AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Bartłomiej Bodziechowski

kierunek studiów: **fizyka techniczna**

specjalność: **fizyka komputerowa**

Analiza jakości kodu przy użyciu metryk oprogramowania na wybranych przykładach

Opiekun: **dr inż. Marian Bubak**

Konsultant: **mgr inż. Eryk Ciepiela**

Kraków, wrzesień 2012

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Kraków, wrzesień 2012 r.

Tematyka pracy magisterskiej i praktyki dyplomowej Bartłomieja Bodziechowskiego, studenta V roku studiów kierunku fizyka techniczna, specjalności fizyka komputerowa

Temat pracy magisterskiej: **Analiza jakości kodu przy użyciu metryk oprogramowania na wybranych przykładach**

Opiekun pracy: dr inż. Marian Bubak
Konsultant pracy: mgr inż. Eryk Ciepiela
Recenzenci pracy:
Miejsce praktyki dyplomowej: Akademickie Centrum Komputerowe CYFRONET AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - zapoznanie się z ideą metryk w statycznej analizie oprogramowania, wyszukanie metryk i literatury związanej z ich weryfikacją,
   - zapoznanie się z narzędziami wspierającymi metryki statycznej analizy oprogramowania,
   - dokonanie pomiarów metryk dla części sytemu GridSpace2,
   - dyskusja i analiza wyników,
4. Zebranie i opracowanie wyników metryk, wyciągnięcie wniosków i zatwierdzenie przez promotora.
5. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie: ……..

....

(podpis kierownika katedry) (podpis opiekuna)

Recenzja 1

Recenzja 2

Streszczenie

Testowanie oraz analiza statyczna oprogramowania to dwa główne filary zapewnienia jakości oprogramowania. Podczas gdy to pierwsze narzędzie jest powszechnie stosowane, drugie nie spotyka się już z taką aprobatą ze strony programistów. Analiza statyczna pozwala ocenić oprogramowanie na wczesnym etapie jego tworzenia, ponieważ bada się albo kod źródłowy albo pliki binarne tworzone przez kompilatory. Wychwycenie różnego rodzaju błędów na tak wczesnym etapie pozwala na redukcję kosztów napraw do minimum. Podstawowym mechanizmem wykorzystywanym w analizie statycznej są metryki, czyli funkcje odwzorowujące jednostki oprogramowania w wartości liczbowe.

Przy pomocy metryk można badać różne obszary związane z pisaniem dobrej jakości oprogramowania. Najpopularniejsze są metryki zliczeń, które sumują liczbę artefaktów na różnych poziomach abstrakcji oraz metryki powiązań, które badają liczbę powiązań między artefaktami. Ta grupa metryk pozwala na ocenę złożoności pisanego oprogramowania. Metryki pozwalają też na weryfikację stosowania różnego rodzaju paradygmatów programowania obiektowego oraz praw i zasad, które winny być przestrzegane w dobrej jakości oprogramowaniu.

W pracy tej przedstawiono przegląd metryk dla wymienionych wyżej obszarów. Omówiono szczegółowo najpopularniejsze zestawy metryk obiektowych: MOOD, zestaw CK oraz Martina. Zweryfikowano również stan aktualnych badań nad omówionymi metrykami w szczególności przywołano literaturę dotyczącą ich teoretycznej i praktycznej weryfikacji. Okazało się, że metryki obiektowe mogą być używane w celu przewidywania takich zewnętrznych cech jakości oprogramowania jak: niezawodność (korelacja wysokich wartości niektórych metryk z liczbą błędów), czy pielęgnowalność.

Kolejnym celem pracy było przeprowadzenie przeglądu narzędzi implementujących pomiary metryk dla języka Java. Za szczególnie użyteczne uznano 3 aplikacje: Stan, RefactorIT oraz Sonar.

W części praktycznej pracy badano jakość kodu systemu GridSpace2 - platformy służącej do tworzenia naukowych aplikacji obliczeniowych przy użyciu zasobów udostępnianych przez polskie i europejskie centra obliczeniowe. Wykryto, że niektóre prawa i zasady programowania obiektowego są łamane, dla wielu metryk zidentyfikowano też klasy i pakiety, które naruszają wartości zalecane metryk i w związku z tym wydano rekomendacje do ich przeglądu.

