możliwych wartości *czułości* modelu predykcji defektów (ciągła linia na rys. 4.5).

W ten sposób można zaobserwować, jak wskaźnik *ZwrotNetto* jest zależny od *czułości* *Rec* przy proponowanej strategii alokacji wysiłków zapewniania jakości. Oczywiście, jakakolwiek wartość współczynnika *ZwrotNetto* dla *Rec* niższa niż 0,5 nie powinna być brana pod uwagę, gdyż ta progowa wartość efektywności predykcji jest tożsama z losowym klasyfikowaniem modułów oprogramowania jako defektogenne bądź nie.

Dyskusyjnym pozostaje założenie, że programiści są w stanie przeciwdziałać 78% wszystkich możliwych do wykrycia defektów jeszcze w fazie pisania kodu, tylko ze względu na fakt, że defektogenne moduły będą znane programistom dzięki predykcji. Należy zauważyć, że w niniejszej symulacji wzięto pod uwagę jedynie defekty, które rzeczywiście zostały naprawione przez programistów. Jediną różnicą jest to, że w rzeczywistym projekcie, skąd wzięto dane do późniejszej symulacji, więcej defektów zostało znalezionych przez testerów i użytkowników końcowych w drugiej i trzeciej fazie projektu, co skutkowało większymi kosztami naprawy defektów w tych fazach. Niemniej jednak defekty były w końcu i tak naprawione właśnie przez programistów. Co więcej, w naszych rozważaniach (jeśli autor rozprawy wykazałby się niepoprawnym optymizmem) można by było pójść o krok dalej i zmodyfikować strategię alokacji zadań związanych z zapewnianiem jakości zakładając, że dostarczymy model predykcji, który wskaże kod odpowiedzialny za 100% wykrywalnych defektów. W takim przypadku relacja między szacowaną liczbą naprawionych defektów w fazie projektowania a *czułością* predykcji będzie opisana zależnością

$$H'_1 = Rec \times H_{total}. \quad (4.27)$$

Teoretycznie, w takim przypadku możemy oczekiwać, że wartość *ZwrotNetto* powinna być jeszcze wyższa niż obliczona w oparciu o proponowaną strategię (linia przerywana na rys. 4.5). Jednakże, ponieważ proponowana strategia alokacji wysiłku opiera się na zasadzie Pareto (tylko 20% kodu odpowiada za 80% defektów), powinno się konsekwentnie oczekiwać, że aby znaleźć i usunąć pozostałe 20% defektów, należałoby zbadać odpowiednio dodatkowe 80% kodu. W takim scenariuszu objęcie nawet 100% kodu wszelkimi niezbędnymi środkami w celu usunięcia każdego wykrywalnego defektu może kosztować znacznie więcej niż rzeczywiste, realne zapewnianie jakości w rozważanym wydaniu projektu (4.0.0). Należy się spodziewać, że pozostałe około 20% wykrywalnych defektów można skutecznie znaleźć i usunąć w dalszych fazach projektu, np. w fazie testowania, przy lepszej efektywności kosztowej, niż w przypadku kompleksowych prac nad zapewnianiem jakości w fazie kodowania, skoncentrowanych na znalezieniu i usunięciu 100% wykrywalnych defektów.

W komercyjnych projektach wytwarzania oprogramowania zapewnianie jakości (usuwanie defektów) pochłania znaczną ilość czasu i zasobów. Dzięki zastosowaniu techniki przewidywania defektów, członkowie projektu mogą uzyskać informacje o możliwych podatnych na defekty elementach oprogramowania, zanim defekty wystąpią, aby optymalnie zaplanować proces zapewniania jakości. To, co zostało zaproponowane w niniejszym rozdziale, to prosta strategia alokacji zadań (zapewniania jakości), która opiera się na predykcji defektów opartej na narzędziu DePress, zasadę Pareto oraz zasadę Boehma. Strategia ta eliminuje większość prac związanych z zapewnianiem jakości w fazach następujących po pisaniu kodu. Co prawda takie podejście znacząco zwiększa koszty zapewniania jakości w fazie programowania, jednak w ogólnym rozrachunku koszty zapewniania jakości spadną (patrz rys. 4.4) w porównaniu z rzeczywistymi, realnymi kosztami zaobserwowanymi w badanym projekcie, ponieważ znacznie mniej wykrywalnych defektów pozostaje do usunięcia w późniejszych fazach, gdzie zgodnie z zasadą Boehma koszty usuwania defektów są znacznie wyższe.

Jednocześnie należy wspomnieć o niskich kosztach inwestycyjnych w przypadku, gdy narzędzie DePress będzie wykorzystywane do predykcji defektów. Niskie koszty inwestycyjne oraz wysoki współczynnik *czułości* nawet prostych predykcji defektów wykonywanych przez DePress mogą skutkować wysoką wartością *ZwrotNetto* dla wspomaganego przez DePress zapewniania jakości planowanego w oparciu o proponowaną strategię alokacji zadań. Złożenie tych czynników spowodowało, że autor niniejszej rozprawy uzyskał zgodę firmy-partnera na dalsze kontynuowanie badań nad predykcją defektów w komercyjnym wytwarzaniu oprogramowania, w tym zgodę na wykorzystanie predykcji jako ważnego elementu w istotnym dla firmy projekcie. O tym traktuje kolejny rozdział niniejszej pracy.

Opisane w tym rozdziale badania pozwoliły zrealizować *piąty cel szczegółowy* nakreślony w punkcie 1.2 niniejszej pracy: Przeprowadzono symulację, w jaki sposób zmieniłyby się koszty

zapewniania jakości w wybranym komercyjnym projekcie rozwoju oprogramowania, gdyby zastosować predykcję defektów oprogramowania i w pełni skorzystać z jej wyników.

5. Metodyka zapewniania jakości w komercyjnym projekcie wytwarzania oprogramowania wykorzystująca predykcję defektów

5.1. Wprowadzenie

Pionierka informatyki, jedna z pierwszych osób na świecie zajmujących się programowaniem komputerów, Grace Murray Hopper, miała kiedyś powiedzieć, że „jeśli to dobry pomysł, zrób to. O wiele łatwiej jest przeprosić niż uzyskać pozwolenie” [169]. Także w przypadku niniejszych badań rada pani komandor Hopper była wyjątkowo kusząca – po prostu użyć predykcji defektów, aby uczynić rozwijane komercyjnie oprogramowanie jeszcze lepszym! Takie postępowanie nie byłoby jednak profesjonalne. Należało najpierw uzyskać akceptację kadry kierowniczej firmy. Jednakowoż, dzięki przeprowadzonym wcześniej badaniom opłacalności (a przede wszystkim – dzięki ich nad wyraz pozytywnym wynikom), jednocześnie niejako wbrew temu, co twierdziła Grace Hopper, udało się uzyskać akceptację i przychylność kierownictwa, a także pozwolenie na wdrożenie predykcji defektów jako istotnego elementu metodyki rozwoju oprogramowania w ważnym dla firmy projekcie.

Aby sprawdzić, jak predykcja defektów wpłynęła na jakość i stosowanie najlepszych praktyk programowania, zbadano jakość dwóch ostatnich wydań (wersji) oprogramowania, jednego stworzonego bez i jednego z predykcją defektów wspieraną przez narzędzie DePress, a następnie porównano oba wydania, używając tych samych kryteriów oceny jakości. Ponieważ cykl wydawniczy jednej wersji wynosił około roku, badania ważnego projektu komercyjnego opisane w tym rozdziale trwały ponad dwa lata.

5.2. Wybrany komercyjny projekt rozwoju oprogramowania

5.2.1. Charakterystyka projektu

Kierownictwo firmy współpracującej przy badaniach będących przedmiotem niniejszej pracy zdecydowało, że kolejny etap prac badawczych – weryfikacja predykcji defektów „in vivo” – odbędzie się w projekcie, którego celem jest rozwój i utrzymanie systemu

informatycznego o umownej nazwie ROPTIS. Znamienne, że był to kolejny projekt, który stał się przedmiotem badań będących tematem niniejszej pracy, po dotychczas badanych projektach „B”, „P”, „V” oraz szerzej opisywanym w rozdziale 4 projekcie Xates, czyli „T”. Jak wcześniej wspomniano, zaproponowanie nowego, *ważnego* projektu do badań, było następstwem obiecujących wyników badania opłacalności kosztowej predykcji defektów. Projekt określano jako *ważny*, ponieważ rozwijany weń system ROPTIS był (i w 2022 roku wciąż jest) wykorzystywany w fabrykach należących do wspomnianego wyżej przedsiębiorstwa.

ROPTIS jest systemem krytycznym dla produkcji, zaprojektowanym w architekturze klient-serwer. Jest on odpowiedzialny głównie za programowanie, testowanie, kalibrację i weryfikację montażu elektrycznego elektronicznych jednostek sterujących (ECU) w procesie produkcji pojazdów. Program dla Microsoft Windows, zwany „ROPTIS Operator”, komunikuje się w czasie rzeczywistym z aplikacją mobilną, znajdującą się na komputerze typu palmtop, używanym przez pracowników fabryki pojazdów, w celu przekazania wszystkich informacji związanych z produkcją do lokalnego serwera. Serwer jest odpowiedzialny za przechowywanie i dystrybucję danych dotyczących konfiguracji, systemu i produktu. Taka komunikacja może w dużych fabrykach generować ekstremalnie duże obciążenia transferem danych, gdy używanych jest ponad 100 aplikacji mobilnych. Dodatkowo system składa się z dwóch kolejnych programów Windows – „ROPTIS Designer” i „ROPTIS Factory Manager”, które są podłączone do serwera wraz z aplikacją internetową o nazwie „ROPTIS Viewer”.

Cykl życia systemu ROPTIS, zarówno w chwili prowadzenia badań, jak i obecnie (w 2022 roku), wygląda następująco: Każda pojedyncza wersja systemu ROPTIS, zwana wewnątrz projektu „wydaniem” jest dostępna dla użytkowników końcowych (techników odpowiedzialnych za montaż ECU w produkowanych pojazdach) przez okres jednego roku. W tym samym czasie, równolegle przygotowywane jest kolejne wydanie aplikacji. Ustalony czas trwania cyklu życia wydania oprogramowania (jeden rok) jest ściśle uzależniony od rocznego cyklu produkcyjnego pojazdów i nie może być zmieniony. Każde przygotowanie nowego wydania może być traktowane jako oddzielny projekt rozwoju oprogramowania z fazami programowania (pisanie kodu), testowania systemu i utrzymania. Jak wspomniano wcześniej, badania trwały ponad 2 lata, ponieważ musiały zostać zbadane dwie wersje systemu, zwane tutaj umownie wersją A i wersją B¹. Jednocześnie trzeba zaznaczyć, że termin „projekt rozwoju oprogramowania” na tym etapie odnosi się do projektu rozwoju wydania B systemu ROPTIS.

Wszystkie aplikacje w ramach systemu ROPTIS zostały opracowane przy użyciu technologii Microsoft .NET (język programowania C#) i Microsoft Visual Studio jako zintegrowanego środowiska programistycznego. Do celów kontroli wersji zastosowano

¹ Określenia *wersja* i *wydanie* będą używane wymiennie.

Microsoft Team Foundation Server. Podczas rozwoju wersji A systemu ROPTIS do śledzenia defektów stosowano aplikację IBM ClearQuest. Od rozwoju wersji B oprogramowanie do śledzenia defektów zostało zmienione na narzędzie Microsoft Team Foundation Server², przy czym w narzędziu tym przechowywane są te same informacje, co w ClearQuest.

5.2.2. Poziom jakości oprogramowania rozwijanego w ramach projektu

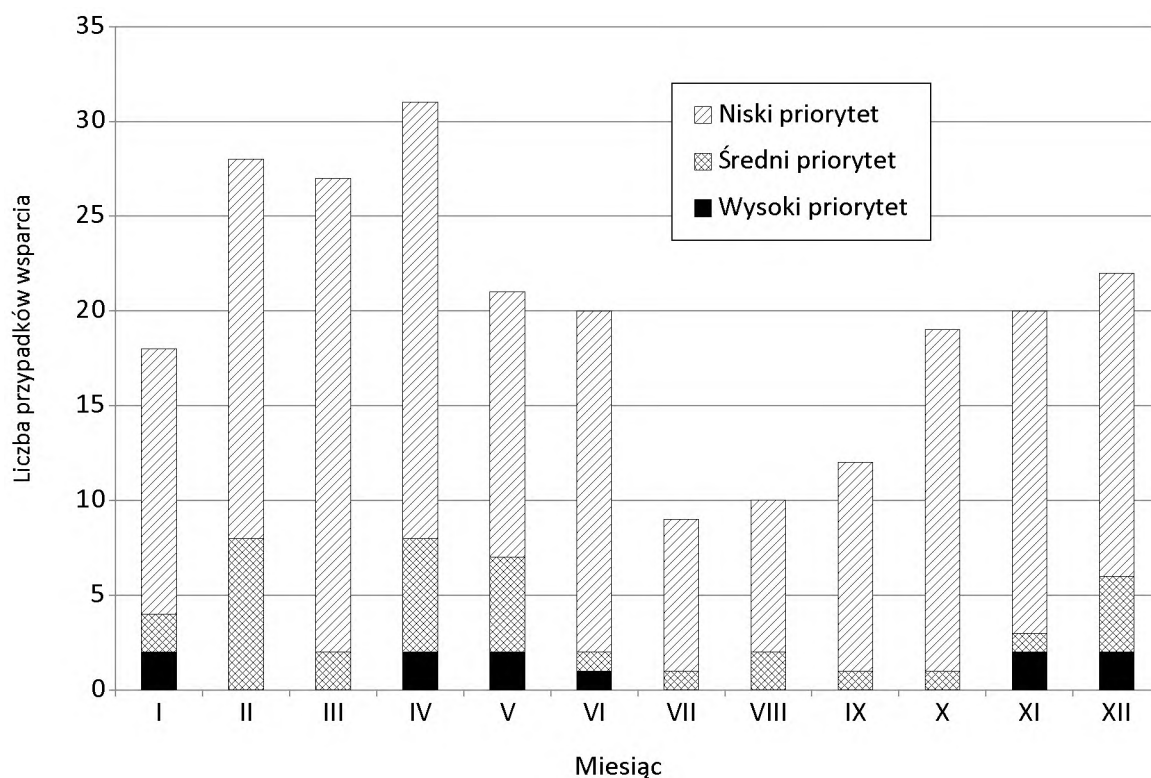
Tabela 5.1: Całkowita liczba defektów wykryta podczas czasu użytkowania wersji A programu ROPTIS

Dotkliwość defektu	Liczba
Krytyczna	162
Wysoka	188
Średnia	248
Mała	200
W tym: znalezione podczas testów systemowych	306
W tym: defekty naprawione niewłaściwie	83

Jakość wytwarzanych produktów (w tym oprogramowania) jest bardzo ważnym aspektem dla firmy – partnera niniejszych badań, szczególnie, jeśli chodzi o rozwój ważnych produktów, takich jak ROPTIS. Jakość jest tutaj rozumiana nie tylko jako niezawodność aplikacji, jej działania czy zarządzania kodem w zakresie samego systemu, lecz także jako efektywność procesu zapewniania jakości, w tym stosowanie najlepszych praktyk podczas naprawy defektów. Kiedy aplikacja zostanie już wydana, czyli gdy jest już dostępna dla użytkownika końcowego, wtedy najważniejszym czynnikiem jest niezawodność i wydajność wydanego systemu.

Przed wydaniem A system ROPTIS cierpiał z powodu dużej liczby poważnych defektów (ponad 160 krytycznych defektów, co powodowało wstrzymanie testów), problemów z wydajnością (cztery krytyczne i jeden poważny defekt związany z wydajnością znalezione późno w fazie testowania systemu), nieskutecznego usuwania defektów (83 zarejestrowane przypadki, kiedy poprawki defektów zostały odrzucone przez testerów) oraz niestabilnych pakietów instalacyjnych. Podsumowanie defektów znalezionych przed wydaniem A zostało przedstawione w tabeli 5.1. Po wydaniu wersji A do środowiska produkcyjnego jakość systemu mogła być mierzona za pomocą zgłoszeń do pomocy technicznej składanych przez użytkowników końcowych. W zależności od wagi zgłoszonego problemu zgłoszenie było później zarejestrowane jako „przypadek wsparcia” i skategoryzowane poprzez przypisanie

² W roku 2019 narzędzie Microsoft Team Foundation Server zmieniło nazwę na Microsoft Azure DevOps Server.



Rysunek 5.1: Liczba przypadków wsparcia dla wersji A systemu ROPTIS

priorytetu: wysoki, średni i niski, gdzie przypadki wsparcia o wysokim priorytecie są krytyczne dla produkcji w fabrykach i muszą być rozwiązane w możliwie najkrótszym czasie (maksymalnie 1 godzina). Przypadki wsparcia o średnim i niskim priorytecie nie są krytyczne i czas rozwiązania może być odpowiednio dłuższy – maksymalnie od jednego dnia do jednego tygodnia. Rozkład przypadków wsparcia dla wersji A jest pokazany na rys. 5.1.

Każda usterka systemu ROPTIS może potencjalnie doprowadzić do wstrzymania produkcji pojazdów, co może przełożyć się na znaczne straty finansowe dla fabryki dotkniętej awarią. Aby tego uniknąć, dla każdego podsystemu ROPTIS w umowie o poziomie usług (ang. *service level agreement*, w skrócie SLA) podpisanej pomiędzy zespołem programistów a przedstawicielem użytkownika końcowego (źródło: dokumentacja systemu ROPTIS) wymagano następujących poziomów dostępności:

- ROPTIS Operator – 99,8%,
- ROPTIS Server – 99,8%,
- ROPTIS Designer – 99,5%,
- ROPTIS Factory Manager – 99,7%.

Aby osiągnąć te wymagane poziomy dostępności, istniało duże zapotrzebowanie na zwiększenie jakości rozwiązania i zmniejszenie liczby defektów. Była to główna motywacja dla zespołu rozwijającego oprogramowanie, aby zastosować nową metodologię zapewniania

5.3. Metodyki zapewniania jakości w przedmiotowym projekcie rozwoju oprogramowania

The diagram illustrates the software testing process with the following components:

- Top Row (Test Phases):**
 - 4. NEGATYWNY WYNIK TESTU SYSTEMOWEGO (Negative System Test Result)
 - 6. RE-TEST
 - 7. TEST ZAKOŃCZONY NIEPOWODZENIEM (Test Ended in Failure)
- Bottom Row (Defect Management):**
 - DEFEKT DO NAPRAWY (Defect to be Fixed)
 - 5. DEFEKT DO NAPRAWY
 - DEFEKT DO NAPRAWY
 - DEFEKT DO NAPRAWY
 - DEFEKT DO NAPRAWY
 - 8. DEFEKT DO NAPRAWY
 - DEFEKT DO NAPRAWY
- Flow and Transitions:**
 - Arrows indicate a sequence from left to right through the testing phases.
 - Feedback loops (arrows pointing back) connect the defect management boxes to the re-test phase (6. RE-TEST).
 - A dashed arrow points from the 'Test Ended in Failure' phase to the defect management section.
- Additional Elements:**
 - A cloud labeled **REFAKTORING** (Refactoring) is connected to the process by a thought bubble.
 - A double-headed arrow labeled **DODATKOWY CZAS** (Additional Time) spans the defect management section.
 - A timeline at the bottom marks **1. FAZA TESTÓW SYSTEMOWYCH** (System Test Phase) and **2. REZERWA CZASOWA** (Time Reserve).
 - An arrow labeled **3. WYDANIE NA ŚRODOWISKO PRODUKCYJNE** (Release to Production Environment) points to the right.

5.3.1. Metodyka zapewniania jakości w czasie tworzenia wydania A

84

systemu ROPTIS do użytku przez użytkowników końcowych, co w nomenklaturze stosowanej w projekcie nosi nazwę *wydania na środowisko produkcyjne*.³

W metodyce STPBQA czas trwania fazy testowania systemu (1) został oszacowany na początku cyklu rozwojowego na podstawie liczby planowanych do wykonania przypadków testów systemowych. Ponadto, dodano rezerwę czasową (2) w celu zminimalizowania ryzyka przekroczenia wstępnie zaplanowanego czasu wydania oprogramowania do środowiska produkcyjnego (3), kiedy użytkownicy końcowi spodziewali się otrzymać nową wersję oprogramowania. Gdy wynik testu systemowego był negatywny (4), oznaczało to znalezienie defektu oprogramowania, który był przekazywany do naprawy przez programistów (5). Po naprawieniu stał się on przedmiotem ponownego testowania (6) aby potwierdzić, że został on rzeczywiście naprawiony. W tym celu poprzednio nieudany przypadek testu systemowego był wykonywany ponownie. Jeżeli przypadek testu systemowego ponownie się nie powiódł (7), defekt był ponownie przydzielany do naprawy (8). Jeżeli zespół programistów nie zdołał usunąć wszystkich wykrytych defektów podczas planowanej fazy testów systemowych, wykorzystywana była rezerwa czasowa (2). Gdy zespół programistów naprawił wszystkie wykryte wady, pozostały czas rezerwy (2) był wykorzystywany do refaktoryzacji (inaczej: refaktoringu) kodu źródłowego.⁴

Pliki źródłowe do refaktoryzacji były wybierane na podstawie analizy *ad hoc* i wspólnej decyzji zespołu programistów w zależności od ich wcześniejszych doświadczeń, co jest popularną praktyką [5, 4, 97].