Metryki statycznej analizy oprogramowania mogą być cennym wsparciem w procesie tworzenia aplikacji. Oprócz tego, że ich wartości znajdują powiązanie z zewnętrznymi cechami jakości oprogramowania, to również pomagają one w lepszym zrozumieniu pisanego oprogramowania, a także umożliwiają kontrolę jego złożoności.

Spis treści

Spis rysunków

Spis tabel

Lista skrótów

SRP	*ang. Single responsibility principle*	Zasada jednej odpowiedzialności
OCP	*ang. Open closed principle*	Zasada otwarte zamknięte
LSP	*ang. Liskov substitution principle*	Zasada podstawienia Liskov
ISP	*ang. Interface segregation principle*	Zasada segregacji interfejsów
DIP	*ang. Dependency inversion principle*	Zasada odwrotnych zależności
REP	*ang. Reuse release principle*	Zasada równoważności użycia i wydania
CRP	*ang. Common reuse principle*	Zasada wspólnego użycia
CCP	*ang. Common close principle*	Zasada wspólnego zamknięcia
ADP	*ang. Acyclic dependency principle*	Zasada acykliczny zależności
DAG	*ang. Directed acycylic graph*	Skierowany graf acykliczny
SDP	*ang. Stable dependency principle*	Zasada stabilnych zależności
SAP	*ang. Stable abstractions principle*	Zasada stabilne abstrakcyjne
LOC	*ang. Lines of code*	Liczba linii kodu
NOF	*ang. Number of fields*	Liczba pól
NOM	*ang. Number of methods*	Liczba metod
TLC	*ang. Top level classes*	Liczba klas bez klas wewnętrznych
CC	*ang. Cyclomatic complexity*	Złożoność cyklomatyczna
NPC	*ang. Number of path complexity*	Liczba możliwych ścieżek metody
DAC	*ang. Class data abstraction coupling*	Liczba obiektów klas zawartych w danej klasie
CFOC	*ang. Class fan out complexity*	Liczba powiązań odśrodkowych klasy
ACD	*ang. Avarage component dependency*	Średnia zależność komponentu
MOOD	*ang. Metrics of object oriented design*	Metryki projektowania obiektowego
AHF	*ang. Atribute hidding factor*	Współczynnika ukrycia atrybutów
MHF	*ang. Method hidding factor*	Współczynnik ukrycia metod
AIF	*ang. Atribute inheritance factor*	Współczynnik dziedziczenia atrybutów
MIF	*ang. Method inheritance factor*	Współczynnik dziedziczenia metod
PF	*ang. Polymorphism factor*	Współczynnik polimorficzności
COF	*ang. Coupling factor*	Współczynnik powiązań
WMC	*ang. Weighted methods per class*	Ważona liczba metod w klasie
RFC	*ang. Response for a class*	Odpowiedź klasy

DIT	*ang. Depth of inheritance tree*	Głębokość drzewa dziedziczenia
NOC	*ang. Number of children*	Liczba bezpośrednich potomków klasy
CBO	*ang. Coupling between objects*	Powiązania między klasami
LCOM	*ang. Lack of cohesion of methods*	Brak spójności metod
Ce	*ang. Efferent coupling*	Powiązania odśrodkowe
Ca	*ang. Afferent coupling*	Powiązania dośrodkowe
I	*ang. Instability*	Niestabilność
A	*ang. Abstractness*	Abstrakcyjność
Dn	*ang. Normalized distance from main sequnece*	Znormalizowana odległość od ciągu głównego
VOD	*ang. Violations of Demeter*	Liczba naruszeń prawa Demeter
MFS	*ang. Minimum feedback set*	Zasada minimalnej informacji zwrotnej
CYC	*ang. Cyclic dependency*	Zależności cykliczne
DIPM	*ang. Dependency inversion principle metric*	Metryka zasada odwrotnych zależności
EP	*ang. Encapsulation principle*	Zasada hermetyczności
DC	*ang. Duplicated code*	Zduplikowany kod
LC	*ang. Line coverage*	Pokrycie linii kodu (testami)
BC	*ang. Branch coverage*	Pokrycie gałęzi (testami)
PB	*ang. Potential bugs*	Potencjalne błędy
GP	*ang. Good practise*	Dobre praktyki
R	*ang. Readability*	Czytelność

1 Wstęp

Analiza statyczna oprogramowania jest bardzo ważna w procesie tworzenia wysokiej jakości aplikacji. Najbardziej oczywistym sposobem zapewnienia jakości oprogramowania jest testowanie i zwykle jemu poświęca się najwięcej uwagi. Natomiast bardzo ważne jest, aby weryfikować napisany kod na jak najwcześniejszym etapie. Analiza statyczna oprogramowania pozwala oceniać jakość kodu już w momencie jego tworzenia. Co nie mniej ważne, umożliwia na bieżąco weryfikowanie stosowania się do reguł projektowania systemów informatycznych, a to przekłada się na zewnętrzne cechy oprogramowania takie jak pielęgnowalność, niezawodność, czy łatwość zarządzania.

Niniejsza praca zawiera studium nad metrykami statycznej analizy oprogramowania. Wydaje się, że stosowanie tych metryk nie jest powszechne wśród programistów. Jak pokazuje niniejsza praca nie ma zbyt wiele literatury dotyczącej tego tematu. Podobnie sytuacja wygląda z narzędziami wspierającymi metryki. Metryki analizy statycznej rzadko stanowią też przedmiot wykładów na uczelniach wyższych. Wśród programów polskich uczelni wyższych znaleziono jedynie jeden wykład omawiający temat metryk obiektowych: „Metryki obiektowe” w ramach kursu „Zaawansowane projektowanie obiektowe” na Politechnice Poznańskiej [21]. Nie ma też wiele opracowań, które ten temat by poruszały, a podręczniki do programowania pomijają kwestię stosowania metryk analizy statycznej.

W pracy tej przeprowadzono gruntowny przegląd metryk statycznej analizy oprogramowania ze szczególnym naciskiem na metryki oprogramowania zorientowanego obiektowo. W tym celu przeprowadzano przegląd publikacji naukowych dotyczących metryk (o ile takie powstały) – zarówno tych, gdzie metryki były proponowane, jak i również tych,

które stanowiły polemikę do pierwotnych propozycji, albo stanowiły opis badań weryfikujących w praktyce ich przydatność. Aby usystematyzować lepiej zagadnienie metryk analizy statycznej, osobno omówiono obszary jakie można badać przy ich udziale. W rozdziale opisującym metryki starano się, w oparciu o dostępną literaturę, nie tylko przedstawić teoretyczną stronę metryk, ale przede wszystkim skupiono się na ich praktycznym wykorzystaniu. Przedstawiono interpretację wartości każdej metryki, wskazano jakie zewnętrzne cechy oprogramowania mogą być oceniane przy ich udziale, a także jakie zagrożenia niesie za sobą nie stosowanie się do przestrzegania zalecanych wartości. Następnie zrobiono przegląd narzędzi implementujących metryki statycznej analizy oprogramowania dla języka Java omawiając jakie zestawy metryk można mierzyć przy ich udziale. Omówiono także inne funkcjonalności zawarte w narzędziach, które ułatwiają wykorzystanie metryk do lepszego zrozumienia kodu i ewentualnie usprawniają refektoryzację. Nabytą wiedzę zastosowano w praktyce. W tym celu dokonano audytu kodu części projektu GridSpace2 (dalej GS2) [2] rozwijanego przez zespół Distributed Computing Environments Team [3] działający w Akademickim Centrum Komputerowym CYFRONET AGH. GS2 jest platformą służącą do tworzenia naukowych aplikacji obliczeniowych tzw. eksperymentów *in silico* z użyciem zasobów udostępnianych przez polskie i europejskie centra obliczeniowe. Do głównych cech platformy można zaliczyć:

- możliwość budowania eksperymentów z wielu istniejących kodów obliczeniowych i aplikacji w ramach zadań lokalnych i gridowych,
- dostęp poprzez wygodny interfejs webowy z dowolnego komputera niezależnie od długości działania uruchamianych eksperymentów,
- dostępność rejestru skonfigurowanych aplikacji i interpreterów na różnych sajtach obliczeniowych,
- zarządzanie plikowymi rezultatami obliczeń,
- możliwość publikacji utworzonych eksperymentów wraz z towarzyszącymi danymi.