5.3.2. Metodyka zapewniania jakości wykorzystująca predykcję defektów oprogramowania (DPDQA) wykorzystana podczas tworzenia wydania B

W związku z zapotrzebowaniem na rozwiązanie, które zwiększa jakość i zmniejsza liczbę defektów, autor niniejszej rozprawy zaproponował wykorzystanie nowej metodyki DPDQA jako podstawy nowego podejścia, przyjmując następujące założenia:

(1) *Użycie najlepszych praktyk pisania kodu zwiększa ogólną jakość tworzonego oprogramowania.* Badacze tacy jak R. Martin przekonują, że gdy programiści przestrzegają zasad architektonicznych tworzonego oprogramowania, stosując wzorce projektowe i praktyki SOLID, ogólna jakość oprogramowania wzrasta [128].

(2) *Predykcja defektów w oprogramowaniu dostarcza wczesnych informacji o lokalizacji kodu źródłowego podatnego na defekty.* Według badań przeprowadzonych przez badaczy, proces uczenia maszynowego może być wykorzystany efektywnie do wyszukiwania

³ Termin „wydanie na środowisko produkcyjne” pochodzi od angielskiego „release to production environment” i nie powinien być utożsamiany z terminem „wydanie” będącym – w tej pracy – synonimem wersji systemu.

⁴ Refaktoryzacja to proces, w wyniku którego kod źródłowy programu komputerowego jest przekształcany do postaci subiektywnie lepszej (np. czytelniejszej, bardziej wydajnej czy łatwiejszej w późniejszym utrzymaniu) niż przed refaktoryzacją.

przewidywania defektów do wsparcia testów integracyjnych lub akceptacyjnych. Jest to jedna z cech charakterystycznych metodyki DPDQA.

(5) *Każda faza rozważanego projektu ma ściśle ustalony czas trwania.* W przypadku przemysłowego rozwoju oprogramowania, czas i zasoby projektu są uważane za bardzo ważne ograniczenia. Czas trwania okresu rozwoju systemu ROPTIS zależy od rocznego cyklu produkcyjnego pojazdu i nie może być przedłużony. W tym konkretnym projekcie przemysłowym nie jest możliwe zastosowanie technik zapewniania jakości w całym kodzie programu z równą, wysoką skutecznością, ze względu na ograniczoną dostępność zasobów i ściśle planowanie czasowe. Nie było możliwości wprowadzenia nowej fazy „predykcja” czy też wydłużenie już istniejącej. Dlatego w naszym przypadku należało włączyć techniki predykcji defektów i zapewniania jakości do jednej z istniejących faz (w tym przypadku do fazy testów systemowych).

Zaproponowana przez autora niniejszej rozprawy metodyka DPDQA jest przedstawiona na rys. 5.3. Ważnym jej elementem jest predykcja defektów i jej rezultat (elementy pogrubione). Wynik predykcji działa jako element decyzyjny, który decyduje o tym, czy praktyki zapewniania jakości wobec kodu są stosowane. W linii czasowej projektu, DPDQA jest stosowana w fazie testowania systemu, kiedy pierwsza wersja nowo przygotowanego programu jest ukończona i stabilna. W ten sposób można wykorzystać metryki zebrane z tego kodu do przeprowadzenia procesu przewidywania defektów, na podstawie wcześniej stworzonego modelu predykcji (por. punkt 5.5).

Podczas fazy testowania systemu wykonanie przypadku testowego systemu (1) może zakończyć się niepowodzeniem testu systemowego (2), co najprawdopodobniej⁵ wskazuje na istnienie defektu, który powinien być naprawiony przez programistów (3). Przydzielony programista bada przyczynę defektu i określa, które z plików kodu źródłowego powinny zostać zmodyfikowane (naprawione), aby usunąć defekt. Następnie, wynik procesu predykcji defektów (4) – tj. lista podatnych na defekty plików kodu źródłowego – jest wykorzystywany do weryfikacji, czy pliki, które muszą być zmodyfikowane (naprawione) są na liście plików wskazanych jako podatnych na defekty (5). Jeżeli weryfikacja jest negatywna (tj. pliku wytypowanego do modyfikacji nie ma wśród wyników predykcji), tylko opis symptomów defektu (6) przygotowany przez testerów oprogramowania jest wykorzystywany przez programistów podczas procesu naprawczego (7). Po tym, jak defekt zostanie uznany za naprawiony przez programistę, staje się on przedmiotem testowania w celu sprawdzenia, czy

⁵ Niepowodzenie testu wskazuje na istnienie defektu oprogramowania tylko wtedy, gdy zostały spełnione określone warunki, takie jak ten, że test został zaprojektowany właściwie a środowisko testowe działa prawidłowo. W przypadku wystąpienia awarii środowiska testowego, np. zaniku połączenia z siecią czy bazą danych, wynik testu może być wynikiem fałszywie pozytywnym, tj. sygnalizować obecność defektu oprogramowania, gdy w istocie takiego defektu brak.

rzeczywiście został naprawiony (retest, 8). Gdy retest się nie powiedzie, cały proces dotyczący tego defektu powtarza się od punktu 3.

Weryfikacja (5) kończy się wynikiem pozytywnym, gdy przynajmniej jeden plik kodu źródłowego, spośród wszystkich wytypowanych do modyfikacji (naprawy) w celu usunięcia defektu (3) jest obecny w wynikach predykcji (czyli na liście plików wskazanych przez proces predykcji defektów jako podatnych na defekty). W takim przypadku modyfikacja (naprawa) wszystkich plików kodu źródłowego uwzględnia zastosowanie dodatkowych technik zapewniania jakości kodu, obejmujących na przykład zastosowanie wzorców projektowych, określonych pryncypiów (zasad) architektonicznych oraz stosowanie praktyk SOLID (patrz punkt 5.4). Dodatkowo, to pozytywny wynik przypadku testu systemowego (1) staje się głównym determinantem tego, czy defekt (3) został naprawiony. Zatem, aby zweryfikować naprawę defektu jeszcze przed wykonaniem retestu przez testera oprogramowania (8), przypadek testu systemowego jest dostarczany programiście w celu poinformowania go, jakiego zachowania programu oczekuje się po usunięciu defektu (test case-driven development; patrz punkt 5.4.6).

Wreszcie, wszystkie zmiany w kodzie powinny być przejrane przez innego programistę (10; patrz punkt 5.4.3). Jeżeli proces przeglądu kodu kończy się wynikiem pozytywnym, przypadek testu systemowego, który doprowadził do wykrycia defektu (1) jest ponownie wykonywany przez testera oprogramowania (retest, 8), aby potwierdzić, że defekt został naprawiony; w przeciwnym razie, jeżeli otrzymano negatywny wynik przeglądu, modyfikuje się kod stosując stosowne techniki zapewniania jakości, aż przegląd kodu będzie pozytywny.

5.4. Techniki zapewniania jakości w metodyce DPDQA

Ważnymi elementami metodyki DPDQA stosowanej podczas rozwoju wydania B systemu ROPTIS, oprócz procesu przewidywania defektów, były techniki zapewniania jakości kodu stosowane w przypadku, gdy plik kodu źródłowego był przewidywany jako podatny na defekty. Techniki te zostały opisane w poniższych punktach.

5.4.1. Wzorce projektowe

Złożoność systemu ROPTIS wymaga długotrwałego wprowadzenia dla każdego nowego programisty, który dołącza do zespołu. Po dziewięciu latach rozwoju ponad 30 różnych osób było zaangażowanych w tworzenie kodu, a każda z nich miała inne umiejętności w kodowaniu. Około dwóch lat to minimalny czas, który należy spędzić w projekcie, aby uzyskać wystarczająco dobre zrozumienie kodu i aby stać się mniej lub bardziej samodzielnym członkiem zespołu.

Jedną z przyczyn tak długiego okresu wprowadzenia był brak znormalizowanego sposobu pisania kodu. Było to również powodem wprowadzania do kodu wielu defektów podczas implementacji nowych, złożonych funkcjonalności (gorzej, że defekty te często ujawniały się jako usterki już po wdrożeniu, u użytkowników końcowych). Zasadniczo, każda nowa funkcjonalność powodowała dalsze zmiany w już istniejącym kodzie, co naruszało jedną z zasad SOLID, zasadę „otwarty-zamknięty” (patrz punkt 5.4.5) i wymuszało testowanie wcześniej przetestowanych funkcji. Aby przeciwdziałać opisanej sytuacji, podjęto decyzję o wprowadzeniu następujących wybranych wzorców projektowych, ze względu na wykazanie ich skuteczności w literaturze przedmiotu [221, 15]:

- Wzorce konstrukcyjne – singleton, fabryka abstrakcyjna;
- Wzorce strukturalne – dekorator, fasada, proxy (pełnomocnik);
- Wzorce czynnościowe – łańcuch odpowiedzialności, polecenie, specyfikacja, strategia.

Dzięki zastosowaniu wzorców projektowych kod stał się bardziej czytelny dla nowych programistów, szczególnie, że od każdego nowego programisty wymagano dobrej znajomości wzorców projektowych i zasad SOLID. Nawet jeśli nadal zachodziła konieczność nauki specyficznej wiedzy biznesowej, przy zastosowaniu metodyki DPDQA sam kod stawał się łatwiejszy do czytania, ponieważ każdy plik kodu źródłowego wybrany na podstawie wyniku predykcji defektów był refaktoryzowany.

5.4.2. Pryncypia architektoniczne

Według członków zespołu projektowego, ponad 30 programistów poświęciło ponad dziewięć lat i wiele wysiłku na wdrożenie złożonego systemu ROPTIS, jednak trudno było utrzymać solidną architekturę systemu. Badania przeprowadzone przez Ase Lindstrom oraz Garlanda i Perry’ego udowodniły wysoką skuteczność stosowania wspólnych zasad architektonicznych w pisaniu kodu dla całych organizacji na rzecz zapewniania jakości i spójności projektu [57, 176]. Idąc ich przykładem, została ustalona ogólna zasada, która wymagała, by każda modyfikacja pliku wskazanego jako „defektogenny” przez proces predykcji defektów (jak opisano to w punkcie 5.3.2) była oceniona przez wyznaczonego architekta oprogramowania pod kątem ustalonego zestawu zasad architektonicznych⁶.

Następujący zestaw jest częścią zbioru zasad architektury systemów informatycznych firmy-partnera [175]:

- zgodność ze specyficznymi normami przemysłowymi,
- autonomiczne i luźne sprzężenie pomiędzy komponentami i systemami,
- prostota rozwiązań i metod pracy,
- dążenie do wykorzystania istniejących usług przedsiębiorstwa,
- solidne rozwiązanie (asynchroniczność, buforowanie itd.),

⁶ W przypadku implementacji zupełnie nowych funkcjonalności także stosowano podobne podejście.

- wydajność od samego początku,
- bezpieczne rozwiązania,
- dobre rozwiązania integracyjne,
- wykorzystanie metod pracy Agile i zwinnych zasad projektowania.

Skupienie się na wyżej wymienionych zasadach doprowadziło do uproszczenia kodu źródłowego i dało lepsze panowanie nad kompletnym projektem. W konsekwencji stosowania metodologii DPDQA, pewne części starego kodu – pliki oznaczone jako podatne na defekty w procesie predykcji – zostały napisane na nowo zgodnie z powyższymi zasadami.

5.4.3. Przeglądy kodu

W początkowej fazie niniejszych badań wprowadzenie wzorców projektowych i zasad architektonicznych okazały się niewystarczająco efektywne do takiego poziomu jakości kodu, aby zapewnić wymaganą umową SLA poziomu dostępności modułów aplikacji (patrz punkt 5.2.2). Działając pod presją czasu, członkowie zespołu mieli tendencję do pomijania dłuższego procesu naprawy defektów i tym samym implementacji wzorców projektowych i stosowania zasad SOLID na rzecz „prostego procesu naprawy defektu” (7 na rysunku 5.3). Postanowiono poszukać rozwiązania tego problemu. Zachęcające wyniki badaczy [191, 44, 51, 125], pokazujące, że:

- Przeglądy kodu przynoszą dzielenie się wiedzą;
- Świeże spojrzenie na kod często prowadzi do prostej, ale wartościowej zmiany;
- Przeglądy ograniczają konflikty przy łączeniu i powielaniu funkcjonalności, np. poprzez wczesne ostrzeżenie, że ktoś z innego zespołu pracuje nad tym samym fragmentem kodu;
- Unika się wprowadzania niektórych defektów,

były czynnikiem decydującym, aby włączyć przeglądy kodu jako dodatkowy element metodyki DPDQA. Wcześniej technika stosowania przeglądów kodu była niejednokrotnie stosowana podczas przygotowania poprzednich, historycznych wydań systemu ROPTIS, ale nie przyniosła pożądanych efektów. Skupiała się ona głównie na komentarzach, konwencji nazewnictwa oraz strukturze klas i metod języka C#. W tamtym czasie świadomość wzorców projektowych i innych praktyk zapewniania jakości kodu była w zespole niska.

Kolejna próba wprowadzenia wartościowych przeglądów kodu została podjęta, kiedy zastosowano proponowaną metodologię DPDQA i skoncentrowano się na zasadach SOLID, wzorcach projektowych i eliminacji kodu napisanego wbrew sztuce tworzenia oprogramowania. Wprowadzono specjalną zasadę, aby kod zmodyfikowany w ramach stosowania metodyki DPDQA, gdy weryfikacja wyników predykcji zakończyła się pozytywnie (5 na rysunku 5.3), nie mógł być wysłany do repozytorium kodu bez wcześniejszego sprawdzenia przez innego

członka zespołu programistów. Nazwisko osoby przeglądającej powinno być również zapisane w repozytorium, razem z przesłanym kodem i nazwiskiem programisty-autora.

W trakcie niniejszych badań zaobserwowano, że zmiana ta została doceniona przez młodszych programistów. Mieli oni możliwość uczenia się na błędach w sposób „miękki” – zanim zły jakości kod mógł wpłynąć na rzeczywisty system i ewentualnie spowodować awarię na środowisku „produkcyjnym”, czyli w fabrykach korzystających na co dzień z systemu ROPTIS.

Kolejną obserwacją było to, że nawet, jeśli zmiany były zatwierdzone przez architekta oprogramowania, rzadko się zdarzało, by osoba wyznaczona do przejrzania kodu nie miała żadnych uwag. W związku z tym podjęto decyzję, aby częściej przeprowadzać przeglądy, nawet jeśli naprawa defektu nie była jeszcze ukończona, w celu zapobieżenia poważnym zmianom w kodzie źródłowym, gdy wątpliwy kod został znaleziony podczas „finalnego” przeglądu po zakończeniu naprawy, bądź – co gorsza – już po wysłaniu kodu do repozytorium.

Wprowadzona praktyka przeglądu kodu wywołała jednak problem z zarządzaniem zespołem. Niektórzy programiści zgłaszali, że czują się upokorzeni uczestnicząc w przeglądach. W takich przypadkach konstruktywne informacje zwrotne były zazwyczaj mylone z krytyką. To prowadziło do unikania przeglądów kodu, a nawet do łamania wspomnianej zasady, że nie należy wprowadzać kodu do repozytorium bez nazwiska osoby przeglądającej. Ostatecznie, niechęć ta została w końcu przezwyciężona na rzecz zrozumienia pozytywnego wpływu przeglądów na jakość dostarczanego oprogramowania.

Obecnie przeglądy kodu są doceniane i realizowane przy wsparciu narzędzi, takich jak Microsoft Azure DevOps Server [139, 26].

5.4.4. Refaktoryzacja

Przed wprowadzeniem metodologii DPDQA pierwszą rozpoznaną potrzebą było wprowadzenie refaktoryzacji kodu do projektu ROPTIS, co było koniecznością, aby uzyskać zauważalne skoki wydajności w module „ROPTIS Viewer”. Sam moduł nie był początkowo przeznaczony do użytku przez użytkowników końcowych w bardzo wymagającym środowisku produkcyjnym. Był raczej prowizorycznym rozwiązaniem przeznaczonym dla członków zespołu, do sprawdzania logów przechowywanych przez serwer. Narzędzie okazało się na tyle przydatne, że decyzją kierownictwa zostało przygotowane do udostępnienia go fabrykom. Ponieważ nie było ono ani ukierunkowane na wydajność ani szczególnie dobrze zaprojektowane, wywołało obawy dotyczące przyszłego poziomu jakości.

Bez gruntownej refaktoryzacji kodu modułu nie można było wyeliminować możliwych defektów typu *timeout* (czyli: gdy komunikacja na linii klient-serwer przekracza maksymalny, dostępny czas na transfer danych). Podjęto decyzję o wprowadzeniu dużej zmiany i po analizie możliwych rozwiązań, zdecydowano się na refaktoryzację całego kodu źródłowego modułu

„ROPTIS Viewer”, ponieważ badania wykazały wysoką skuteczność tej techniki zapewniania jakości kodu [133, 49].

W wyniku refaktoryzacji modułu „ROPTIS Viewer” czas wyszukiwania danych do wyświetlania został zredukowany z minut do kilku sekund. To duże ulepszenie zmieniło również nastawienie kadry kierowniczej projektu – do tej pory uważano za niepotrzebne ryzyko, aby poświęcać czas i wykorzystywać zasoby na działania, które nie przynoszą nowych funkcji⁷.

Na podstawie tego studium przypadku zdecydowano się na wprowadzenie procesu refaktoryzacji do metodyki DPDQA i zastosowanie tzw. „reguły skauta”, która brzmi następująco: „zawsze zostawiaj kod lepszy, niż go znalazłeś” [129]. Pozwoliło to na refaktoryzację małych części kodu podczas każdego usuwania defektu.

Co ważne, aby zapobiec wprowadzaniu przekłamań („szumu”) do danych wejściowych dla predykcji defektów, zmiany kodu następujące w wyniku stosowania się do „reguły skauta” wysyłane były w osobnym „commicie”, zgodnie z zasadami omówionymi w rozdziale 3.

Każda naprawa defektu, która wymagała dłuższego czasu na przepisanie kodu przy użyciu wzorców projektowych, zasad architektonicznych i zasad SOLID, była traktowana w projekcie jako oddzielne zadanie, aby zapewnić odpowiednią ilość czasu na ukończenie tego zadania i późniejszy przegląd kodu.

5.4.5. Praktyki SOLID

Wskutek zaobserwowanego wzrostu produkcji w firmie współpracującej przy badaniach, nowych produktów i modernizacji ich architektury elektronicznej, oczekiwano, że system ROPTIS zostanie w najbliższych latach znacznie rozbudowany. To mogłoby potencjalnie powodować problemy w utrzymaniu systemu, długie czasy wprowadzania zmian oraz powodować dużą liczbę różnych zapachów kodu⁸. W swoich badaniach Martin argumentował, że można uniknąć powyższego poprzez zastosowanie w projektowaniu programów obiektowych pięciu zasad, nazwanych przez niego SOLID [128]. Wspólnie z zespołem programistów zdecydowano, że praktyki zapewniania jakości kodu powinny również zapewnić poprawę łatwości utrzymania kodu poprzez przestrzeganie zasad zaproponowanych przez Martina. Ich nazwa wzięła się od pierwszych liter ich angielskich nazw:

- **Single responsibility principle** – zasada jednej odpowiedzialności mówiąca, że nigdy nie powinno być więcej niż jednego powodu do istnienia metody lub klasy [128];
- **Open/closed principle** – zasada otwarte/zamknięte mówiąca, że elementy kodu źródłowego (klasy, metody, funkcje itp.) powinny być zamknięte na modyfikacje, a otwarte

⁷ Ta zmiana nastawienia kierownictwa umożliwiła też inne prace nie przekładające się bezpośrednio na nowe funkcjonalności systemu – mowa tu o wprowadzeniu metodyki DPDQA.

⁸ Więcej o zapachach kodu i zastosowaniu predykcji defektów w tym kontekście Czytelnik dowie się z rozdziału 6.

na rozszerzenia, zatem zmiana zachowania tego elementu powinna być możliwa bez modyfikacji jego kodu [128];

- **Liskov substitution principle** – zasada podstawienia, sformułowana po raz pierwszy w 1987 przez Barbarę Liskov [115], mówiąca, że klasa dziedzicząca powinna rozszerzać zachowanie klasy bazowej ale nie powinna zmieniać jej pierwotnych możliwości;
- **Interface segregation principle** – zasada segregacji interfejsów mówiąca, że zaprojektowanie kilku dedykowanych interfejsów (abstrakcyjnych typów danych dysponujących jedynie operacjami) jest lepsze, niż jednego ogólnego [128];
- **Dependency inversion principle** – zasada odwrócenia zależności mówiąca, że wysokopoziomowe (ogólne) elementy programu nie powinny zależeć od elementów niskopoziomowych (specyficznych) – zależności te powinny wynikać ze stosowania abstrakcji [128].

5.4.6. Test Case-Driven Development

W ramach metodologii DPDQA znalazło się także wykorzystanie rozwoju sterowanego testami (TDD, od ang. *Test Driven Development*), który jest metodą rozwoju oprogramowania wynikającą z potrzeby wdrożenia możliwych sposobów ograniczenia liczby głównych i krytycznych defektów, *zanim* oprogramowanie zostanie przetestowane przez zespół testowy [13, 92, 140, 118].

W poprzednio stosowanej metodyce STPBQA po tym, jak defekt został naprawiony i przetestowany przez programistę, informacja o naprawie defektu była przekazywana do wyznaczonego testera oprogramowania w celu przeprowadzenia testu potwierdzającego (retestu). Z obserwacji wynikało, że często czynności testowe były przerywane przez poważne defekty, jak np. brakujące pliki w pakietach instalacyjnych, niedziałające funkcjonalności (na które wpłynęła naprawa defektu, ale które nie zostały bezpośrednio zmienione podczas tejże naprawy) lub inne zachowania oprogramowania niezgodne ze specyfikacją.

Aby ograniczyć podobne problemy, zaproponowano podejście, nazwane roboczo "Test Case-driven Development": dla każdego defektu znalezionej w sklasyfikowanych jako defektogenne plikach kodu źródłowego wyznaczany był zestaw przypadków testowych. Zestaw ten był następnie dostarczany do odpowiedniego programisty zajmującego się naprawą wspomnianego defektu (najlepiej przed rozpoczęciem naprawy, jeśli przypadki testowe były już gotowe).

Rozwiązanie to dawało możliwość sprawdzenia poprawki i zastosowania wymaganych zmian w dokumencie podczas naprawy. Jako ostateczny kontrakt z zespołem testującym oprogramowanie programista dostarczał raport z wykonania przypadków testowych – listę wykonanych przypadków testowych z odpowiednim rezultatem ich wykonania („pozytywny” lub „negatywny”).

To podejście różni się od najczęściej stosowanej praktyki programowania sterowanego testami tym, że zamiast twórcy oprogramowania piszącego testy, przypadki testowe przygotowywane są przez testera oprogramowania. W ten sposób eliminuje się problem wskazany m.in. przez Maricka, który argumentował, że najbardziej efektywne testowanie oprogramowania można osiągnąć, gdy testy są przygotowywane przez inną osobę (testera oprogramowania) niż programista [127]. Wykorzystanie przygotowanych przypadków testowych daje informacje o możliwych niepożądanych zachowaniach programu, które mogłyby zostać przeoczone przez programistę.

5.5. Model predykcji zbudowany przy pomocy narzędzia DePress

W proponowanej metodologii DPDQA decyzja, czy dana technika zapewniania jakości (patrz punkt 5.3.1) powinna być zastosowana czy też nie na danym pliku kodu źródłowego, opiera się na liście plików sklasyfikowanych jako podatnych na defekty. Taka lista jest wynikiem procesu klasyfikacji, w którym stosowany jest model predykcji. W dalszej części niniejszego rozdziału zostanie przedstawione wybrane na tym etapie podejście do budowania serii modeli predykcji, po to, aby wybrać najbardziej efektywny. Ponieważ jednym z celów badań było zweryfikowanie oprogramowania DePress w komercyjnych projektach rozwoju oprogramowania, zostanie on użyty jako główne narzędzie do tego zadania.

5.5.1. Wykorzystane dane

Dla projektu ROPTIS dane dostępne do celów budowy modelu predykcji to metryki kodu, informacje o odkrytych defektach oraz dane opisujące zmiany w kodzie spowodowane przez te defekty. Aby zbudować model predykcji, który byłby w stanie wesprzeć tworzenie wydania B aplikacji ROPTIS, wykorzystano dane zawierające metryki kodu źródłowego wydania A, zebrane zanim jakakolwiek akcja naprawcza została na nim wykonana.

Narzędzie Microsoft Visual Studio, które zostało wykorzystane jako środowisko programistyczne dla tego projektu, dzięki wykorzystaniu wbudowanej wewnątrz funkcji zwanej *SourceMonitor*, dostarczyło następujące metryki kodu (opisane szerzej w punkcie 2.3.1):

- indeks utrzymywalności kodu,
- złożoność cyklomatyczna,
- głębokość drzewa dziedziczenia,
- sprzężenie klas C#,
- całkowita liczba linii kodu,
- liczba wykonywalnych linii kodu.

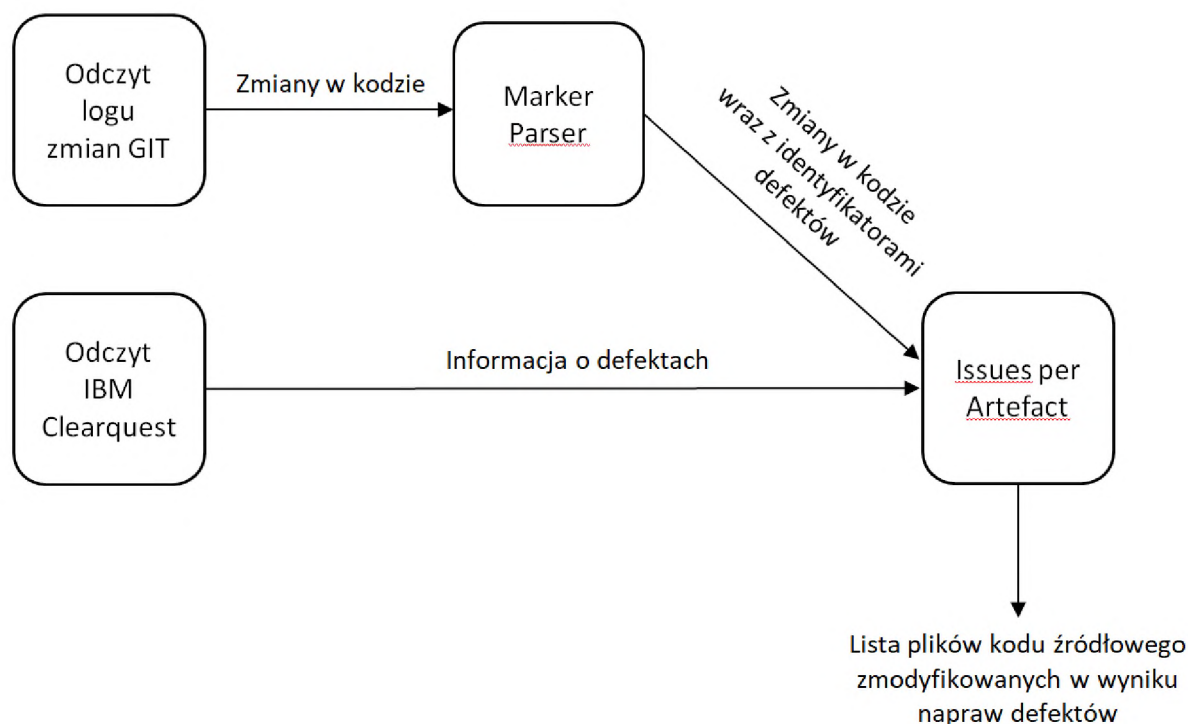
Proces predykcji defektów za pomocą narzędzia DePress, stosowany na tym etapie badań, pozostawał spójny z dotychczasowo przyjętym (opisanym w rozdziałach 3 oraz 4), który okazał się zarówno wykonalny w środowisku komercyjnego wytwarzania oprogramowania, jak i efektywny kosztowo. Implementując to podejście na potrzeby wersji B projektu ROPTIS, pierwszym krokiem potrzebnym do skonstruowania zbioru danych treningowych było rozpoznanie, które z części plików z kodem źródłowym wersji A zostały zmodyfikowane w wyniku usunięcia defektów o różnym stopniu ciężkości. Jak zostało to opisane w punkcie 4.5.2 niniejszej pracy, dokonanie takiej obserwacji stało się możliwe dzięki węzłowi narzędzia DePress o nazwie *Issues per Artefact*, która wykorzystuje dane wyeksportowane z repozytorium kodu Team Foundation Server (w formacie logu „commitów” GIT) oraz z narzędzia śledzenia defektów IBM ClearQuest, które w tym przypadku eksportuje dane w postaci pliku xml. Poprawki defektów zarejestrowane w repozytorium kodu zostały rozpoznane na podstawie danych z narzędzia śledzenia defektów. Stało się to możliwe w wyniku zaistnienia w projekcie ROPTIS okoliczności opisanych w rozdziale 3 jako konieczne do przeprowadzenia predykcji defektów: dzięki unikalnemu identyfikatorowi defektu, który jest przypisany do każdego wykrytego defektu i jest przechowywany jako komentarz w repozytorium „commitów” zawierających kod naprawiający usterki. Także i w tym przypadku „commity” zawierające unikalne identyfikatory defektów w polach komentarzy zostały wskazane przez węzeł DePress o nazwie *Marker Parser*. Diagram przepływu danych implementujący powyższy proces jest przedstawiony na rys. 5.4.

W wyniku tego procesu otrzymano dane opisujące rozkład defektów wśród plików kodu źródłowego. Dane te zostały wykorzystane do odróżnienia metryk kodu opisujących pliki kodu źródłowego podatne na defekty od metryk opisujących pliki kodu nie podatnego na defekty. Szczegóły tego procesu są opisane w punkcie 5.5.2. Ostatnim etapem tworzenia zbioru danych treningowych było połączenie informacji o rozmieszczeniu defektów (między plikami kodu źródłowego) z metrykami zebranymi dla każdego pliku.

Podobny zbiór danych, potrzebny do przygotowania predykcji defektów dla wydania B systemu ROPTIS, został zebrany z metryk kodu wydania B.

5.5.2. Klasyfikacja

Dążąc do wdrożenia proponowanej metodyki DPDQA zdecydowano, że proste, kategoryczne rozróżnienie między plikami kodu źródłowego podatnymi na defekty i plikami kodu źródłowego, które nie są podatne na defekty, będzie wystarczające do tego celu. Po przeanalizowaniu liczby, dotkliwości i rozmieszczenia wykrytych defektów w wydaniu A podjęto decyzję o oznaczeniu plików kodu źródłowego, w których działania związane z usuwaniem usterek były spowodowane przez co najmniej dwa defekty o stopniu dotkliwości „major” („duży defekt”) lub wyższym, jako podatne na defekty. Inne pliki, w tym również



Rysunek 5.4: Zmodyfikowany diagram przepływu z rys. 3.2, pozwalający uzyskać informacje o rozkładzie zmian w kodzie źródłowym projektu ROPTIS, powstałych w wyniku napraw defektów.

te pliki, w których zostały usunięte *pojedyncze* defekty o dotkliwości „*major*” lub większej, były oznaczone jako nie podatne na defekty. Pliki podatne na defekty zostały oznaczone w przygotowanych wcześniej danych treningowych w dodatkowej kolumnie, za pomocą binarnej jedynki – „1” a pliki nie podatne na defekty były oznaczane binarnym zerem – „0”. W wyniku tego rodzaju kategoryzacji otrzymano surowy zbiór danych treningowych, który charakteryzował się silnym niezbalansowaniem klas, z dominującą klasą „0” (pliki nie podatne na defekty) w stosunku 62:1.

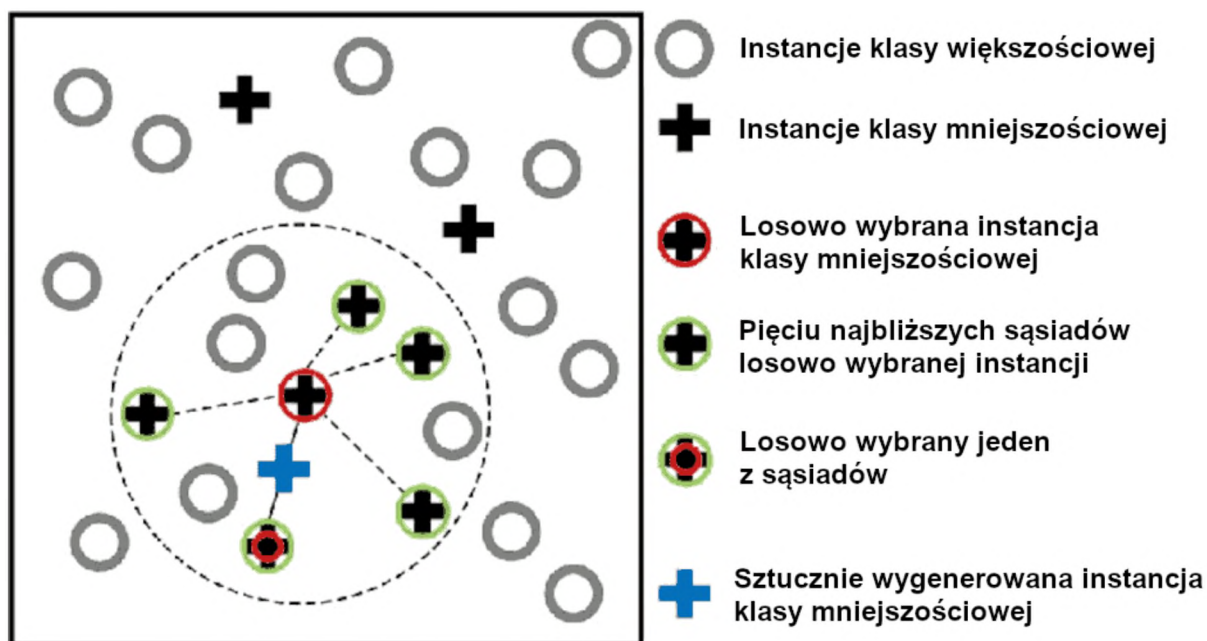
5.5.3. Działania przeciwdziałające niezbalansowaniu klas

W celu przeciwdziałania niezbalansowaniu klas, w toku badań zdecydowano się na przetestowanie dwóch różnych rozwiązań i sprawdzenie, które da lepsze rezultaty.

Pierwsze podejście polegało na zmniejszeniu liczby instancji klasy większościowej poprzez losowe jej próbkowanie, podobnie, jak zostało to zrealizowane wcześniej i przedstawione w punkcie 4.3.2. Po podzieleniu zbioru danych treningowych pomiędzy klasy („1” i „0”) zastosowano węzeł próbkowania tak, jak to przedstawiono na rys. 4.1. Użyto metody prób i błędów, aby określić wielkość próby i znaleźć stosunek klas, który daje najlepszy model predykcji, bez uwzględniania żadnych innych czynników.

Jako alternatywne podejście wybrano użycie algorytmu, który został wprowadzony do narzędzia KNIME/DePress na tym etapie badań, zwanego Synthetic Minority Oversampling

Technique (SMOTE), by sztucznie zwiększyć liczbę instancji klasy mniejszościowej tak, aby osiągnąć stosunek klas 1:1. W tym przypadku wykorzystano sprawdzoną formę algorytmu, którą efektywnie stosowali Chawla et al. [27] – polegającą na losowym wyborze pięciu najbliższych sąsiadów (otaczających aktualnie rozpatrywaną instancję) jako podstawy do zbudowania syntetycznej instancji pośredniej, jak ilustruje to rysunek 5.5.



Rysunek 5.5: Ilustracja działania algorytmu SMOTE [177].

5.5.4. Procedura wyboru cech

W procesie przygotowania predykcji defektów do celów metodyki DPDQD wykorzystano także mechanizm wyboru cech narzędzia KNIME/DePress opisany w punkcie 4.3.3.

5.6. Wyniki i podsumowanie tego etapu badań

Z praktycznego punktu widzenia, sensownym jest zrobienie listy plików źródłowych podatnych na defekty, bazujących na możliwie najbardziej efektywnym modelu predykcji. W tym podrozdziale opisane zostaną rezultaty przygotowania i walidacji kilku wybranych modeli predykcji. Następnie przedstawione zostaną koszty implementacji w komercyjnym projekcie wytwarzania oprogramowania – dysponując wynikami predykcji, metodyka DPDQA została wykorzystana w procesie tworzenia wersji B systemu ROPTIS; obserwacje zebrane podczas pracy z metodyką zebrane zostały w punkcie 5.5. Następnie pokazany zostanie wpływ proponowanej metody pracy na jakość i wyniki wydania B systemu. W punkcie 5.6.3 opisane

zostaną zagrożenia dla rzetelności badań (ang. *threats to validity*) przeprowadzonych na tym etapie.

5.6.1. Predykcja plików podatnych na defekty

Aby przeprowadzić predykcję defektów dla plików z kodem źródłowym, która zostanie wykorzystana jako kluczowy element metodyki DPDQA, zbudowany i przetestowany został szereg modeli predykcji z wykorzystaniem uczącego zbioru danych (opisanego w punkcie 5.5.1), podzielonego na dwie partycje z wykorzystaniem próbkowania warstwowego (*stratified sampling*): jedna partycja jest wykorzystana do zbudowania modelu, zaś druga – do jego walidacji. Wybrano następujące klasyfikatory, które są dostępne w narzędziu KNIME – trzy użyte już wcześniej: naiwny klasyfikator Bayesa, probabilistyczną sieć neuronową i drzewo decyzyjne oraz jeden „nowy” klasyfikator, wprowadzony do narzędzia tuż przed rozpoczęciem tego etapu badań – las losowy, aby poddać je walidacji celem wybrania najbardziej efektywnego z nich. Ponieważ kluczową rolą tworzenia modelu było zapewnianie jak najlepszego przewidywania najbardziej podatnych na defekty części kodu (które, zgodnie z proponowaną metodyką, wymagają dalszego ulepszenia), z miar stosowanych do oceny klasyfikatorów (opisanych w punkcie 2.2.1) wybrano następujące jako decydujące: *czułość* (udział plików rzeczywiście defektogennych w ogólnej liczbie tych, które zostały sklasyfikowane jako defektogenne [216]) dla plików oznaczonych jako podatne na defekty (co jest istotne dla czasochłonności zadań, które będą podejmowane jako konsekwencja interpretacji wyników predykcji) oraz *współczynnik korelacji Matthews* pomiędzy zaobserwowaną a przewidzianą klasyfikacją. Dodatkowo, dla pełniejszego zobrazowania wyników, zebrano także wartości *precyzji*, *swoistości* oraz *miary F* obliczone przez narzędzie KNIME.

W przypadku, gdy do przeciwdziałania niebalansowaniu klas zastosowano metodę próbkowania klasy większościowej, dla każdego klasyfikatora zastosowano pięć różnych stosunków klas; wyniki są przedstawione w tabeli 5.2. Podobnie, wyniki drugiej metody przeciwdziałania niebalansowaniu klas (zastosowanie algorytmu SMOTE) są przedstawione w tabeli 5.3. Wyższe wartości *czułości* i *współczynnika korelacji Matthews* zaobserwowano w przypadku użycia algorytmu SMOTE, niezależnie od zastosowanego klasyfikatora. Dla obu użytych metod przeciwdziałających niebalansowaniu klas, drzewo decyzyjne i las losowy uzyskały najlepsze wyniki.

Najwyższe wartości *współczynnika korelacji Matthews* (0.928) i *czułości* (0.979) zostały uzyskane przy użyciu kombinacji SMOTE i lasu losowego oraz podzbioru pięciu metryk (indeks utrzymywalności, złożoność cyklopatyczna, głębokość drzewa dziedziczenia, sprzężenie klas i liczba linii kodu) z łącznie sześciu dostępnych cech (metryk kodu) zbioru danych.

Zatem powyższa kombinacja algorytmów i powyższy podzbiór cech zestawu danych zostały wykorzystane do predykcji podatnych na defekty plików kodu źródłowego przy użyciu metryk kodu zebranych z wydania B. Wynik predykcji, kluczowy dla proponowanej metodologii DPDQA, był sporządzany w formie listy plików kodu źródłowego, odpowiednio sklasyfikowanych przez model predykcyjny.

Tabela 5.2: Wyniki walidacji modelu predykcji obszarów w kodzie podatnych na defekty – wykorzystanie próbkowania klasy większościowej.

Udział klasy większościowej	Liczba wybranych metryk	Precyzja	Czułość	Swoistość	Miara F	Współczynnik korelacji Matthews’a
Naiwny klasyfikator Bayesa						
1:1	4	0.762	0.821	0.487	0.790	0.322
1:1.02	4	0.872	0.526	0.846	0.656	0.358
1:1.25	2	0.831	0.821	0.667	0.826	0.484
1:1.5	4	0.797	0.654	0.667	0.718	0.304
1:2	2	0.744	0.821	0.436	0.780	0.274
Probabilistyczna sieć neuronowa						
1:1	2	0.782	0.872	0.513	0.824	0.415
1:1.02	4	0.819	0.756	0.667	0.787	0.410
1:1.25	4	0.850	0.872	0.692	0.861	0.572
1:1.5	2	0.770	0.859	0.487	0.812	0.374
1:2	3	0.728	0.962	0.282	0.829	0.354
Drzewo decyzyjne						
1:1	4	0.902	0.949	0.800	0.925	0.770
1:1.02	3	0.854	0.897	0.700	0.875	0.614
1:1.25	3	0.750	0.923	0.400	0.828	0.393
1:1.5	2	0.791	0.872	0.550	0.829	0.449
1:2	5	0.892	0.846	0.800	0.868	0.633
Las losowy						
1:1	4	0.837	0.923	0.650	0.878	0.610
1:1.02	4	0.864	0.974	0.700	0.916	0.733
1:1.25	3	0.814	0.897	0.600	0.854	0.530
1:1.5	3	0.822	0.949	0.600	0.881	0.611
1:2	2	0.729	0.897	0.350	0.805	0.301

5.6.2. Koszta implementacji

Koszta implementacji metodyki i narzędzi mogą być potencjalnie dużym ograniczeniem, a nawet przeszkodą blokującą możliwość wprowadzenia proponowanej zmiany. Koszta wdrożenia procesu predykcji bazującego na DePress zostały zbadane i przedstawione w niniejszej pracy w rozdziale 4 i są one podobne w tym przypadku, głównie za sprawą darmowego dostępu oprogramowania DePress z licencją GNU, a także dzięki jego interfejsowi, który jest łatwy w nauce i użyciu.

Tabela 5.3: Wyniki walidacji modelu predykcji obszarów w kodzie podatnych na defekty – wykorzystanie SMOTE.