Wybrano GS2 ponieważ jest to względnie duży projekt (ponad 400 klas podzielonych na prawie 70 pakietów) rozwijany przez zespół programistów. Ponadto jest to aplikacja wielowarstwowa – posiada internetowy panel administracyjny (ang. *WebGUI*), warstwę logiki oraz API do zewnętrznych systemów.

Audyt polegał na zebraniu wartości wcześniej omówionych metryk przy pomocy wybranych narzędzi. Następnie przeprowadzono interpretację wartości każdej metryki i tam

gdzie to było konieczne wydano rekomendacje do przeglądu kodu. Na koniec w rozdziale poświęconym wnioskom omówiono przydatność stosowania metryk w inżynierii oprogramowania i najważniejsze wnioski płynące z pracy.

Cele pracy:

- zidentyfikowanie obszarów w ramach których można stosować metryki statycznej analizy oprogramowania,
- wyszukanie i pogrupowanie metryk statycznej analizy w oparciu o dostępną literaturę i przeprowadzone dotychczas badania,
- przegląd darmowych narzędzi implementujących metryki statycznej analizy oprogramowania dla języka Java,
- zastosowanie metryk analizy statycznej w praktyce dla GS2 oraz wskazanie obszarów wymagających przeglądu i ewentualnej poprawy,
- ocena przydatności metryk statycznej analizy w inżynierii jakości oprogramowania.

2 Metryki a jakość oprogramowania

Doświadczeni programiści potrafią tworzyć oprogramowanie o wysokiej jakości. Szczególnie gdy od początku znane są wymagania stawiane projektowi, jest szansa, aby zaprojektować system o przejrzystej architekturze, wolny od zbędnej złożoności. Bardzo często taka sytuacja kończy się przy okazji pierwszego wydania. Wtedy okazuje się, że system trzeba zmieniać. Może być ku temu wielu powodów: dodane są nowe funkcjonalności, odkrywane są przeoczenia, wymagania funkcjonalne są doprecyzowywane, następują zmiany wymagań sprzętowych, redefiniowane są cele klienta itp. Bardzo trudno jest napisać oprogramowanie tak, aby było przygotowane na każdego rodzaju zmiany. System zaczyna więc tracić swoją pierwotną jakość i coraz bardziej zaczyna wymykać się spod kontroli. Narastają trudności w testowaniu, utrzymaniu, czy rozszerzalności. Robert Martin porównał tę sytuację do kawałka mięsa, które zaczyna gnić i wydawać nieprzyjemne „zapachy" [6]. System nabiera następujące cechy:

- sztywność (ang. *rigidity*) – trudno jest go zmienić, ponieważ każda zmiana wymusza wiele innych zmian,
- kruchość (ang. *fragility*) – zmiany mogą spowodować, że system zacznie się łamać, często nawet w miejscach koncepcyjnie niepowiązanych z modyfikowanym obszarem,
- znieruchomienie (ang. *immobility*) – brak możliwości powtórnego użycia części systemu z powodu powiązań z innymi jego częściami,
- lepkość (ang. *viscosity*) – trudniej jest wprowadzać zmiany w poprawny sposób, tak aby zachowana była poprawna architektura,

- zmętnienie (ang. *opacity*) – ciężko się czyta kod i trudniej go zrozumieć.

Duże systemy mają tendencje do stawania się coraz bardziej skomplikowanymi. Utrzymanie ich złożoności pod kontrolą jest kluczowe dla ich pielęgnowalności i możliwości zrozumienia. Wyraźnie rysuje się więc potrzeba wprowadzenia jakiegoś systemu kontroli nad systemem w trakcie jego okresu życia. Takim narzędziem wspomagającym kontrolę jakości systemu w trakcie jego życia mogą być metryki.