Udział klasy większościowej	Liczba wybranych metryk	Precyzja	Czułość	Swoistość	Miara F	Współczynnik korelacji Matthews'a
Naiwny klasyfikator Bayesa						
1:1	5	0.761	0.650	0.795	0.701	0.450
Probabilistyczna sieć neuronowa						
1:1	1	0.897	0.867	0.901	0.882	0.768
Drzewo decyzyjne						
1:1	6	0.933	0.957	0.931	0.945	0.889
Las losowy						
1:1	5	0.950	0.979	0.949	0.964	0.928

Tabela 5.4: Koszty wprowadzenia predykcji defektów w projekcie ROPTIS zarejestrowane przy użyciu oprogramowania SAP CATS.

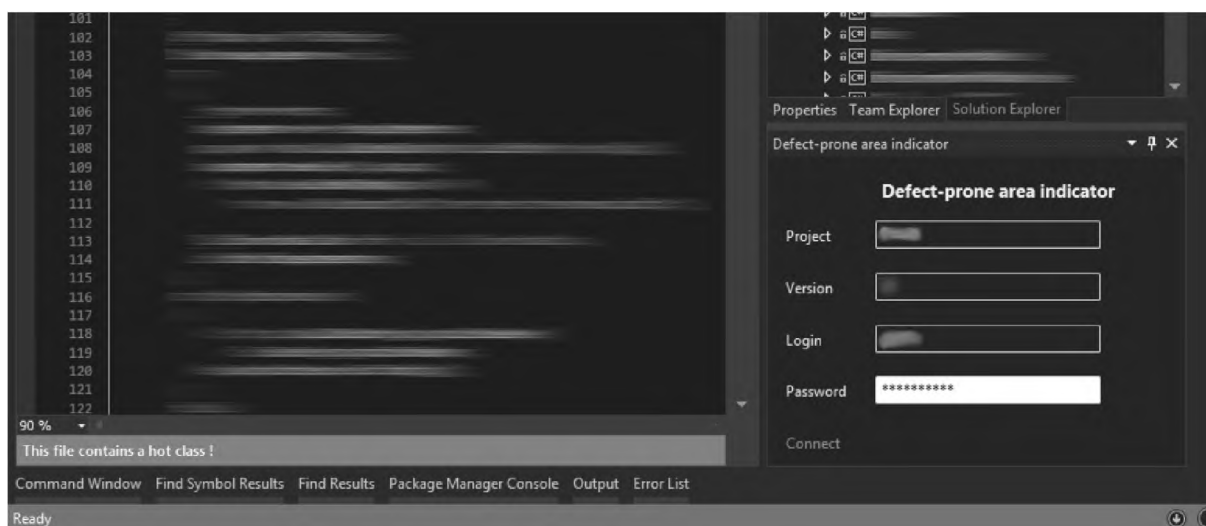
Wyszczególnienie	Koszt [roboczogodziny]
Nabycie narzędzia DePress i koszty instalacji	0
Szkolenie (nauka)	4
Zebranie danych	8
Przygotowanie predykcji defektów	1
Łączne koszty	13

Łączne koszty implementacji predykcji pokazane w tabeli 4.7 zostały uzyskane z oprogramowania śledzącego czas pracy wyprodukowanego przez firmę SAP, zwanego Cross-Application Time Sheet (CATS) i wykorzystywanego przez zespół projektowy. Jak widać, założenia teoretyczne sporządzone w toku badań opisanych w rozdziale 4 praktycznie pokrywają się z rzeczywistym czasem, jaki poświęcono na wdrożenie i przeprowadzenie predykcji defektów.

Jak pisano wcześniej w punkcie 5.2.1, ograniczenia czasowe i zasobowe projektu nie mogą ulec zmianie. Z tego powodu koszty pracy dla wydania A i B są porównywalne. Wykorzystanie metodyki DPDQA zmieniło jedynie organizację pracy, ale nie wpłynęło znacząco na jej nakład.

5.6.3. Spostrzeżenia z pracy z metodyką DPDQA

Przed wprowadzeniem metodyki DPDQA przygotowano zestaw wewnętrznych warsztatów, które zostały poprowadzone przez najbardziej doświadczonych, zorientowanych na zapewnianie jakości oprogramowania, członków zespołu. Młodszy programiści wiedzieli o



Rysunek 5.6: W dolnej części okna środowiska programowania widoczny komunikat ostrzegawczy o pliku podatnym na uszkodzenia („This file contains a hot class!”), dostarczany przez wtyczkę korzystającą z wyników predykcji defektów. Po prawej stronie widoczne okno dialogowe ustawień wtyczki.

istnieniu SOLID, ale po tym, jak wpływ reguł na jakość została im pokazana w praktyce, zapadł do wdrożenia zasad SOLID był duży. Po wprowadzeniu metodyki każdy przegląd kodu koncentrował się na powyższych zasadach i dlatego każde naruszenie było natychmiast wskazywane. Sukces warsztatów doprowadził do kolejnej decyzji, aby przeprowadzać więcej warsztatów jako inwestycję w jakość kodu. Bardzo interesującym efektem ubocznym było to, że metodyka ta stała się popularna wśród pracowników firmy spoza projektu, którzy dostrzegli znaczenie predykcji defektogennych obszarów w kodzie w połączeniu ze stosowaniem dobrych praktyk programowania.

Aby przyspieszyć proces przekazywania informacji mówiącej, czy dany plik kodu źródłowego został sklasyfikowany jako podatny na defekty, opracowano specjalną wtyczkę do zintegrowanego środowiska programistycznego (Microsoft Visual Studio). Kiedy programista rozpoczynał pracę z plikiem źródłowym, unikalny ciąg – składający się ze ścieżki do pliku i jego nazwy – był porównywany z wynikiem predykcji: jeśli plik był sklasyfikowany jako podatny na defekty, wtyczka to sygnalizowała (jak na rys. 5.6⁹). W myśl metodyki DPDQA, programista był w takim wypadku zobowiązany zastosować uzgodnione najlepsze praktyki kodowania (jak opisano w punkcie 5.3.2) w procesie naprawy defektów. Dodatkowo, w celu narzucenia przeglądu kodu, w Microsoft Team Foundation Server używanym do kontroli wersji, została ustanowiona specjalna reguła, która zabraniała wysyłania do repozytorium kodu pliku rozpoznanego przez wtyczkę jako defektogenny, jeśli „commit” nie zawierał nazwiska osoby,

⁹ Autorowi nie wolno ujawniać szczegółów kodu źródłowego, dlatego kod na rysunku zrzutu ekranu został zamazany.

która przejrzała kod pod kątem ustalonych praktyk. Te mechanizmy pomogły w przestrzeganiu metodyki DPDQA.

Alert generowany przez wspomnianą wtyczkę był również wykorzystywany do podejmowania decyzji o użyciu testów jednostkowych w sygnalizowanym kodzie. W zależności od przewidywanej pracochłonności refaktoryzacji pliku, członkowie zespołu mogli zdecydować, czy zaimplementować też testy jednostkowe, czy też położyć nacisk na testowanie funkcjonalności realizowanych za pomocą kodu zawartego w pliku na poziomie systemowym. Bez względu na wynik powyższej decyzji, świadomość, że kod był sklasyfikowany jako podatny na defekty, zazwyczaj powodowała bardziej wnikliwe testy (jednostkowe bądź systemowe).

Przykładem korzyści z wdrożenia programowania sterowanego przypadkami testowymi (opisanego w punkcie 5.4.6), jako części metodyki DPDQA, było rozwiązanie problemu powtarzających się nieudanych testów weryfikujących instalację jednego z modułów ROPTIS. Gdy programistom dostarczono krytyczny scenariusz wywołujący problemy z instalacją, zdecydowali oni o przeprowadzeniu dodatkowych testów instalacji modułu, na specjalnie stworzonych maszynach wirtualnych. Maszyny te odzwierciedlały specjalną wersję środowiska produkcyjnego, wykorzystującego dodatkowe pakiety instalacyjne, które nie brane były wcześniej pod uwagę. Wprowadzenie tej zmiany wyeliminowało krytyczne defekty spowodowane, jak się okazało, tym, że testerzy nie dysponowali owymi dodatkowymi pakietami instalacyjnymi.

Proces testowania zyskał dodatkowe korzyści z alertów dostarczanych przez mechanizm predykcji defektów. Biorąc pod uwagę dostarczane przezeń predykcje, zespół testujący stosował dodatkowe przypadki testowe względem funkcjonalności obsługiwanych przez pliki sklasyfikowane jako defektogenne. Nawet jeśli testerzy nie posiadali dokładnych informacji o zawartości kodu C#, świadomość ryzyka powodowała bardziej staranne testowanie.

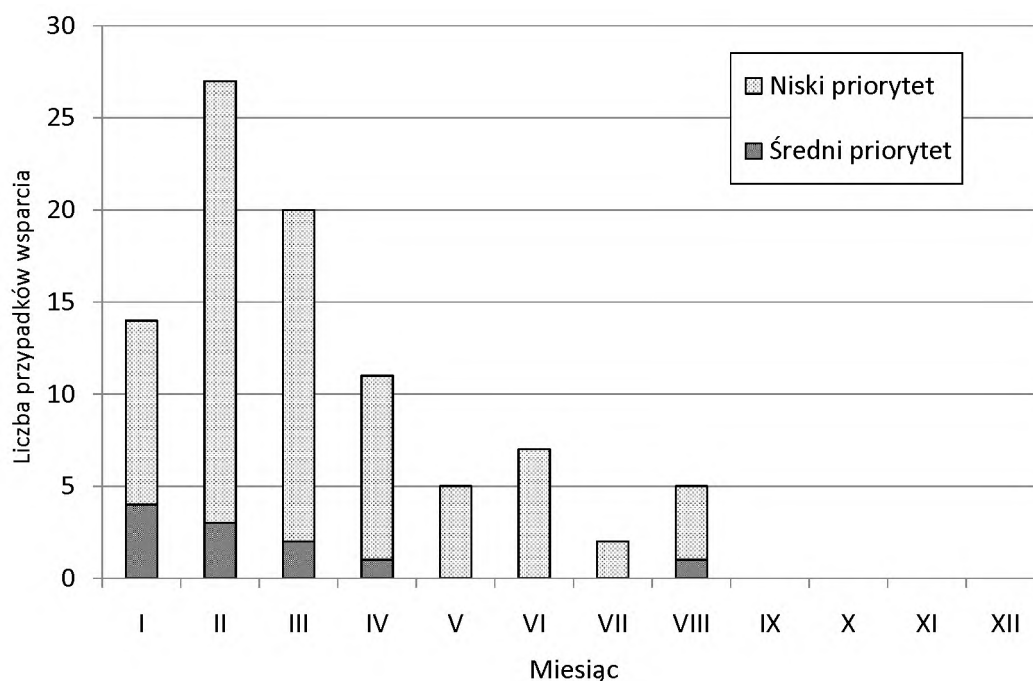
W tym miejscu może nasuwać się pytanie o to, że – skoro alert generowany przez „wtyczkę” do środowiska programistycznego wymuszał stosowanie dodatkowych zabiegów zapewniania jakości przez programistów – dlaczego by po prostu nie zmienić sposobu pracy programistów, czyli pracować tak, jakby dla każdego miejsca w kodzie wyświetlał się alert? Należy jednak pamiętać, że termin wdrożenia każdej nowej wersji systemu ROPTIS jest nieprzekraczalny, a głównym zadaniem programistów jest implementacja nowych funkcjonalności oprogramowania, na co czekają użytkownicy końcowi. Analiza ryzyka pokazała, że gdyby traktować każdą naprawę defektu „ze szczególną starannością”, a nie tylko te defekty, które dotyczą kodu wskazanego przez klasyfikator, byłoby wysoce prawdopodobnym, że nieprzekraczalny *deadline*, ustalany zawsze na koniec roku kalendarzowego, nie zostałby dotrzymany, lub nie wszystkie defekty o wysokiej dotkliwości zostałyby naprawione przed wdrożeniem.

5.6.4. Zaobserwowany wpływ zastosowania predykcji defektów na jakość oprogramowania komercyjnego

Tabela 5.5: Liczba defektów w porównywanych wersjach ROPTIS.

Dotkliwość defektu	Wersja A	Wersja B	Różnica
Krytyczna	162	56	-65%
Wysoka	188	101	-48%
Średnia	248	152	-38%
Niska	200	110	-45%
Wszystkie defekty	798	419	-47%
W tym: defekty naprawione niewłaściwie	83	23	-72%

Aby zaobserwować wpływ predykcji defektów w proponowanej formie na jakość oprogramowania wytwarzanego komercyjnie, porównana została liczba defektów, podzielonych ze względu na stopień dotkliwości, w dwóch badanych wydaniach systemu ROPTIS: wydanie A (kiedy zastosowano „starą” metodykę STPBQA) i wydanie B (z zastosowaniem „nowej” metodyki DPDQA). Wartości zostały obliczone, gdy zakończył się cykl życia wydania B i użytkownicy rozpoczęli używanie kolejnej wersji oprogramowania (nazwijmy ją wydaniem C). Tym samym, w wydaniu B nie można było już wykryć żadnych defektów. W wynikach, przedstawionych w tabeli 5.5, zaobserwowano 47,5% spadek całkowitej liczby defektów odkrytych podczas cyklu życia wydania B, gdy zastosowano metodykę DPDQA. Warto zauważyć, że zaobserwowane znaczące spadki liczby krytycznych i poważnych defektów w wydaniu B wynosiły odpowiednio 65% i 48%. Wpływ zaobserwowanych zmian jakościowych na użytkowników końcowych można zmierzyć za pomocą liczby opisanych w punkcie 5.2.2 „przypadków wsparcia” zarejestrowanych w wyniku zgłoszeń złożonych przez użytkowników po wdrożeniu wersji B systemu ROPTIS na środowisko produkcyjne. Liczba przypadków wsparcia obliczona na miesiąc, kiedy wydanie B było używane przez użytkowników końcowych, jest przedstawiona na wykresie na rys. 5.7. W porównaniu z podobnym wykresem sporządzonym na podstawie liczby przypadków wsparcia dla wydania A opracowanego przy użyciu „starej” metodyki STPBQA (rys. 5.1) można zaobserwować brak przypadków wsparcia o priorytecie krytycznym oraz spadek liczby przypadków o priorytecie wysokim. Dodatkowo, począwszy od września, można zauważyć, że



Rysunek 5.7: Liczba przypadków wsparcia dla wersji B systemu ROPTIS.

liczba przypadków wsparcia spada do zera¹⁰. Fakt ów można interpretować jako osiągnięcie bardzo wysokiej jakości produktu końcowego wydanego użytkownikom końcowym.

Dodatkowo zaobserwowano znaczny spadek liczby nieudanych testów potwierdzających¹¹. Test potwierdzający kończy się niepowodzeniem, gdy defekt zostaje oznaczony przez programistę jako „naprawiony”, a następnie nie przechodzi testu wykonanego przez testera oprogramowania w celu potwierdzenia usunięcia defektu. Zmniejszenie liczby nieudanych testów potwierdzających także można interpretować jako pozytywny skutek wdrożenia metodyki wykorzystującej wytwarzanie sterowane przypadkami testowymi i predykcję defektów.

Z uwagi na znaczącą, pozytywną zmianą w jakości oprogramowania, która została określona przez spadek liczby defektów, jak również i „przypadków wsparcia”, postanowiono dodatkowo zbadać możliwe zmiany w wydajności produktu. W tym celu porównano liczbę defektów związanych z testami wydajności w dwóch rozważanych wersjach oprogramowania (tabela 5.6). Również w tym obszarze zaobserwowano zauważalny spadek wykrytych defektów

¹⁰ Autor rozprawy, początkowo nie dowierając takim wynikiem, kilkakrotnie weryfikował, czy nie doszło np. do awarii systemu rejestrującego przypadki wsparcia lub celowego usunięcia tychże. Z uwagi na istotność projektu ROPTIS dla całego koncernu, gdzie prowadzono badania, prawdopodobieństwo wystąpienia takich sytuacji jest ekstremalnie niskie.

¹¹ Test potwierdzający to test wykonywany w celu potwierdzenia naprawy defektu przez programistę.

Tabela 5.6: Liczba defektów związanych z wydajnością programu.

Dotkliwość defektu	Wersja A	Wersja B
Krytyczna	4	0
Wysoka	1	1
Średnia	2	2
Niska	1	0

Tabela 5.7: Zaobserwowane zmiany w wydajności po wprowadzeniu proponowanej metodyki.

Zadanie testowe	Wersja A Średni czas [s]	Wersja B Średni czas [s]	Różnica
Otwórz sesję	1.8	1.9	6%
Zmodyfikuj sesję	34.5	21.0	-39%
Uruchom, zmodyfikuj i opuść	43.7	25.6	-41%
Zmodyfikuj i opuść	32.8	19.2	-41%
Sekwencja modyfikacji	52.6	34.6	-34%
Umożliw pracę	55.1	43.0	-22%
Uruchom, zachowaj i opuść	22.4	13.8	-38%
Zachowaj i wyślij	57.2	56.2	-2%
Wyszukaj i rozszerz	1.9	2.0	5%
Wyszukaj w archiwum	0.4	0.2	-50%
Wyszukaj i wyświetl	0.4	0.3	-25%

i nie znaleziono żadnych defektów o krytycznym znaczeniu. W następnym kroku porównano wyniki testów wydajności, mierzone w sekundach, które odzwierciedlają czas trwania testów wydajnościowych zastosowanych do krytycznych funkcji systemu (tabela 5.7). Poza dwoma przypadkami, w których zaobserwowano niewielkie wzrosty czasu (prawdopodobnie z powodu dodania w wydaniu B dodatkowych funkcjonalności), zaobserwowano pozytywną zmianę w wydajności. Więcej przekonujących dowodów na zasadność stosowania proponowanej metodyki zapewniania jakości znaleziono przez porównanie wyników testów dotyczących wybranych kwestii związanych z wydajnością. W niektórych przypadkach zaobserwowano znaczące obniżenie czasu działania wybranych testów wydajnościowych aż o 98% (tabela 5.8).

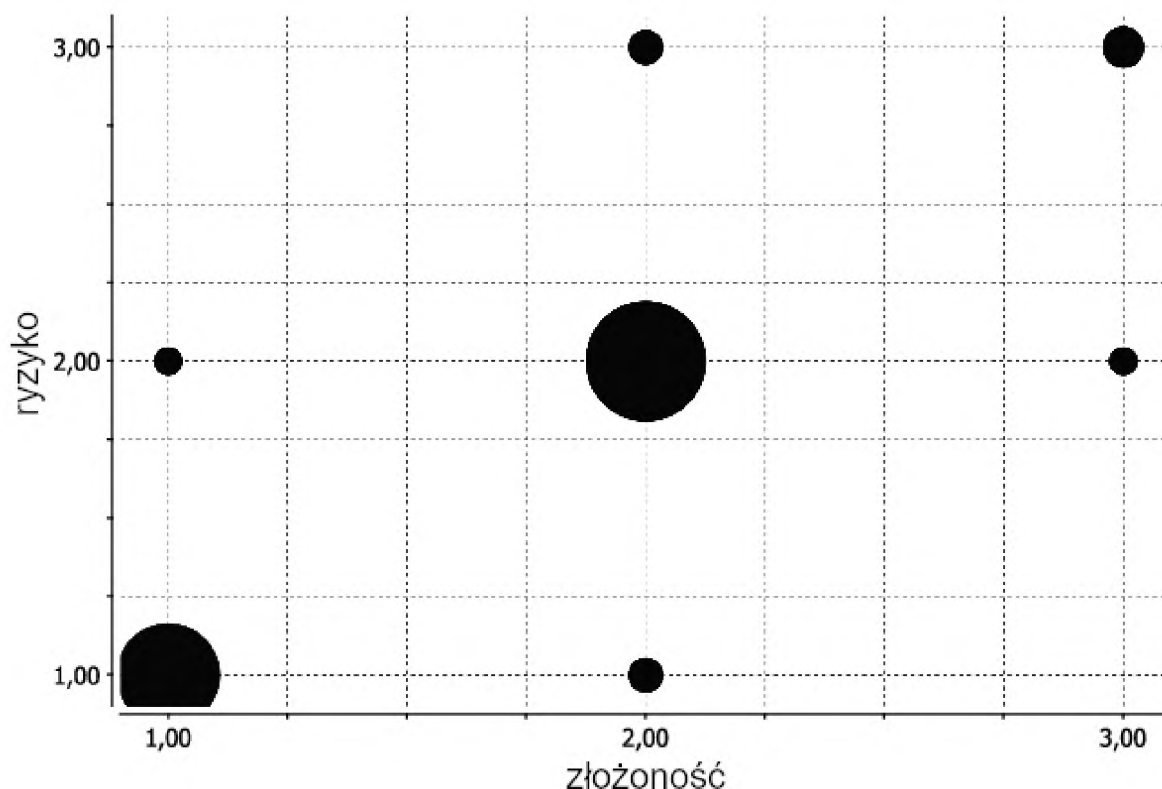
Zaobserwowany wzrost jakości nie powinien być jednakowoż przypisany tylko i wyłącznie na konto proponowanej metodyki, ponieważ techniki zapewniania jakości (w znacznie

Tabela 5.8: Dziesiątka największych przyrostów wydajności w wersji B.