Jakość nie jest prosta do zdefiniowania i może być różnie postrzegana. W odniesieniu do oprogramowania na przestrzeni lat pojawiały się różne mniej lub bardziej formalne definicje jakości. Definiowano jakość jako „zgodność z wymaganiami" albo „przydatność do użytku" [26]. Najpopularniejsze jednak ujęcie to powiązanie jakości z brakiem defektów. Oczywiście za brakiem defektów stoi jednak wiele cech oprogramowania powiązanych z jakością. Definiuje je standard ISO/IEC 9126-1. Według tego standardu dobre oprogramowanie powinny charakteryzować następujące cechy:

Funkcjonalność
- odpowiedniość
- dokładność
- współdziałanie
- zgodność
- bezpieczeństwo

Niezawodność
- dojrzałość
- tolerancja błędów
- odtwarzalność

Użyteczność
- zrozumiałość
- łatwość uczenia
- łatwość posługiwania się

Efektywność
- charakterystyka czasowa
- wykorzystanie zasobów

Pielęgnowalność
- dostępność
- podatność na zmiany
- stabilność
- łatwość walidacji

Przenośność
- dostosowywalność
- instalacyjność
- zgodność
- zamienność

Przedmiotem tej pracy jest użycie metryk statycznej analizy oprogramowania w celu zmierzania jakości oprogramowania. Według standardu IEEE 1061-1998: *metryka to funkcja odwzorowującą jednostkę oprogramowania w wartość liczbową. Ta wyliczona wartość jest interpretowalna jako stopień spełnienia pewnej własności jakości jednostki oprogramowania.*

W inżynierii jakości oprogramowania spotkać można wiele rodzajów metryk, najważniejsze to [26]:

- metryki jakości produktu np.: metryka częstości defektów, metryka problemów klienta, metryki zadowolenia klienta,
- metryki wewnątrzprocesowe: częstość defektów podczas testów maszynowych, efektywność usuwania defektów,
- metryki serwisowe oprogramowania: opóźnienie napraw i wskaźnik zarządzania opóźnieniem, czas odpowiedzi na zgłoszenie i możliwość naprawy przez serwis, odsetek źle wykonanych napraw, jakość naprawy.

Wymienione wyżej rodzaje metryk dotyczą albo analizy dynamicznej oprogramowania (gotowego produktu w trakcie testów – według standardu ISO/IEC 9126 są to metryki zewnętrzne ang. *external metrics*), albo produktu w trakcie eksploatacji – według standardu ISO są tzw. metryki w użyciu (ang. *quality in use metrics*). Metryki te pozwalają ocenić jakość gotowego wyrobu ale nie pokazują gdzie jest problem. Poza tym wykrywanie różnego rodzaju błędów na tak późnym etapie skutkuje wysokimi kosztami naprawy.

Z pewnością więc każde narzędzie, które pozwoli wykryć potencjalne problemy na wcześniejszym etapie będzie bardzo pomocne. I tutaj przydatne mogą się okazać metryki analizy statycznej oprogramowania (analizuje się kod źródłowy, albo pliki binarne tworzone przez kompilatory) , których omówienie, wykorzystanie i krytyka przydatności są przedmiotem tej pracy. Metryki analizy statycznej stanowią część metryk wewnętrznych (ang. *internal metrics*) według standardu ISO/IEC 9126. Oprócz metryk analizy statycznej do metryk wewnętrznych zalicza się też metryki obejmujące próbę „zmierzenia” wcześniejszych etapów tworzenia oprogramowania takich jak: zapytania ofertowe od klientów, definicja wymagań, specyfikacja projektowa. Te jednak nie są przedmiotem niniejszej analizy.

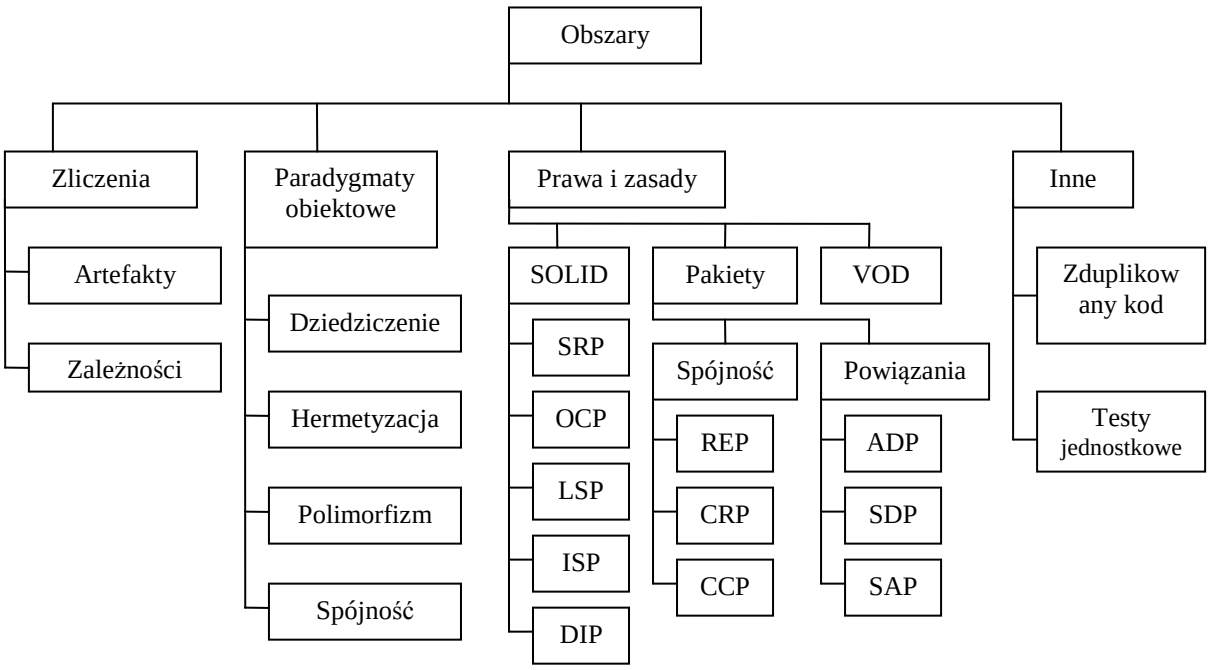
Zalecane jest aby stosować metryki wewnętrzne, które mają silne powiązanie z metrykami zewnętrznymi. Wtedy można użyć metryk wewnętrznych do przewidzenia wartości metryk zewnętrznych. Innymi słowy metryki analizy statycznej muszą mieć przełożenie na zewnętrzne cechy oprogramowania. W praktyce metryki analizy statycznej mogą być użyte do kontroli następujących cech oprogramowania: funkcjonalności, niezawodności, przenośności, efektywności, czy pielęgnowalności. Pominięta została jedynie użyteczność, która bardziej wiąże się z odpowiednim rozeznaniem wymagań klienta i definicją specyfikacji.

Abreu zaproponował następujący zestaw kryteriów, które powinny być spełniane przez metryki analizy statycznej powiązane z obiektowymi paradygmatami programowania [19]:

- sposób w jaki wyznaczane są metryki powinien być formalnie określony – wynik pomiaru tego samego systemu powinien być zawsze taki sam,
- metryki nie należące do kategorii metryk rozmiaru powinny być niezależne od rozmiaru systemu,
- metryki powinny być bezwymiarowe lub wyrażone w pewnej spójnej jednostce systemowej,
- konstrukcja metryk powinna być taka, aby można było zbierać ich wartości w jak najwcześniejszej fazie projektu,
- metryki powinny być skalowalne w dół – możliwość stosowania dla całego systemu, jak i dla wybranych modułów,
- metryki powinny być łatwo obliczalne – umożliwi to opracowanie automatycznych narzędzi,
- metryki powinny być niezależne od języka programowania.

3 Obszary mierzone przy użyciu metryk analizy statycznej oprogramowania

W rozdziale tym omówione zostały obszary kodu, które podlegają pomiarom przy użyciu metryk w statycznej analizie. Zaproponowano podział na cztery kategorie: proste zliczenia, paradygmaty obiektowe, prawa i zasady oraz kategorię inne dla dwóch aspektów programowania nie wpisujących się w poprzednie trzy kategorie. Dla poprawienia czytelności oraz w celu stworzenia pewnego rodzaju mapy tego rozdziału cały podział zaprezentowano na Rysunku 1.



Rysunek 1. Obszary mierzone przy użyciu metryk statycznej analizy oprogramowania - podział.