Oryginalna nazwa testu wydajności	Wersja A Średni czas [s]	Wersja B Średni czas [s]	Różnica
TestCheck	73.8	1.68	-98%
TestMultipleTests	72.6	1.56	-98%
TestProgrammingPlus3	72.0	1.50	-98%
TestAlterParam	72.0	1.50	-98%
TestProgramming	72.0	1.50	-98%
TestMultipleTestsPlus	72.0	1.50	-98%
TestProgrammingPlus2	72.0	1.50	-98%
TestFaultCodes	71.4	1.44	-98%
TestProgrammingPlus1	71.4	1.62	-98%
TestCheckPlus	70.8	1.50	-98%

mniej stopniu wynikającym z ograniczeń czasowych, ale jednak) były obecne przed fazą testów systemowych (kiedy pisany był kod implementujący nowe funkcje). Może to być interpretowane jako walidacja wewnętrzna zagrożeń powiązanych z zaobserwowanymi wynikami [33]. Jednakże wybrane techniki, a w szczególności zasada „zostaw kod w lepszym stanie, niż go zastałeś” (kiedy cały plik źródłowy staje się obiektem refaktoryzacji, nawet gdy tylko jego część musi zostać zmieniona w celu naprawy defektu), mogą być także traktowane jako źródło dodatkowych defektów (pomijając aspekt wprowadzania „szumu” do danych wejściowych dla modelu predykcji – sposób przeciwdziałania temu przedstawiono w punkcie 5.4.4). Tych ewentualnych defektów można było uniknąć, jeżeli kod byłby zmieniany tylko tam, gdzie to konieczne.

Dodatkowo, jednym z możliwych zagrożeń dla walidacji statystycznej wniosków (błędne wnioski o relacji między zabiegami a wynikiem eksperymentów [217]) jest obserwacja faktu, iż w porównaniu z wersją A w wersji B znaleziona została znacznie mniejsza liczba defektów. Czyżby znacznie mniejsza liczba defektów wynikała z możliwości, że w wersji B wprowadzono znacznie mniej elementów, które mogłyby wprowadzać defekty? Aby zweryfikować tę możliwość, dokonano porównania obu wyżej wymienionych wydań w kontekście wprowadzanych nowych funkcjonalności. Każda nowa funkcjonalność, implementowana w obu wersjach, była sklasyfikowana pod względem ryzyka i złożoności (skomplikowania), z wykorzystaniem trzystopniowej skali: od 1 (małe ryzyko lub złożoność) do 3 (duże ryzyko lub złożoność). Każdą funkcjonalność naniesiono następnie na wykresie ryzyko-złożoność i porównano wyniki. Liczba zaimplementowanych funkcji w

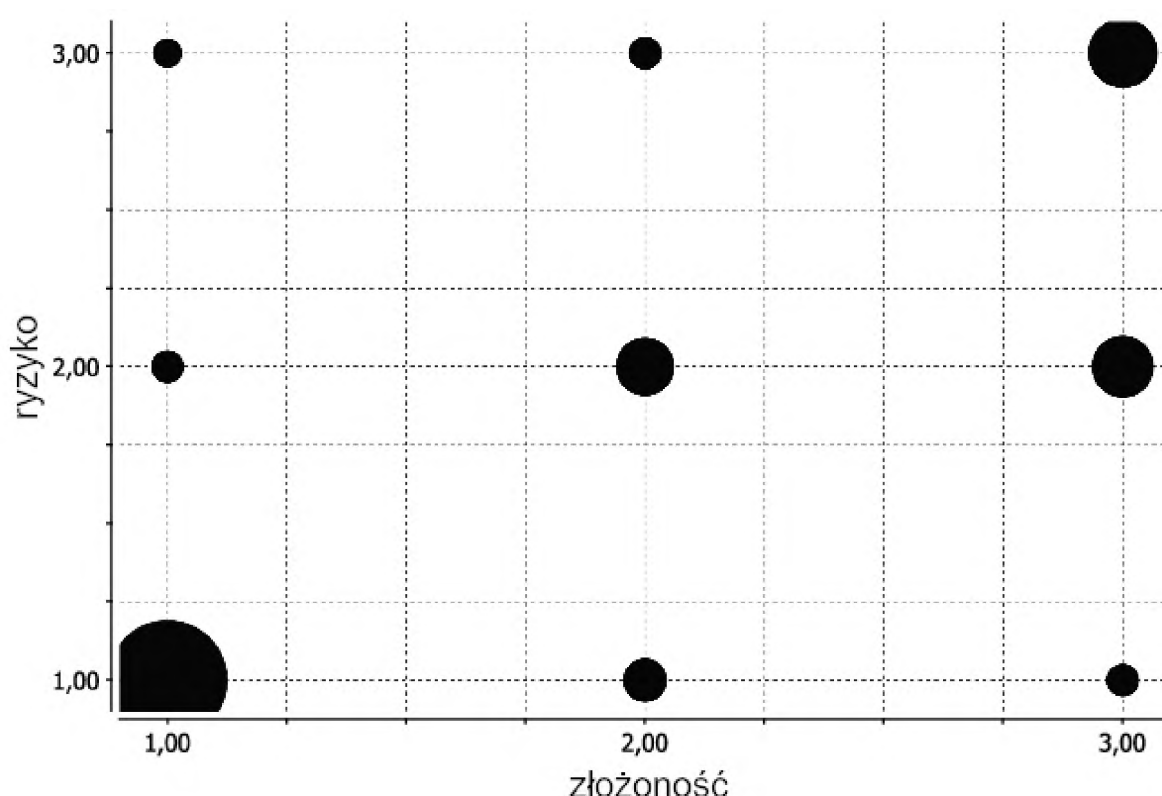


Rysunek 5.8: Charakterystyka funkcji wprowadzonych w wersji A systemu ROPTIS. Średnica koła jest proporcjonalna do całkowitej liczby funkcji zaklasyfikowanych do tej samej grupy ryzyka i złożoności.

obu przypadkach była podobna; w przypadku wydania A można zauważyć, że większość funkcji została sklasyfikowana jako te o niskim ryzyku z niskim stopniem złożoności i o średnim ryzyku ze średnim stopniem złożoności (rys. 5.8). W wersji B zaobserwowano podobną liczbę funkcji z niskim ryzykiem i niskim poziomem złożoności, ale w porównaniu z wersją A, nastąpiło niejako „rozproszenie” naniesionych, sklasyfikowanych funkcjonalności na prawą-górną stronę wykresu – więcej funkcjonalności było sklasyfikowanych jako te o wysokiej złożoności i średnim i wysokim ryzyku (rys. 5.9). Należałoby zatem oczekiwać, że w przypadku wersji B zaobserwujemy więcej defektów, ponieważ zaimplementowano więcej „trudniejszych” funkcjonalności. Tymczasem mniejsza liczba defektów oraz „przypadków wsparcia” zaobserwowanych w wersji przygotowywanej z zastosowaniem proponowanej metodyki zapewniania jakości wskazuje na jej skuteczność.

Jednak pomimo zastosowania metodyki DPDQA, zespół testowy ROPTIS nadal otrzymywał kod z defektami o wysokiej i krytycznej dotkliwości. Znalaziono dwa możliwe źródła takich sytuacji: 1) przepływy programowe nieobjęte wymaganymi przypadkami testowymi, co wynika z braku wiedzy biznesowej lub po prostu zwykłych ludzkich błędów oraz 2) bardziej kłopotliwe: doświadczono defektów, które powinny być dość łatwo wychwycone podczas testów przeprowadzanych przez programistów. Po analizie wyciągnięto wniosek, że dostarczenie członkom zespołu programistycznego gotowego zestawu

przypadków testowych do wykonania, ograniczało samodzielne myślenie o innych możliwych scenariuszach testowych. Wykonanie dostarczonego zestawu testów zazwyczaj było uważane za wystarczające, aby zakończyć testowanie w tym momencie. Ten problem został rozwiązany podczas dalszego rozwoju (po wydaniu B): członkowie zespołu, w szczególności programiści, zostali zachęceni do szukania możliwych scenariuszy testowych, które nie były objęte przygotowanymi przypadkami testowymi. Gdy taki scenariusz zostaje znaleziony, jest on rejestrowany jako „możliwość ulepszenia”. Następnie, kolejnym krokiem jest analiza: oszacowanie wysiłku i potencjalnego ryzyka związanego z niewykonaniem nowego scenariusza. Jeżeli korzyści są potwierdzone, utworzony nowy scenariusz testowy jest traktowany priorytetowo i zarządzany jako nowy przypadek (lub przypadki) testowy.



Rysunek 5.9: Charakterystyka funkcji wprowadzonych w wersji B systemu ROPTIS. Średnica koła jest proporcjonalna do całkowitej liczby funkcji zaklasyfikowanych do tej samej grupy ryzyka i złożoności.

Opisane w tym rozdziale podejście do programowania sterowanego przypadkami testowymi dla testów wyższych poziomów (*test case-driven development*), które okazało się być skutecznym narzędziem zapewniania jakości (np. z uwagi na wpływ, jakie miało na zredukowanie niewłaściwego naprawiania defektów, por. tabela 5.5) jest mało odzwierciedlone w pracach z dziedziny inżynierii oprogramowania. Z tego powodu technika ta powinna być dalej dokładnie badana, szczególnie w porównaniu z innymi podejściami, jak to ma miejsce w przypadku „klasycznego” programowania sterowanego testami (*test-driven development*) [124, 118, 119].

Opisane w tym rozdziale badania pozwoliły zrealizować następujące cele szczegółowe, nakreślone w punkcie 1.2 niniejszej pracy:

Cel 6: Wdrożono predykcję defektów jako istotny element procesu zapewniania jakości w komercyjnym projekcie rozwoju oprogramowania i zaproponowano autorską metodykę zapewniania jakości oprogramowania wykorzystującą predykcję defektów oprogramowania.

Cel 7: Zaproponowano sposoby zbadania, w jaki sposób metodyka wykorzystująca predykcję defektów wpłynęła na jakość wytwarzanego przemysłowo oprogramowania.

Cel 8: Kontynuowano też rozpoczęte wcześniej badania, jak wybrane rozwiązania z obszaru uczenia maszynowego (klasyfikatory, algorytmy wyboru cech, algorytmy przeciwdziałania niezrównoważeniu klas itp.) osiągają najlepsze rezultaty w szczególnym środowisku, jakim są komercyjne projekty wytwarzania oprogramowania.

6. Modyfikacja procesu predykcji defektów uwzględniająca zapachy kodu

Propagator ciągłego doskonalenia procesów wytwórczych, osoba, której przypisuje się odpowiedzialność za sukces wielu japońskich przedsiębiorstw [151], William Edwards Deming, stwierdził ongiś, że „nie wystarczy pracować dobrze, musimy wiedzieć co jest do zrobienia i potrafić zrobić to lepiej”. Podążając za tą wskazówką, należało się zastanowić, czy jest coś, co można by usprawnić w zaprezentowanej w poprzednim rozdziale predykcji defektów?

6.1. Cel modyfikacji

Jak wspomniano na wstępie (punkt 2.1.1), spośród wszystkich aspektów związanych z predykcją defektów oprogramowania, jednym z kluczowych elementów jest odpowiedni dobór metryk wykorzystywanych do zbudowania modelu predykcji. Najbardziej obiecującymi typami metryk są metryki kodu źródłowego i metryki procesowe. W toku dotychczas opisywanych badań wykorzystywano tylko metryki kodu; w opisaney w rozdziale 5 predykcji defektów, będącej kluczowym elementem metodyki zapewniania jakości, wykorzystano zaledwie pięć metryk kodu źródłowego. Czy jest możliwość podniesienia efektywności predykcji stosunkowo niskim kosztem, tj. poprzez rozszerzenie danych wejściowych o nowe metryki, w tym – metryki procesowe?

Jedna z grup metryk procesowych zawiera metryki bazujące na tak zwanych zapachach kodu (ang. *code smells*). Termin ten został zaproponowany przez Kenta Becka w 2006 roku [14], a następnie spopularyzowany przez Martina Fowlera w książce „Refactoring. Improving the Structure of the Existing Code” [50], w której to właśnie Kent Beck był współautorem rozdziału o zapachach kodu. Autor ów, na swojej stronie internetowej, wyjaśnił całą ideę stojącą za tym terminem: *Należy zauważyć, że zapach kodu oznacza, że coś może być nie tak [z kodem źródłowym], aczkolwiek nie można tego uznać za pewnik. Perfekcyjny [kod] może być uznany za zapach kodu, ponieważ często jest on używany w nieodpowiedni sposób lub istnieje prostsza alternatywa, działająca w większości przypadków. Nazwanie czegoś zapachem kodu to nie atak. Jest to po prostu znak, że dokładne przyjrzenie się kodowi jest uzasadnione* [14].

Ze względu na taką naturę zapachu kodu, wciąż dyskusyjnym pozostaje, czy może on zostać użyty w sposób efektywny w zapewnianiu jakości wytwarzania oprogramowania [224, 221, 107, 3]. W przypadku obszaru zapewniania jakości będącego obiektem badań opisywanych w niniejszej pracy, czyli predykcji defektów oprogramowania, Piotrowski i Madeyski dokonali systematycznego przeglądu literatury pod kątem zastosowania tamże zapachów kodu [165]. Na podstawie wyników ich pracy można uznać, że znakomita większość publikacji skupia się na badaniach projektów otwarto-źródłowych, wykorzystujących język programowania Java [61]. Opracowań dotyczących zastosowania zapachów kodu w predykcji defektów w *komercyjnym* wytwarzaniu oprogramowania istnieje znikoma liczba – spośród najważniejszych można wymienić badania Holschuh et al. nad skutecznością użycia metryk zapachów kodu w predykcji defektów w projekcie komercyjnym wykorzystującym język Java [75], oraz prace Soltanifar et al. nad modelem predykcji defektów używającym zapachów kodu z dwóch projektów komercyjnych, pisanych w językach PHP, JavaScript oraz Java [194].

Tym samym nie można wykluczyć, że nie istnieje żadne naukowe opracowanie zastosowania metryk zapachów kodu do predykcji defektów oprogramowania w projekcie komercyjnym, używającym technologii .NET (wykorzystującej język C#) – a takim właśnie jest opisywany wcześniej ROPTIS.

Ponieważ w projekcie ROPTIS, jak opisano w poprzednim rozdziale, aktywnie wykorzystuje się predykcję defektów w metodyce zapewniania jakości oprogramowania, stanowiło to okazję do przeprowadzenia eksperymentu z zastosowaniem zapachów kodu źródłowego języka C# do predykcji defektów, aby odpowiedzieć na pytanie: w jaki sposób stosowanie metryk bazujących na zapachach kodu może wpływać na predykcję defektów w komercyjnym projekcie wytwarzania oprogramowania, jakim jest ROPTIS?

6.2. Proces badawczy

W eksperymencie, mającym na celu zbadanie potencjalnego wpływu metryk zapachów kodu na predykcję defektów stworzono następujący plan działania, który można podzielić na dwie części.

Część pierwsza:

1. Wygenerowanie metryk (wymienionych w punkcie 5.5.1) kodu źródłowego wycofanej wersji oprogramowania ROPTIS¹ za pomocą narzędzia Microsoft SourceMonitor (podobnie, jak robiono to wcześniej).
2. Dla tego samego kodu źródłowego wygenerowanie metryk zapachów kodu za pomocą dodatkowego, specjalnego narzędzia – CodeAnalysis.

¹ Czyli takiej wersji, dla której mielibyśmy pełną informację o kodzie rzeczywiście defektogennym – w celu przeprowadzenia walidacji modelu predykcji. Trzymając się nomenklatury z rozdziału 5, byłaby to wersja poprzedzająca wersję „A”.

3. Połączenie zmian w kodzie źródłowym z wywołującymi je defektami (przy założeniu, że jeżeli kod został zmieniony w wyniku naprawy defektu, to przed modyfikacją był to kod defektogenny).
4. Połączenie informacji o modułach kodu źródłowego – klasach – zarówno defektogennych, jak i nie-defektogennych, z odpowiadającymi im metrykami kodu z narzędzi SourceMonitor i/lub CodeAnalysis. Stworzenie tym samym dwóch zbiorów danych: jednego, który zawiera oba rodzaje metryk (SourceMonitor i CodeAnalysis) oraz drugiego, „klasycznego”, który zawiera metryki jedynie z SourceMonitor.
5. Podział powyższych zbiorów danych za pomocą próbkowania warstwowego na podzbiory – uczący i ewaluacyjny.
6. Wykorzystanie różnych kombinacji technik do zbudowania modelu predykcji, a następnie jego ewaluacji na dwóch rodzajach danych – zawierających metryki zapachów kodu i nie zawierających ich.
7. Zebranie wyników i wyłonienie takiej kombinacji technik, która skutkowałą najlepszym wynikiem predykcji.

Na tym skończy się pierwszy etap eksperymentu. W drugim etapie plan zakłada użycie do procesu predykcji najlepszej kombinacji technik, wyłonionej w etapie pierwszym.

Część druga planu zakłada:

1. Podzielnice klas kodu źródłowego, składających się na badaną tutaj, wycofaną wersję oprogramowania ROPTIS, na 20 części.
2. Na podstawie powyższego podziału skonstruowanie zbiorów danych zawierających tylko metryki SourceMonitor, tylko metryki CodeAnalysis oraz oba typy metryk (w sumie 60 zbiorów danych).
3. Zbudowanie modelu predykcji defektów a następnie jego walidacja dla wszystkich zbiorów danych z użyciem (wytypowanego w pierwszej części) mechanizmu wyboru cech, oraz bez użycia mechanizmu wyboru cech.
4. Zebranie i interpretację wyników.

6.2.1. Narzędzie CodeAnalysis jako źródło metryk związanych z zapachami kodu

W niniejszym eksperymencie, oprócz dotychczas wykorzystywanego narzędzia SourceMonitor (część środowiska Microsoft Visual Studio) do uzyskania metryk kodu źródłowego projektu ROPTIS skorzystano także z nowego narzędzia, jakim jest Microsoft CodeAnalysis, celem zebrania metryk zapachów kodu. Czynnikiem decydującym były koszty: CodeAnalysis, podobnie jak SourceMonitor, jest jedną ze składowych oprogramowania Microsoft Visual Studio dla projektów bazujących na technologii .NET. Dlatego też nie pojawiły się żadne dodatkowe koszty związane z wprowadzeniem tego narzędzia do projektu. Twórca narzędzia CodeAnalysis, firma Microsoft, opisuje go następująco: *CodeAnalysis*

Tabela 6.1: Ostrzeżenia dotyczące zapachów kodu w narzędziu CodeAnalysis

Kategoria ostrzeżenia	Czego dotyczy
Projekt	Przestrzeganie wytycznych .NET Framework Design Guidelines w pisanych bibliotekach
Globalizacja	Uniezależnienie bibliotek i aplikacji od lokalizacji geograficznej
Współwymienialność	Współpraca z klientami COM
Utrzymywalność	Utrzymanie bibliotek i aplikacji
Mobilność	Efektywne wykorzystanie energii przez urządzenia mobilne
Nazewnictwo	Zgodność z konwencjami nazewnictwa zawartymi w wytycznych projektowych .NET Framework
Wydajność	Wysokowydajne biblioteki i aplikacje
Przenaszalność	Możliwość przenoszenia na różne platformy
Niezawodność	Niezawodność bibliotek i aplikacji, np. prawidłowe wykorzystanie pamięci i wątków
Bezpieczeństwo	Bezpieczniejsze biblioteki i aplikacje
Użytkowanie	Właściwe użytkowanie .NET Framework

gromadzi i dostarcza informacje, takie jak naruszanie zasad projektowych i programowania ustanowionych w poradnikach projektowania oprogramowania Microsoft [136]. Zatem narzędzie wydaje się jak najbardziej odpowiednie do identyfikacji zapachów kodu, szczególnie w projekcie operującym technologiami firmy Microsoft, jakim jest projekt ROPTIS. Zgodnie z dokumentacją, w CodeAnalysis znajduje się około 200 zasad [136], które wywołują ostrzeżenia o zapachach kodu podzielone na 11 kategorii (tabela 6.1). Narzędzie to może być uruchomione z linii komend, a wyniki są magazynowane w pliku XML, który może następnie zostać wczytany do narzędzia DePress i poddany dalszej analizie.

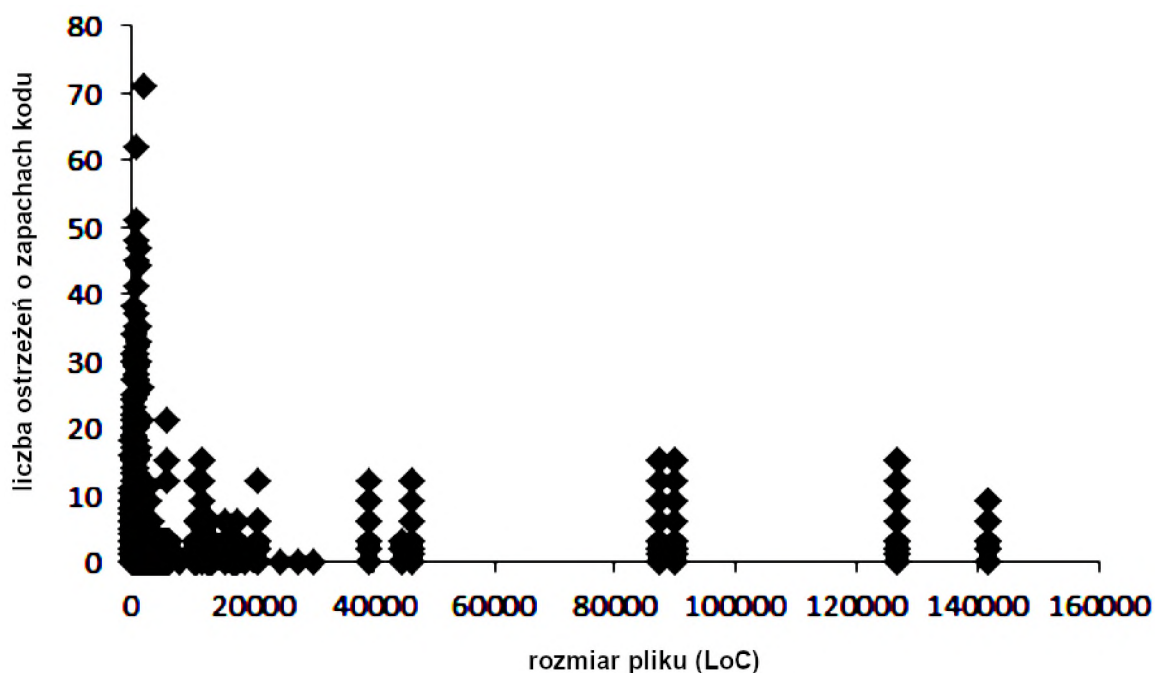
6.3. Wyniki

W niniejszym podrozdziale zaprezentowano wyniki eksperymentu przeprowadzonego zgodnie z planem przedstawionym wcześniej.

6.3.1. Zaobserwowane anomalie wynikające z automatycznej generacji kodu

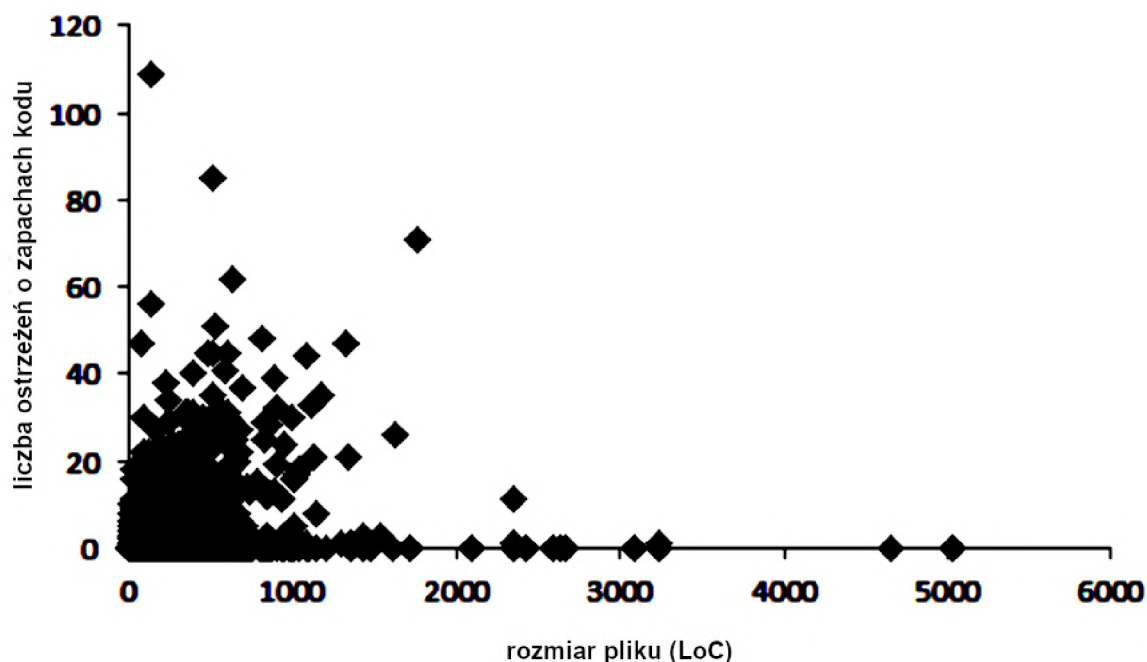
Po przeanalizowaniu relacji między liczbą ostrzeżeń związanych z zapachami kodu i metryką liczby linii kodu dla kompletnego systemu w połączonych zbiorach danych pobranych z narzędzi CodeAnalysis i SourceMonitor zauważono, że różna liczba ostrzeżeń CodeAnalysis

występuje dla tych samych wartości rozmiaru plików (rys. 6.1). Jako że analizowane oprogramowanie posiadało relatywnie niewielką liczbę dużych objętościowo plików, pozwalało to potraktować te dane jako anomalie: różna liczba ostrzeżeń o zapachach kodu pojawiła się w wynikach dla tych samych, dużych plików. Po przeanalizowaniu sytuacji okazało się, że analizowane pliki systemowe, zawierające więcej niż 1000 linii kodu (LOC), są generowane automatycznie i zawierają więcej niż jedną klasę na plik, podczas gdy CodeAnalysis zlicza liczbę ostrzeżeń na klasę, nie plik. Ta rozbieżność skutkowała nietypową liczbą defektów w stosunku do metryki linii kodu LOC: różna liczba ostrzeżeń o zapachach kodu pojawiała się dla tych samych wartości LOC. Przyczyną tej sytuacji było to, że wartość liczby ostrzeżeń wyliczana była dla różnych klas zlokalizowanych w tych samych plikach, określanych przez taką samą liczbę LOC.



Rysunek 6.1: Anomalie w wartościach metryki „liczba ostrzeżeń o zapachach kodu” per rozmiar pliku (mierzony w LOC), wprowadzone w wyniku stosowania automatycznie generowanego kodu; później pominięte w analizie.

Ponieważ automatycznie generowane pliki kodu (zawierające wiele klas w jednym pliku) istnieją tylko na potrzeby instalacji i wdrożenia, oraz nie są one pokrywane przez testy ani nie są dostępne dla użytkownika końcowego, postanowiono potraktować je jako źródło szumu informacyjnego i usunąć z dalszej analizy. Po tym kroku zależność między liczbą ostrzeżeń a długością plików była poprawna (rys. 6.2).



Rysunek 6.2: Finalny rozkład wartości „liczba ostrzeżeń o zapachach kodu” per rozmiar pliku (mierzony w LOC) po wprowadzeniu poprawek.

6.3.2. Różnice w metrykach i tego konsekwencje

Po analizie wyżej przedstawionego problemu odkryto, że różnice w liczbie ostrzeżeń dla tej samej wartości LOC były spowodowane faktem, że narzędzia, których użyto do gromadzenia metryk, stosowały różne metody ich zliczania: CodeAnalysis oblicza metryki dla każdej z klas kodu źródłowego, a SourceMonitor dla każdego pliku. Przy pierwszym podejściu do połączenia danych z obu narzędzi w jeden zbiór, metryki SourceMonitor (czyli „zwykłe” metryki kodu źródłowego), podawane dla każdego pliku, zostały podzielone przez liczbę klas w tym pliku, aby mogły być „rozdzielone pomiędzy” rekordy danych wygenerowane przez CodeAnalysis (zawierające metryki zapachów kodu). Podejście to ilustruje tabela 6.2.

Aby przeciwdziałać anomaliiom związanym z metrykami opisanym w punkcie 6.3.1 oraz możliwemu pojawieniu się szumu informacyjnego w badanym zbiorze danych, zmieniono podejście i układ danych tak, aby w wynikowym zbiorze zawierającym połączone metryki z obu narzędzi, pojedynczy rekord danych opisywał jeden plik kodu źródłowego. Aby to osiągnąć, metryki zapachów kodu generowane przez CodeAnalysis były dodawane, jeśli w jednym pliku kodu źródłowego występowała więcej niż jedna klasa (tabela 6.3).

6.3.3. Wybór optymalnego procesu predykcji defektów

W celu wybrania optymalnego procesu predykcji zdecydowano się na przetestowanie kilku kombinacji: klasyfikatorów, algorytmów wyboru cech oraz wykorzystania (bądź nie) algorytmu

Tabela 6.2: Przykład wycinka zbioru danych z podejścia pierwszego – pojedyncza klasa per rekord (wartości metryk z narzędzia SourceMonitor są sztucznie podzielone przez liczbę klas w pliku źródłowym).

Plik	Klasa	SourceMonitor (LOC)	CodeAnalysis (liczba ostrzeżeń)
File1.cs	Class1	20	3
File1.cs	Class2	20	20
File1.cs	Class3	20	6
File2.cs	Class4	30	15

Tabela 6.3: Przykład zbioru danych z podejścia drugiego – pojedynczy plik per rekord (wartości metryk zapachów kodu zostały zsumowane dla plików wieloklasowych).

Plik	Klasa	SourceMonitor (LOC)	CodeAnalysis (liczba ostrzeżeń)
File1.cs	Class1...3	60	29
File2.cs	Class4	30	15

SMOTE (tabela 6.4) dla dwóch zbiorów danych: zawierającego metryki zapachów kodu i bez metryk zapachów kodu.

W przeciwieństwie do procesu opisanego w punkcie 5.5.3, w tym przypadku ograniczono się tylko do używania algorytmu SMOTE [27] celem przeciwdziałania niezbalansowaniu klas, ponieważ we wcześniejszej konfrontacji z drugą techniką – próbkowaniem klasy większościowej – wykazywał się lepszym wpływem na końcową jakość predykcji defektów.

Za to – znów w przeciwieństwie do wcześniej przyjętego podejścia, np. opisanego w punkcie 4.3.3 – w ramach eksperymentu zdecydowano się rozszerzyć mechanizm wyboru cech o dodatkowy algorytm. Oprócz wbudowanego w narzędzie DePress/KNIME algorytmu eliminacji wstecznej, użyto także, alternatywnie, meta-heurystycznego algorytmu symulowanego wyżarzania opisanego przez Kirpatricka et al. [102] w formie zaproponowanej przez Brownlee [23].

Tabela 6.4: Kombinacje różnych rozwiązań.

Klasyfikator	Wybór cech	SMOTE	Metryki zapachów kodu?
Naiwny Bayes	Brak	Tak	Tak
Las losowy	Eliminacja	Nie	Nie
PNN	Wyżarzanie		

Jako możliwe klasyfikatory wykorzystano, podobnie jak wcześniej, popularne w wielu badaniach nad predykcją defektów [67, 143, 94, 182]: naiwny klasyfikator Bayesa, probabilistyczną sieć neuronową (PNN) i klasyfikator lasu losowego [22], z uwagi na ich dostępność i łatwe użycie (co przekłada się na badany wcześniej zwrot z inwestycji) w narzędziu DePress.

Wyniki uzyskane za pomocą badanych kombinacji zostały zaprezentowane w tabeli 6.5. Dwa zbiory danych – zawierające i niezawierające metryk zapachów kodu zostały podzielone za pomocą próbkowania warstwowego (*stratified sampling*) na dwa podzbiory, pierwszy do budowania określonych modeli predykcji, drugi do jego ewaluacji. Modele predykcji, podobnie jak we wcześniejszych badaniach, zostały ocenione przy wykorzystaniu współczynnika korelacji Matthews (MCC), opisanego w punkcie 2.2.1.

Najwyższa (choć wciąż zaskakująco niska) wartość współczynnika korelacji Matthews (0,0775) została zaobserwowana gdy zastosowano następującą kombinację: zbiór danych wykorzystujący metryki zapachów kodu, algorytm SMOTE, wybór cech wykorzystujący algorytm eliminacji wstecznej, oraz las losowy jako klasyfikator. Mimo i tak niskiej wartości współczynnika, ocena ta pozwoliła wybrać powyższą kombinację do drugiej części badań, czyli ostatecznej oceny wykorzystania zapachu kodu w procesie predykcji defektów.

6.3.4. Ewaluacja zbiorów danych: metryki zapachów kodu w porównaniu do metryk kodu źródłowego

Dla ostatecznej oceny, czy metryki zapachów kodu mogły pozytywnie wpłynąć na efektywność predykcji defektów, dane opisujące system ROPTIS (opisano je w punkcie 5.5.1) – czyli listę klas kodu źródłowego wraz z informacją o defektach, klasa per rekord – podzielono na 20 mniejszych, podobnych rozmiarowo podmodułów (około 700 rekordów). Dokładniejszy podział kodu nie był możliwy z czysto technicznych względów. Dla każdego podmodułu zebrano metryki za pomocą SourceMonitor i/lub CodeAnalysis, celem zbudowania różnych zbiorów danych:

- 20 zbiorów danych tylko z metrykami SourceMonitor (metryki kodu źródłowego);
- 20 zbiorów danych z metrykami CodeAnalysis (metryki zapachów kodu);
- 20 łączonych zbiorów metryk: SourceMonitor + CodeAnalysis.

Co więcej, każdy typ zbiorów danych został poddany procesowi wyboru cech. Podczas ewaluacji modelu predykcji zebrano wartości współczynnika korelacji Matthews (MCC), miarę F i czułości dla klas podatnych na defekty (tabela 6.6).

Tabela 6.5: Wyniki tej części eksperymentu, której celem było ustalenie najlepszego zestawienia rozwiązań używanych w procesie predykcji (dla klasy „defektogennej”).

Klasyfikator	SMOTE	Wybór cech	Metryki zapachów kodu dołączone?									
			Nie					Tak				
			MCC	TP	FP	TN	FN	MCC	TP	FP	TN	FN
Naiwny Bayes	Nie	Wyżarzanie	-0,6133	23	161	3330	142	-0,7559	15	81	3410	150
		Eliminacja	-0,9968	0	1	3490	165	-1	0	0	3491	165
		Brak	-0,4823	31	276	3215	134	0,3948	40	371	3120	125
	Tak	Wyżarzanie	-0,4525	1497	482	3008	1994	-0,3611	1695	599	2891	1796
		Eliminacja	-0,3505	1736	598	2892	1755	-0,3237	1807	621	2869	1684
		Brak	-0,4603	1476	475	3015	2015	-0,3738	1657	591	2899	1834
PNN	Nie	Wyżarzanie	-1	0	0	3491	165	-1	0	0	3491	165
		Eliminacja	-1	0	0	3491	165	-1	0	0	3491	165
		Brak	-1	0	0	3491	165	-1	0	0	3491	165
	Tak	Wyżarzanie	-0,3404	2187	303	3187	1304	-0,2877	2335	333	3157	1156
		Eliminacja	-1	0	0	3491	165	-0,2256	2568	320	3170	923
		Brak	-0,3597	2147	282	3208	1344	-0,3305	2204	316	3174	1287
Las losowy	Nie	Wyżarzanie	-0,9309	9	13	3478	156	-0,9048	15	15	3476	150
		Eliminacja	-0,9227	15	9	3482	150	-0,9356	13	7	3484	152
		Brak	-0,9025	14	17	3474	151	-0,9048	15	15	3476	150
	Tak	Wyżarzanie	0,0427	3364	189	3301	127	0,0382	3407	130	3360	84
		Eliminacja	0,0321	3390	142	3348	101	0,0775	3435	147	3343	56
		Brak	0,0473	3383	173	3317	108	0,0382	3407	130	3360	84

Tabela 6.6: Pomiary dla rekordów określonych jako podatne na defekty.

Zbiór danych	Miara	Średnia	Odchylenie std.
SourceMonitor bez wyb. cech	Czułość	0.9608	0.0278
	miara F	0,9433	0,0188
	MCC	0,0832	0,0809
CodeAnalysis bez wyb. cech	Czułość	0.6660	0.2961
	miara F	0,6447	0,1157
	MCC	-0,0313	0,658
SourceMonitor + CodeAnalysis bez wyb. cech	Czułość	0.9637	0.0303
	miara F	0,9494	0,0228
	MCC	0,0733	0,0733
SourceMonitor z wyb. cech	Czułość	0.9824	0.0146
	miara F	0,9704	0,0120
	MCC	0,0725	0,066
CodeAnalysis z wyb. cech	Czułość	0.8424	0.0542
	miara F	0,8286	0,0559
	MCC	0,0446	0,0704
SourceMonitor + CodeAnalysis z wyb. cech	Recall	0.9859	0.0206
	miara F	0,9792	0,0136
	MCC	0,0579	0,0742

6.4. Podsumowanie tego etapu badań

Najlepsze rezultaty, po przebadaniu 20 różnych podmodułów („części”) kodu źródłowego, odnotowano w zbiorze danych zawierającym jedynie „standardowe” metryki kodu (pobrane z narzędzia SourceMonitor), bez metryk zapachów kodu (tabela 6.6 – $MCC = 0,0832$, miara $F = 0.9433$). Jednakże dla tego samego zestawu eksperymentalnego, ale dla zbioru danych wzbogaconego o metryki zapachów kodu, wartość współczynnika korelacji Matthews’a była jedynie nieznacznie niższa, wynosząca $MCC = 0,0733$ przy odchyleniu standardowym wynoszącym 0,0733, co czyni różnicę między wynikami wykorzystującymi tylko metryki SourceMonitor oraz tymi, które także zawierają metryki zapachów kodu pochodzące z CodeAnalysis, nieistotną.

Co interesujące, w przypadku pierwszej części eksperymentu, której rezultaty umieszczono w tabeli 6.5, najwyższą wartość współczynnika MCC odnotowano dla połączonego zbioru danych wykorzystującego metryki „standardowe” i metryki zapachów kodu. Jak zauważono wcześniej, i tak jest to niska wartość, a wartości MCC dla zbioru danych bez metryk zapachów kodu są jedynie nieznacznie niższe (0,0473 i 0,0427).

Warty odnotowania jest spadek efektywności predykcji w pierwszej części eksperymentu, dla wybranego zestawu eksperymentalnego (czyli połączenie lasu losowego, algorytmu SMOTE i wyboru cech za pomocą eliminacji wstecznej) gdy proces metryki zapachów kodu zostały usunięte z zbioru danych.

Interesującym rezultatem, wartym dalszego zbadania, są jedynie ujemne wartości współczynnika korelacji Matthews’a (czyli sygnalizujące predykcję gorszą od klasyfikacji losowej) dla kombinacji innych, niż las losowy w połączeniu a algorytmem SMOTE.

Podsumowując, wyniki eksperymentu z wykorzystaniem metryk zapachów kodu w celu zwiększenia efektywności predykcji defektów (a tym samym polepszenia metodyki zapewniania jakości wykorzystującej predykcję), wykazały nieistotny – według autora – wpływ na efektywność procesu w momencie, gdy bazowy zbiór danych (pobrany z oprogramowania SourceMonitor) został poszerzony o metryki zapachów kodu pozyskane z użyciem CodeAnalysis. Nawet jeśli wynik predykcji dla zbioru zawierającego zapachy kodu okazał się nieznacznie lepszy w pierwszej części eksperymentu (gdy używano zbioru metryk opisujących cały kod źródłowy), to i tak pozostaje on w marginesie błędu. Aczkolwiek, w drugiej części eksperymentu, gdzie badano mniejsze zbiory danych, najlepsze wyniki predykcji osiągnięto dla zbioru danych bez metryk zapachów kodu, dodanie tychże metryk do zbioru pogorszyło wyniki, a przy wykorzystaniu samych metryk zapachów kodu uzyskano najgorsze wyniki, z predykcją gorszą od zgadywania losowego ($MCC = -0,0313$) włącznie.

Można zatem powiedzieć, że w oprogramowaniu wytwarzanym komercyjnie, takim jak ROPTIS, wpływ użycia metryk zapachu kodu na predykcję defektów jest bardzo mały, a użycie metryk bazujących na zapachach kodu, przynajmniej tych generowanych przez narzędzie

CodeAnalysis, nie powinno być brane pod uwagę ze względu na to, że dodatkowy wysiłek (mimo, że mały) potrzebny na ich przygotowanie i użycie nie jest rekompensowany – wyniki predykcji mogą finalnie być nawet gorsze.

Opisane w tym rozdziale badania pozwoliły zrealizować *dziewiąty cel szczegółowy* nakreślony w punkcie 1.2 niniejszej pracy: przeprowadzono modyfikację procesu predykcji i zbadano, w jaki sposób na wyniki predykcji defektów oprogramowania wpłynie zastosowanie mało popularnych w środowisku komercyjnym metryk opartych o tzw. zapachy kodu źródłowego.

7. Podsumowanie

Wyniki badań opisywanych w niniejszej pracy, w tym zastosowania w komercyjnym wytwarzaniu oprogramowania zaproponowanej metodyki zapewniania jakości wykorzystującej predykcję defektów, zostały zaprezentowane kierownictwu firmy – partnera badań. Poza znaczącą redukcją liczby znajdowanych defektów zarówno przed, jak i po wdrożeniu oprogramowania, entuzjastycznie przyjęto wzrost wydajności systemu ROPTIS – w najlepszym przypadku zaobserwowano skrócenie czasu realizacji jednej z operacji o ok. 72 sekundy. Aby dać lepsze wyobrażenie kierownictwu firmy o pozytywnym wpływie zaproponowanej w niniejszej rozprawie metodyki DPDQA na rozwijane oprogramowanie, uświadomiono kierownictwu, że wspomniana operacja w jednej tylko fabryce, należącej do przedsiębiorstwa, została wykonana aż 114780 razy w ciągu roku. Oznacza to, że dzięki jednemu ulepszeniu, firma może zaoszczędzić prawie 2300 godzin czasu rocznie w jednej fabryce. Wynik ów można pomnożyć, uwzględniając to, że 1) system ROPTIS jest stosowany w 174 fabrykach i że 2) brana jest pod uwagę tylko ta jedna zaobserwowana poprawa wydajności. Biorąc pod uwagę całą poprawę jakości i wydajności uzyskane we wszystkich fabrykach dzięki zastosowaniu metodyki DPDQA, należy spodziewać się realnych oszczędności dla całego przedsiębiorstwa. Również z punktu widzenia kosztów, niniejsze badania pokazują, że pozyskanie, wdrożenie i użycie przyjaznych dla użytkownika narzędzi służących predykcji defektów oprogramowania, takich jak oparte na licencji GNU DePress, nie wpływa na budżet projektu rozwoju oprogramowania, a nawet – jak opisano w niniejszej rozprawie – można liczyć na bardzo duży zwrot z inwestycji, w skrajnym przypadku nawet 7300%.

W konsekwencji, kierownictwo międzynarodowej firmy będącej partnerem – i niejako gospodarzem – niniejszych badań, zachęczone przedstawionymi wynikami stosowania predykcji defektów i możliwymi oszczędnościami, które można osiągnąć, ustanowiło w organizacji DPDQA jako jedną z oficjalnych metodyk, która może być stosowana globalnie w projektach rozwoju oprogramowania. Zostały zapewnione odpowiednie fundusze i został powołany specjalny zespół, który ma wspierać każdą grupę projektową chcącą wykorzystywać predykcję defektów w procesie rozwoju oprogramowania.

Programiści nie pozostali jedynymi odbiorcami przewidywania kodu podatnego na defekty. Zespoły testerów oprogramowania również zaczęły wykorzystywać wyniki predykcji, aby skupić dodatkowe wysiłki testowe na funkcjonalnościach, które używają kodu źródłowego sklasyfikowanego jako podatny na defekty. Takie podejście powinno skutkować lepszym

testowaniem, lepszą alokacją zasobów testowych i ogólną dalszą poprawą procesu zapewniania jakości. Dlatego jednym z tematów, na którym skupia się dalsza praca autora niniejszej rozprawy, jest opłacalne rozwiązanie problemu: jak rozpoznać i wskazać wszystkie przypadki testowe, które są związane z obszarami kodu źródłowego sklasyfikowanymi jako defektogenne. Jednym z możliwych rozwiązań jest dostarczenie specjalnego narzędzia, które połączy wyniki predykcji defektów oprogramowania z przypadkami testowymi. Ustalona konwencja nazewnictwa w kodzie źródłowym, dostarczająca informacji o realizowanych przez kod funkcjonalnościach może pomóc w dostarczeniu takiego rozwiązania.

Z obserwacji autora podczas kilkuletniej pracy z zastosowaniem predykcji defektów w komercyjnym wytwarzaniu oprogramowania wynika, że programiści są podekscytowani tą nowatorską techniką wykorzystującą uczenie maszynowe. Zatem kolejnym etapem przyszłej pracy badawczej będzie sprawdzenie, czy możliwe jest **efektywne kosztowo** dalsze rozwijanie metodyki wykorzystującej predykcję defektów i wskazanie kodu podatnego na defekty przed fazą testów systemowych – właśnie wtedy, gdy kod jest pisany przez programistów.

Niestety, mimo ekscytacji nowatorskimi technologiami – w tym ową, „legendarną” można by rzec, sztuczną inteligencją – można było także zaobserwować, że zespoły projektowe miały obawy przed użyciem nowych narzędzi, włączając w to nawet te łatwe w nauce i obsłudze, jak DePress. W związku z tym, jednym z przyszłych zadań autora będzie zautomatyzowanie od początku do końca procesu predykcji defektów – od pozyskiwania danych, poprzez budowę modeli predykcji, aż do automatycznego wskazywania elementów podatnych na defekty w narzędziach do programowania i testowania.

Wraca też uwagę, jak długi horyzont czasowy przedstawionych w niniejszej rozprawie badań, wpłynął na ich metody: możemy obserwować, jak z czasem pojawiają się nowe funkcjonalności w stosowanym środowisku narzędziowym (zmiana algorytmów na bardziej wydajne, np. zamiana drzewa decyzyjnego na las losowy, losowego próbkowania klasy większościowej na algorytm SMOTE itp.) czy też nawet nowości w samym obszarze nauki, jakim jest predykcja defektów oprogramowania (np. zastąpienie miary F przez współczynnik korelacji Matthews'a). Z przymrużeniem oka można podsumować, że niniejsze badania po raz kolejny potwierdziły to, co Heraklit z Efezu stwierdził już pięć wieków przed naszą erą.

Jak Czytelnik bez wątpienia zauważył, w tekście pojawiały się cytaty ze znamienitych ludzi nauki. Takż i autor chciałby pozwolić sobie zakończyć niniejszą rozprawę cytatem z kolejnego „człowieka nauki”, mianowicie profesora stworzonej przez siebie katedry mniemanologii stosowanej [197] i wielokrotnego gościa Trzeciego Programu Polskiego Radia, Jana Tadeusza Stanisławskiego:

„I to by było na tyle.”

Dodatki

A. Charakterystyki badanych projektów

Zostały tutaj przedstawione cechy projektów rozwoju oprogramowania wytypowanych przez firmę-partnera badań nad wdrożeniem predykcji defektów (bez projektu ROPTIS, który szerzej został opisany w punkcie 5.2.1). Informacje zawarte w poniższych tabelach zostały dostarczone przez zespoły projektowe – więcej na ten temat czytelnik znajdzie w punkcie 3.1. Zachowano oryginalną pisownię odpowiedzi.

Tabela A.1: Charakterystyka projektu V

Wykorzystywane narzędzie do śledzenia defektów	IBM ClearQuest, JIRA
Od kiedy defekty są rejestrowane?	od 2005
Zakładana data ukończenia projektu	Projekt w ciągłej fazie utrzymania (pielęgnacji)
Czy wykonywane są testy nowych funkcjonalności?	Nie
Czy wykonywane są testy regresji?	Tak
Pokrycie kodu źródłowego przez testy jednostkowe	10% LOC
Czy został sporządzony plan testów i przypadki testowe?	Tak
Rozmiar zespołu	8 programistów i 3 testerów

Tabela A.2: Charakterystyka projektu P

Wykorzystywane narzędzie do śledzenia defektów	IBM ClearQuest, JIRA, HP Quality Center
Od kiedy defekty są rejestrowane?	od 1 września 2008 roku
Zakładana data ukończenia projektu	Brak
Czy wykonywane są testy nowych funkcjonalności?	Tak
Czy wykonywane są testy regresji?	Tak
Pokrycie kodu źródłowego przez testy jednostkowe	5%-74% LOC (w zależności od modułu)
Czy został sporządzony plan testów i przypadki testowe?	Tak
Rozmiar zespołu	7 programistów i 7 testerów

Tabela A.3: Charakterystyka projektu T (Xates)

Wykorzystywane narzędzie do śledzenia defektów	JIRA
Od kiedy defekty są rejestrowane?	od 2009
Zakładana data ukończenia projektu	Brak
Czy wykonywane są testy nowych funkcjonalności?	Tak
Czy wykonywane są testy regresji?	Nie
Pokrycie kodu źródłowego przez testy jednostkowe	3% LOC
Czy został sporządzony plan testów i przypadki testowe?	Tak
Rozmiar zespołu	3 programistów i 1 tester

Tabela A.4: Charakterystyka projektu B

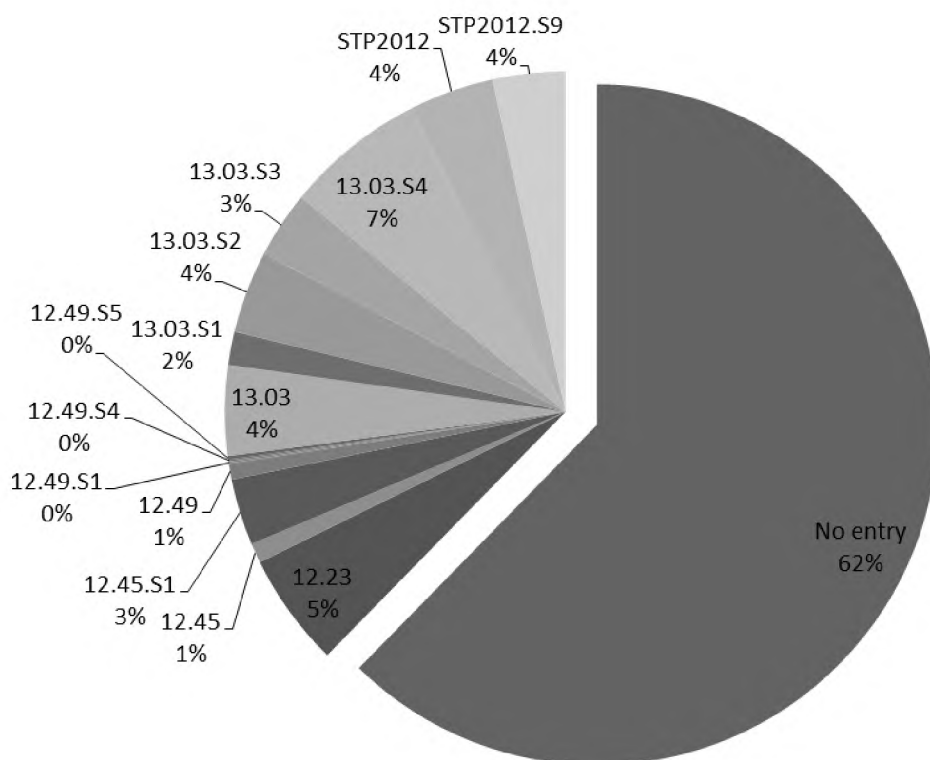
Wykorzystywane narzędzie do śledzenia defektów	JIRA
Od kiedy defekty są rejestrowane?	od 18.11.2011
Zakładana data ukończenia projektu	Brak
Czy wykonywane są testy nowych funkcjonalności?	Tak
Czy wykonywane są testy regresji?	Nie
Pokrycie kodu źródłowego przez testy jednostkowe	17% LOC
Czy został sporządzony plan testów i przypadki testowe?	Tak
Rozmiar zespołu	8 programistów i 2 testerów

Tabela A.5: Charakterystyka projektu M

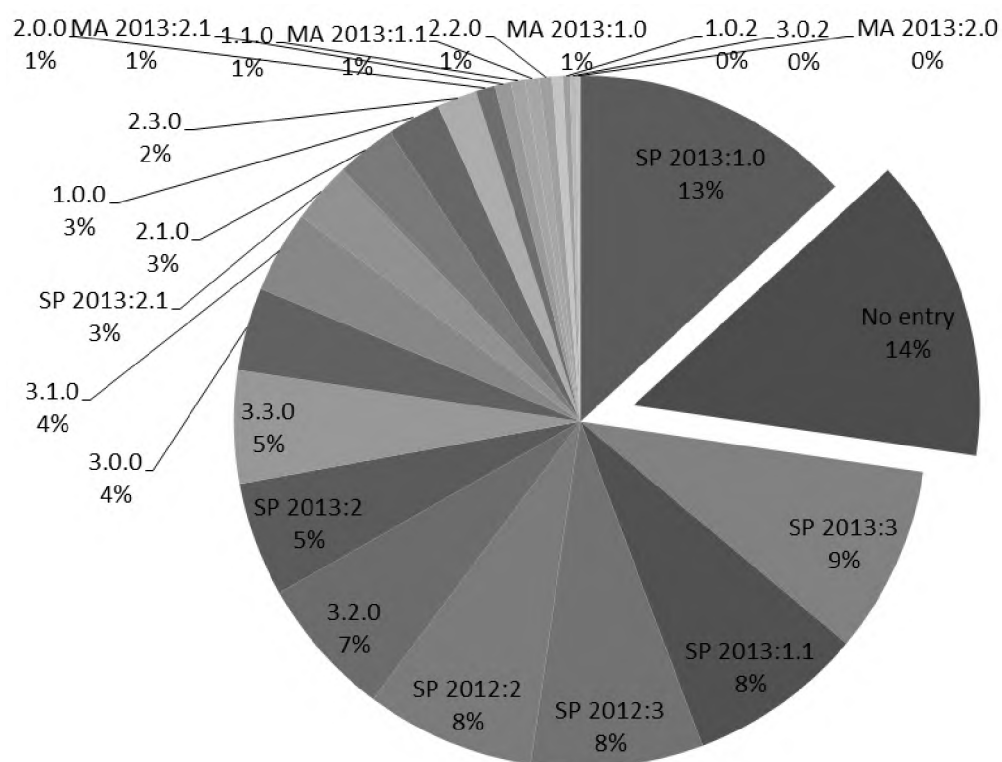
Wykorzystywane narzędzie do śledzenia defektów	Microsoft Team Foundation Server, maile
Od kiedy BR-y są rejestrowane?	od kwietnia 2014
Zakładana data ukończenia projektu	Początkowo: 2016, finalnie: 2021
Czy wykonywane są testy nowych funkcjonalności?	Nie
Czy wykonywane są testy regresji?	Nie
Pokrycie kodu źródłowego przez testy jednostkowe	Brak testów jednostkowych
Czy został sporządzony plan testów i przypadki testowe?	Nie
Rozmiar zespołu	2 programistów

B. Wyniki analizy numeracji wersji oprogramowania widocznej w narzędziach śledzenia defektów

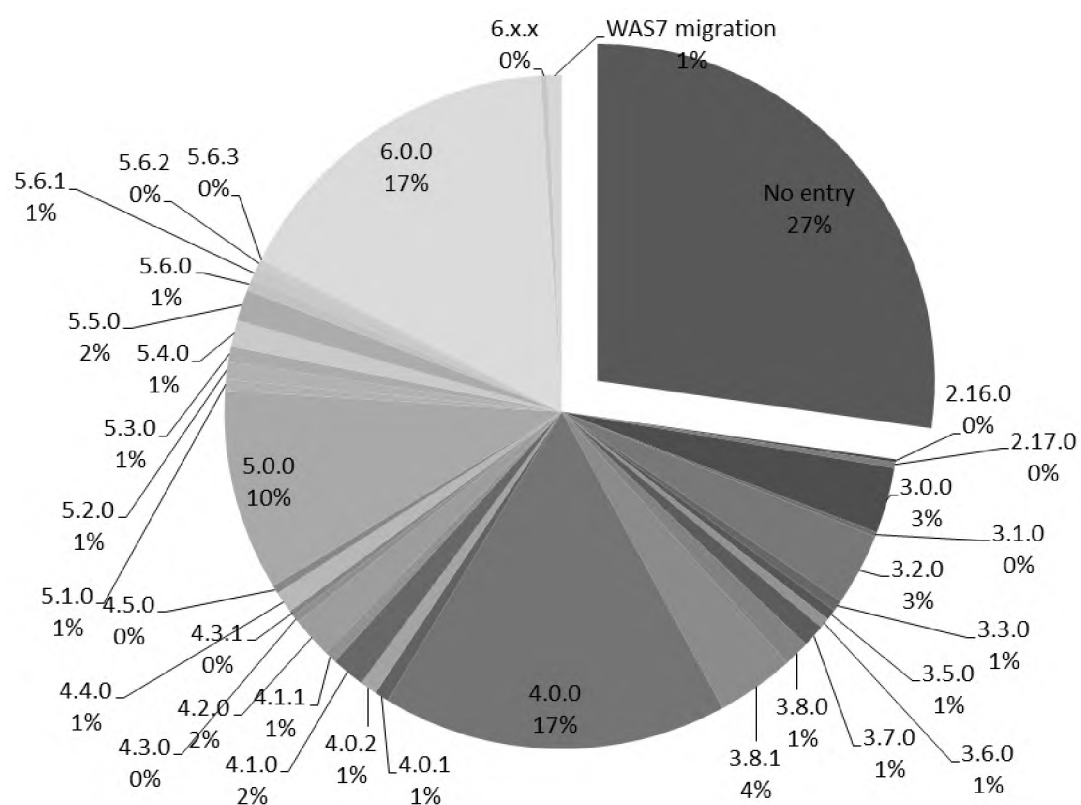
W niniejszym dodatku przedstawiono najczęściej pojawiające się numery wersji oprogramowania w raportach defektów badanych projektów (poza projektem „M” – gdzie występował tylko jeden numer wersji: 1.0.0.0 oraz projektem „ROPTIS”, którego danych o wersjach nie pozwolono opublikować). Numery dotyczą wersji oprogramowania, w których znaleziono defekt. Zwraca uwagę duży odsetek zareportowanych defektów bez informacji o wersji („No entry”) oraz niespójna numeracja wersji w ramach tego samego projektu. Więcej na ten temat można znaleźć w punkcie 3.2.1.



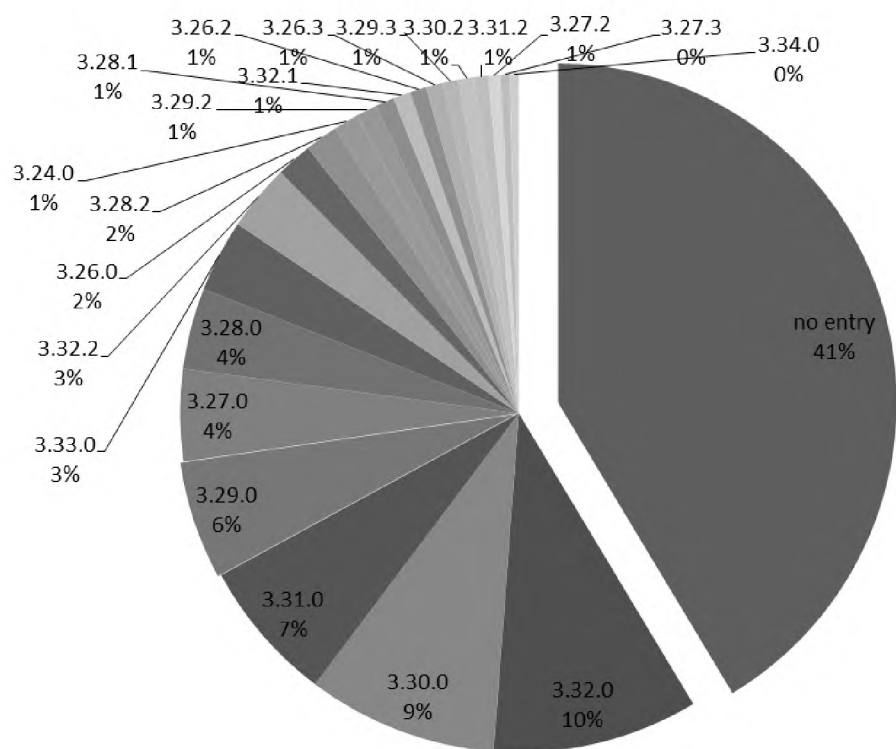
Rysunek B.1: Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie B



Rysunek B.2: Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie P



Rysunek B.3: Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie T (Xates)



Rysunek B.4: Odsetek defektów zarejestrowanych dla określonych numerów wersji oprogramowania w projekcie V

Bibliografia

- [1] Agena. AgenaRisk WWW, 2022. Dostęp: 2022.07.03.
- [2] Fumio Akiyama. An Example of Software System Debugging. In *Proceedings of the International Federation of Information Processing Societes Congress*, pages 353–359, 1971.
- [3] Tarek Alkhaeir and Bartosz Walter. The effect of code smells on the relationship between design patterns and defects. *IEEE Access*, 9:3360–3373, 2020.
- [4] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.
- [5] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.
- [6] Harald Altinger, Steffen Herbold, Jens Grabowski, and Franz Wotawa. Novel Insights on Cross Project Fault Prediction Applied to Automotive Software. *Testing Software and Systems*, 9447:141–157, 2015.
- [7] Erik Arisholm, Lionel C Briand, and Magnus Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 215–224. IEEE, 2007.
- [8] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. *The Journal of Systems and Software*, 83(1):2–17, 2010.
- [9] Atlassian. JIRA Homepage, 2022. Dostęp: 2022.07.02.
- [10] Alberto Bacchelli, Marco D’Ambros, and Michele Lanza. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010.
- [11] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus A. F. Andersen, and Henrik Nielsen. Assessing the Accuracy of Prediction Algorithms for Classification : An Overview. *Bioinformatics*, 16(5):412–424, 2000.
- [12] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.
- [13] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [14] Kent Beck. Code Smell, 2022. Dostęp: 2022.07.02.

- [15] Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O. Coplien, Lutz Dominick, and Frances Paulisch. Industrial experience with design patterns. In *ICSE '96 Proceedings of the 18th international conference on Software engineering*. IEEE, 1996.
- [16] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Looking for bugs in all the right places. In *ISSTA '06 Proceedings of the 2006 international symposium on Software testing and analysis*, pages 61–72, 2006.
- [17] Markland J. Benson. Toward Intelligent Software Defect Detection – Learning Software Defects by Example. In *Proceedings of the 2011 IEEE 34th Software Engineering Workshop*. IEEE, 2011.
- [18] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.
- [19] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don’t touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.
- [20] Barry W. Boehm. Software Engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, 1976.
- [21] Guillaume Boissier and Keith Cassell. Eclipse Metrics 2 Homepage, 2022. Dostęp: 2022.07.02.
- [22] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [23] Jason Brownlee. *Clever Algorithms. Nature-Inspired Programming Recipes*. Jason Brownlee, 2011.
- [24] Alexander Bubeck, Florian Weisshardt, Tobias Sing, Ulrich Reiser, Martin Hagele, and Alexander Wilhelm Verl. Implementing best practices for systems integration and distributed software development in service robotics - the Care-O-bot@robot family. In *System Integration (SII), 2012 IEEE/SICE International Symposium on*. IEEE, 2012.
- [25] Cagatay Catal and Banu Diri. A Systematic Review of Software Fault Prediction Studies. *Expert Systems with Application*, 36(4):7346–7354, 2009.
- [26] Chaminda Chandrasekara and Pushpa Herath. Branching with azure git repos. In *Hands-on Azure Repos*, pages 171–206. Springer, 2020.
- [27] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence*, pages 321–357, 2002.
- [28] Huo Yan Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.
- [29] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):1–13, 2020.

- [30] Davide Chicco, Niklas Tötsch, and Giuseppe Jurman. The matthews correlation coefficient (mcc) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData mining*, 14(1):1–22, 2021.
- [31] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.
- [32] Samuel Daniel Conte, Hubert E Dunsmore, and Vincent Y Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., 1986.
- [33] Thomas D Cook and Donald Thomas Campbell. *Quasi-experimentation: Design and analysis for field settings*. Rand McNally, 1979.
- [34] Marco D’Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.
- [35] Kathleen Coleman Dangle, Patricia Larsen, Michele Shaw, and Marvin V. Zelkowitz. Software Process Improvement in Small Organizations: A Case Study. *IEEE Software*, 22(6):68–75, 2005.
- [36] Philip K. Dick. *The Collected Short Stories of Phillip K. Dick, volume 4: Minority Report*. Underwood–Miller Inc., 1987.
- [37] Karthik Dinakar. Agile development: overcoming a short-term focus in implementing best practices. In *OOPSLA ’09 Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009.
- [38] Marco D’Ambros and Michele Lanza. Distributed and Collaborative Software Evolution Analysis with Churrasco. *Science of Computer Programming*, 75(4):276–287, 2010.
- [39] Marco D’Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.
- [40] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.
- [41] Trygve Eftestøl. Controlling true positive rate in roc analysis. In *2009 36th Annual Computers in Cardiology Conference (CinC)*, pages 353–356. IEEE, 2009.
- [42] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering*. Addison-Wesley, 2003.
- [43] Emelie Engström, Per Runeson, and Greger Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 75–78. IEEE, 2010.
- [44] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [45] Norman Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.
- [46] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Łukasz Radliński, and Paul Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499, 2008.

- [47] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do We Need Hundreds of Classifiers to Solve Real World Classification Problems? *J. Mach. Learn. Res.*, 15(1):3133–3181, 2014.
- [48] Brian Fitzgerald, Gerard Hartnett, and Kieran Conboy. Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*, 15(2):200–213, 2006.
- [49] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [50] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2006.
- [51] Ilenia Fronza, Arto Hellas, Petri Ihantola, and Tommi Mikkonen. Code reviews, software inspections, and code walkthroughs: Systematic mapping study of research topics. In *International Conference on Software Quality*, pages 121–133. Springer, 2020.
- [52] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.
- [53] G. Ann Campbell. Cognitive Complexity, Because Testability != Understandability, 2022. Dostęp: 2022.07.04.
- [54] Tihana Galinac and Željka Car. Software verification process improvement proposal using six sigma. In Jürgen Münch and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement: 8th International Conference, PROFES 2007, Riga, Latvia, July 2-4, 2007. Proceedings*, pages 51–64, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [55] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.
- [56] Hinam Garg and Mohit Garg. Analysing The Quality Attributes of AOP Using CYVIS Tool. *International Journal of Computers & Technology*, 4(4):648–653, 2013.
- [57] David Garlan and Dewayne Perry. Software architecture: practice, potential, and pitfalls. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*. IEEE, 1994.
- [58] Diane L. Gibson, Dennis R. Goldenson, and Keith Kost. Performance Results of CMMI-Based Process Improvement, 2006. Special Technical Report, Software Engineering Institute.
- [59] Emanuel Giger, Marco D’Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.
- [60] Dennis R. Goldenson and Diane L. Gibson. Demonstrating the Impact and Benefits of CMMI: An Update and Preliminary Results, 2003. Special Technical Report, Software Engineering Institute.
- [61] Ai Gong, Yi Zhong, Weiqin Zou, Yangyang Shi, and Chunrong Fang. Incorporating android code smells into java static code metrics for security risk prediction of android applications. In 2020

- IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 30–40. IEEE, 2020.
- [62] Arnulf BA Graf and Silvio Borer. Normalization in support vector machines. In *Joint Pattern Recognition Symposium*, pages 277–282. Springer, 2001.
 - [63] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview. *arXiv preprint arXiv:2008.05756*, 2020.
 - [64] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.
 - [65] Ines Grützner, Stephan Weibelzahl, and Patrick Waterson. Improving courseware quality through life-cycle encompassing quality assurance. In *SAC '04 Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004.
 - [66] L. Guo, Y. Ma, B. Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004.
 - [67] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.
 - [68] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier, 1977.
 - [69] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
 - [70] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
 - [71] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.
 - [72] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.
 - [73] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.
 - [74] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems*, 3(3):143–158, 1996.
 - [75] Tilman Holschuh, Markus Pauser, Kim Herzig, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects in SAP Java code: An experience report. In *ICSE-Companion 2009. 31st International Conference on Software Engineering*, pages 172–181, 2009.

- [76] Jarosław Hryszko. Nowa metodyka zapewnienia jakości: Jak sztuczna inteligencja pomogła wybrać, gdzie zastosować najlepsze praktyki pisania kodu. In *Testowanie w praktyce. Studium przypadków*. Wydawnictwo Naukowe PWN, 2016.
- [77] Jarosław Hryszko and Lech Madeyski. Bottlenecks in Software Defect Prediction Implementation in Industrial Projects. *Foundations and Computing and Decision Sciences*, 40(1):17–33, 2015.
- [78] Jarosław Hryszko and Lech Madeyski. Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project. In Lech Madeyski, Michał Śmiałek, Bogumiła Hnatkowska, and Zbigniew Huzar, editors, *Software Engineering: Challenges and Solutions*, volume 504 of *Advances in Intelligent Systems and Computing*, pages 77–90. Springer, 2016.
- [79] Jarosław Hryszko, Lech Madeyski, Marta Dąbrowska, and Piotr Konopka. Defect Prediction with Bad Smells in Code. In Bogumiła Hnatkowska and Michał Śmiałek, editors, *Software Engineering: Improving Practice through Research*, pages 163–176, 2016.
- [80] Jarosław Hryszko, Lech Madeyski, and Radosław Samlik. Application of Defect Prediction-Driven Quality Assurance Methodology in Industrial Software Development Project, 2016. Pre-print.
- [81] Chin-Yu Huang and Jung-Hua Lo. Optimal Resource Allocation For Cost and Reliability of Modular Software Systems in The Testing Phase. *Journal of Systems and Software*, 79(5):653–664, 2006.
- [82] Nam Jaechang. Survey on Software Defect Prediction, 2014. HKUST PhD Qualifying Examination.
- [83] Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.
- [84] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289. IEEE, 2013.
- [85] Jim Holmes. SourceMonitor WWW, 2022. Dostęp: 2022.07.02.
- [86] Marian Jureczko and Lech Madeyski. Towards Identifying Software Project Clusters with Regard to Defect Prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10*, pages 9:1–9:10, New York, NY, USA, 2010. ACM.
- [87] Marian Jureczko and Lech Madeyski. A Review of Process Metrics in Defect Prediction Studies. *Metody Informatyki Stosowanej*, 30(5):133–145, 2011.
- [88] Marian Jureczko and Jan Magott. QualitySpy: a framework for monitoring software development processes. *Journal of Theoretical and Applied Computer Science*, 6(1):35–45, 2012.
- [89] Marian Jureczko and Diomidis Spinellis. Using Object-Oriented Design Metrics to Predict Software Defects. In *Models and Methodology of System Dependability. Fifth International Conference on Dependability of Computer Systems DepCoS*, page 69–81, 2008.
- [90] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

- [91] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.
- [92] Itir Karac and Burak Turhan. What do we (really) know about test-driven development? *IEEE Software*, 35(4):81–85, 2018.
- [93] Taghi M. Khoshgoftaar, Edward B. Allen, John P. Hudepohl, and Stephen J. Aud. Application of Neural Networks To Software Quality Modelling Of a Very Large Telecommunications System. *IEEE Transactions on Neural Networks*, 8(4):902–909, 1997.
- [94] Taghi M. Khoshgoftaar, Abhijit S. Pandya, and David L. Lanning. Application of Neural Networks for Predicting Faults. *Annals of Software Engineering*, 1(1):141–154, 1995.
- [95] Taghi M. Khoshgoftaar and Naeem Seliya. Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. *Empirical Software Engineering*, 9(3):229–257, 2004.
- [96] Taghi M. Khoshgoftaar and Naeem Seliya. Assessment of a New Three-Group Software Quality Classification Technique: An Empirical Case Study. *Empirical Software Engineering*, 10(2):183–218, 2005.
- [97] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.
- [98] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
- [99] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *33rd International Conference on Software Engineering (ICSE)*, pages 481–490, 2011.
- [100] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.
- [101] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.
- [102] Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(13):671–680, 1983.
- [103] Michael Kläs, Frank Elberzhager, Jürgen Münch, Klaus Hartjes, and Olaf von Graevemeyer. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 119–128. ACM, 2010.
- [104] Michael Kläs, Haruka Nakao, Frank Elberzhager, and Jürgen Münch. Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data-A Case Study. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 17–26, 2008.

- [105] KNIME.COM AG. KNIME Framework Documentation, 2022. Dostęp: 2022.07.02.
- [106] Young Hoon Kwak and Frank T. Anbari. Benefits, obstacles, and future of six sigma approach. *Technovation*, 26(5-6):708–715, 2004.
- [107] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.
- [108] Michele Lanza, Andrea Mocci, and Luca Ponzanelli. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software*, 33(6):102–105, Nov 2016.
- [109] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro Interaction Metrics for Defect Prediction? In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 311–321, 2011.
- [110] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking Classification Models for Software Defect Prediction. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.
- [111] Chris Lewis, Zhonpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does Bug Prediction Support Human Developers? Findings From a Google Case Study. In *ICSE '13 Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381, 2013.
- [112] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.
- [113] Paul Luo Li, James Herbsleb, Mary Shaw, and Brian Robinson. Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc. In *Proceedings of the 28th International Conference on Software Engineering*, pages 413–422, 2006.
- [114] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing Software Metrics Tools. In *Proceedings of The 2008 International Symposium on Software Testing and Analysis*, pages 131–142, 2008.
- [115] Barbara Liskov. Keynote address-data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, 1987.
- [116] Huihua Lu and Bojan Cukic. An adaptive approach with active learning in software fault prediction. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, pages 79–88. ACM, 2012.
- [117] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.
- [118] Lech Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, (Heidelberg, London, New York), 2010.

- [119] Lech Madeyski. The Impact of Test-First Programming on Branch Coverage and Mutation Score Indicator of Unit Tests: An Experiment. *Information and Software Technology*, 52(2):169–184, 2010.
- [120] Lech Madeyski and Marian Jureczko. Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. *Software Quality Journal*, 23(3):393–422, 2015.
- [121] Lech Madeyski and Marek Majchrzak. Software Measurement and Defect Prediction with DePress Extensible Framework. *Foundations and Computing and Decision Sciences*, 39(4):249–270, 2014.
- [122] Lech Madeyski and Marek Majchrzak. Software Measurement and Defect Prediction with Depress Extensible Framework. *Foundations of Computing and Decision Sciences*, page 249–270, 2014.
- [123] Lech Madeyski and Marek Majchrzak. ImpressiveCode DePress (Defect Prediction for software systems) Extensible Framework, 2016.
- [124] Lech Madeyski and Łukasz Szała. The impact of test-driven development on software development productivity — an empirical study. In Pekka Abrahamsson, Nathan Baddoo, Tiziana Margaria, and Richard Messnarz, editors, *Software Process Improvement*, volume 4764 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin Heidelberg, 2007.
- [125] Aygul Malikova and Giancarlo Succi. Modern code reviews: Preliminary results of an analysis of the state of the art with respect to the role played by human factors. *ICSOFT*, pages 253–260, 2021.
- [126] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.
- [127] Brian Marick. Classic Testing Mistakes. In *Software Quality Approaches: Testing, Verification, and Validation: Software Best Practice I*. Springer, 2001.
- [128] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.
- [129] Robert Cecil Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- [130] Brian W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Bioinformatics*, 405(2):442–451, 1975.
- [131] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 4(2):308–320, December 1976.
- [132] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.
- [133] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126 – 139, 2004.
- [134] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

- [135] Tim Menzies, Omid Jalali, Jairus Hihn, Dan Baker, and Karen Lum. Stable Rankings for Different Effort Models. *Automated Software Engineering*, 17(4):409–437, 2010.
- [136] Microsoft. Code Analysis for Managed Code Overview, 2016. Dostęp: 2022.07.02.
- [137] Microsoft. Code metrics - Maintainability index range and meaning, 2022. Dostęp: 2022.07.04.
- [138] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130, 2000.
- [139] Ritesh Modi. Infrastructure as code. In *Deep-Dive Terraform on Azure*, pages 1–8. Springer, 2021.
- [140] Myint Myint Moe. Comparative study of test-driven development tdd, behavior-driven development bdd and acceptance test–driven development atdd. *International Journal of Trend in Scientific Research and Development*, 3:231–234, 2019.
- [141] Jefferson Seide Molléri, Fabiane Barreto, and Vavassori Benitti. SESRA: A Web-based Automated Tool to Support the Systematic Literature Review Process. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 24:1–24:6, New York, NY, USA, 2015. ACM.
- [142] Akito Monden, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, and Kenichi Matsumoto. Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing. *IEEE Transactions on Software Engineering*, 39(10):1345–1357, 2013.
- [143] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A Comparative Analysis of The Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 181–190, 2008.
- [144] Matthias M. Müller and Frank Padberg. About the Return on Investment of Test-Driven Development. In *International Workshop on Economics-Driven Software Engineering Research EDSE-5*, pages 26–31, 2003.
- [145] John C. Munson and Taghi M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.
- [146] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.
- [147] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.
- [148] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006.
- [149] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318. IEEE, 2010.
- [150] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

- [151] Junji Noguchi. The legacy of w. edwards deming. *Quality Progress*, 28:35–38, 1995.
- [152] University of Maryland. Strona domowa narzędzia FindBugs, 2022. Dostęp: 2022.07.03.
- [153] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.
- [154] Oracle Corporation. Java Software Homepage, 2022. Dostęp: 2022.07.02.
- [155] Thomas J. Ostrand and Elaine J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *SIGSOFT Software Engineering Notes*, 27:55–64, 2002.
- [156] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
- [157] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
- [158] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Programmer-Based Fault Prediction. In *Proceedings of the Sixth International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.
- [159] Ganesh J Pai and Joanne Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on software Engineering*, 33(10), 2007.
- [160] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.
- [161] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019.
- [162] Parag C. Pendharkar. Exhaustive and Heuristic Search Approaches For Learning a Software Defect Prediction Model. *Engineering Applications of Artificial Intelligence*, 23:34–40, 2010.
- [163] Fayola Peters and Tim Menzies. Privacy and utility for defect prediction: Experiments with morph. In *Proceedings of the 34th International Conference on Software Engineering*, pages 189–199. IEEE Press, 2012.
- [164] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.
- [165] Paweł Piotrowski and Lech Madeyski. Software defect prediction using bad code smells: A systematic literature review. *Data-Centric Business and Applications*, pages 77–99, 2020.
- [166] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE, 2021.
- [167] David MW Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.
- [168] Roger Pressman. *Software Engineering: A Practitioner’s Approach*. McGraw-Hill, 2010.
- [169] Cliff Purington, Chris Butler, and Sarah Fister Gale. *Built to Learn: The Inside Story of How Rockwell Collins Became a True Learning Organization*. Redux Books, 2003.

- [170] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.
- [171] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.
- [172] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.
- [173] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 322–331. ACM, 2011.
- [174] Foyzur Rahman, Khatri Sammer, Earl T. Barr, and Premkumar Devanbu. Comparing Static Bug Finders and Statistical Prediction. In *Software Engineering, 2014. ICSE '14. ACM/IEEE International Conference on*. ACM, 2014.
- [175] Åke Lilienberg. Enterprise Architecture from a VG perspective, 2012. Dostęp: 2021.11.04.
- [176] Åsa Lindström. On the Syntax and Semantics of Architectural Principles. In *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference on*. IEEE, 2006.
- [177] Rich Data. SMOTE explained for noobs – Synthetic Minority Over-sampling TEchnique line by line, 2022. Dostęp: 2022.07.04.
- [178] Dick Riehle. Lessons Learned from Using Design Patterns in Industry Projects. In *Transactions on Pattern Languages of Programming II*. Springer, 2011.
- [179] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the Apache server. In *ICSE '08 Proceedings of the 30th international conference on Software engineering*. ACM, 2008.
- [180] Muhammad Rizwan and Muzaffar Iqbal. Application of 80/20 Rule in Software Engineering Waterfall Model. In *Proceedings of the International Conference on Information and Communication Technologies '09*, 2009.
- [181] Frank Sauer. Eclipse Metrics Homepage, 2022. Dostęp: 2022.07.02.
- [182] Richard W. Selby and Adam Porter. Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1756, 1988.
- [183] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.
- [184] Tushar Sharma and Diomidis Spinellis. Do we need improved code quality metrics? *arXiv preprint arXiv:2012.12324*, 2020.
- [185] Vincent Yun Shen, Tze-Jie Yu, Stephen M. Thebaut, and Lorri R. Paulsen. Identifying error-prone software—an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4):317–324, 1985.

- [186] Martin Shepperd, David Bowes, and Tracy Hall. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions in Software Engineering*, 40(6):603–616, 2014.
- [187] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of systems and software*, 26(3):197–210, 1994.
- [188] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310. ACM, 2011.
- [189] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.
- [190] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.
- [191] Forrest Shull and Carolyn Seaman. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software*, 25(1):88–90, 2008.
- [192] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating The Cost of Software Quality. *Communications of the ACM*, 41(8):67–73, 1998.
- [193] Jacek Śliwowski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *Proceedings of The 2005 International Workshop on Mining Software Repositories*, 2005.
- [194] Behjat Soltanifar, Shirin Akbarinasaji, Bora Caglayan, Ayse Basar Bener, Asli Filiz, and Bryan M Kramer. Software analytics in practice: a defect prediction model using code smells. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 148–155, 2016.
- [195] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.
- [196] wynagrodzenia.pl – wszystko o wynagrodzeniach i wynagradzaniu, 2015. Dostęp: 2015.02.28.
- [197] Jan T. Stanisławski. *Mniemanologia stosowana: krótki kurs z wypisami*. Wydawnictwo Zebra, 1991.
- [198] Giancarlo Succi, Witold Pedrycz, Milorad Stefanovic, and James Miller. Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics. *Journal of Systems and Software*, 65(1):1–12, 2003.
- [199] Seyyed Ehsan Salamaty Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 270–279. IEEE, 2013.
- [200] Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC medical imaging*, 15(1):1–28, 2015.
- [201] The Apache Foundation. Apache Maven Homepage, 2022. Dostęp: 2022.07.02.
- [202] The Apache Foundation. Apache Subversion Homepage, 2022. Dostęp: 2022.07.02.

- [203] The Eclipse Foundation. Strona domowa Eclipse IDE, 2022. Dostęp: 2022.07.02.
- [204] Ayse Tosun, Ayse Bener, Burak Turhan, and Tim Menzies. Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry. *Information and Software Technology*, 52(11):1242–1257, 2010.
- [205] Ayse Tosun, Burak Turhan, and Ayse Bener. Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry. In *Proceedings of the Fifth International Conference on Predictor Models in Software Engineering*, page 11, 2009.
- [206] Burak Turhan, Gozde Kocak, and Ayse Bener. Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry. *Expert Systems with Applications*, 36(6):9986–9990, 2009.
- [207] Burak Turhan, Tim Menzies, Ayse Bener, and Justin Di Stefano. On the Relative Value of Cross-Company and within-Company Data for Defect Prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.
- [208] Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101–1118, 2013.
- [209] VG IT. Roptis system documentation, 2019.
- [210] Huanjing Wang, Taghi M. Khoshgoftaar, and Naeem Seliya. How Many Software Metrics Should be Selected for Defect Prediction? In *24th Florida Artificial Intelligence Research Society Conference*, pages 69–74, 2011.
- [211] Shuai Wang, Bo Kang, Jinlu Ma, Xianjun Zeng, Mingming Xiao, Jia Guo, Mengjiao Cai, Jingyi Yang, Yaodong Li, Xiangfei Meng, et al. A deep learning algorithm using ct images to screen for corona virus disease (covid-19). *European radiology*, 31(8):6096–6104, 2021.
- [212] Shinya Watanabe, Haruhiko Kaiya, and Kenji Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 19–24. ACM, 2008.
- [213] Sholom M. Weiss, Robert S. Galen, and Prasad V. Tadepalli. Optimizing The Predictive Value of Diagnostic Decision Rules. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 512–526, 1987.
- [214] Elaine J. Weyuker, Robert M. Bell, and Thomas J. Ostrand. We’re Finding Most of the Bugs, but What are We Missing? In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 313–322, 2010.
- [215] F. George Wilkie, Donald McFall, and Fergal McCaffery. An evaluation of CMMI process areas for small- to medium-sized software development organisations. *Software Process: Improvement and Practice*, 10(2):189–201, 2005.
- [216] Ian H. Witten, Elbe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.
- [217] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

- [218] W. Eric Wong, Joseph Horgan, Michael Syring, Wayne Zage, and Dolores Zage. Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System. *Software: Practice and Experience*, 30(14):1587–1608, 2000.
- [219] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.
- [220] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
- [221] Cheng Zhang and David Budgen. What Do We Know about the Effectiveness of Software Design Patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2011.
- [222] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM, 2014.
- [223] Hongyu Zhang and Rongxin Wu. Sampling program quality. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
- [224] Min Zhang, Tracy Hall, Nathan Baddoo, and Paul Wernick. Do bad smells indicate "trouble" in code? In *DEFACTS '08 Proceedings of the 2008 workshop on Defects in large software systems*, pages 43–44. ACM, 2008.
- [225] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 531–540. IEEE, 2008.
- [226] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 435–438. IEEE, 2009.
- [227] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.