3.1 Liczba artefaktów i zależności miedzy nimi

Liczba artefaktów. Najprostszą formą pomiaru kodu źródłowego jest policzenie różnego rodzaju artefaktów takich jak: linie kodu, pola w klasie, atrybuty metod, metody, klasy, interfejsy, klasy abstrakcyjne, pakiety. Można zliczać artefakty całościowo w systemie, albo też na różnym poziomie abstrakcji np. liczba metod w klasie, czy liczba klas w pakiecie, i na tej podstawie próbować wyciągać wnioski o jakości analizowanego systemu, czy jego fragmentu. Badanie liczby występowania danego artefaktu w typie o poziomie abstrakcji o jeden stopień wyższym (np. klas w pakiecie) pozwala również wysnuć wnioski o złożoności tego typu.

Liczba zależności. Szacowanie różnego rodzaju zależności – powiązań (ang. *coupling*) między artefaktami stanowi również obszar mierzony przy użyciu metryk. Analiza tych powiązań, ich liczby i charakteru, pozwala ocenić złożoność badanego systemu, czy jego fragmentu. Dzięki analizie powiązań projektant może mieć kontrolę nad złożonością, bo lepiej rozumie zależności między komponentami.

3.2 Paradygmaty programowania zorientowanego obiektowo

Programowanie zorientowane obiektowo wprowadziło wiele mechanizmów, a także posiada wiele cech, które pozwalają na tworzenie produktów o wysokiej jakości. Poniżej omówione pokrótce wybrane paradygmaty programowania zorientowanego obiektowo, które stanowią przedmiot pomiaru metryk statycznej analizy oprogramowania. Poniższe definicje powstały w oparciu o książkę Booch'a [4].

Dziedziczenie *(ang. inheritance)***.** Jest to relacja między klasami, w której klasa dziedzicząca nabywa charakterystykę klasy, z której dziedziczy. Umożliwia ono tworzenie specjalizowanych obiektów na podstawie bardziej ogólnych. Dzięki dziedziczeniu wprowadzamy mechanizm powtórnego użycia kodu, przez co ograniczamy ilość zduplikowanego kodu.

Hermetyzacja *(ang. encapsulation)*. Polega ona na ukrywaniu implementacji. Na zewnątrz rozpatrywanego modułu widoczny jest tylko interfejs, który definiuje zakres działania dopuszczalny na nim przez moduł zewnętrzny. Dzięki temu, poza wybranym

zakresem, moduły zewnętrzne nie mogą zmieniać stanu rozpatrywanego modułu. Z hermetyzacją wiąże się też ukrywanie informacji. Na przykładzie klasy: jedynie część metod i pól jest publicznych, reszta jest jedynie do użytku wewnętrznego klasy.

Polimorfizm *(ang. polymorphism)*. Mechanizm pozwalający na używaniu referencji i kolekcji obiektów różnych typów przy jednoczesnej pewności, iż w razie użycia metody dla danej referencji obiektu, metoda ta wywoła zachowanie obiektu odpowiednie dla jego pełnego typu.

Spójność *(ang. cohesion)***.** Jest to stopień wewnętrznego powiązania i zakresu odpowiedzialności rozpatrywanego modułu. Im moduł jest bardziej spójny tym lepiej. Moduł spójny powinien być trudny do podziału i operować na tych samych danych.

3.3 Prawa i zasady programowania zorientowanego obiektowo

Wraz z rozwojem programowania zorientowanego obiektowo zaczęto formułować prawa i zasady wspierające dobre praktyki w pisaniu programów przekładające się na ich jakość. Zasady i prawda doradzają jakie praktyki należy stosować, a jakich należy unikać.

3.3.1 Prawo Demeter

Jest to prawo, które zostało zaprezentowane w roku 1988 na konferencji OOPSLA w wystąpieniu K. J. Lieberherr'a, I. M. Holland'a, A. J. Riel'a [5]. Prawo Demeter mówi, że metoda danego obiektu może odwoływać się jedynie do metod:

- należących do tego samego obiektu,
- metod dowolnego parametru przekazanego do niej,
- dowolnego obiektu przez nią stworzonego,
- dowolnego składnika klasy do której należała dana metoda.

W praktyce oznacza to, że obiekt powinien unikać wywołania metod klasy, której obiekt został mu dostarczony przez inny obiekt. np. w języku Java kod `a.b.metoda()` łamie prawo Demeter, podczas gdy `a.metoda()` tego prawa nie łamie. Tak więc prawo Demeter ogranicza długie łańcuchy wywołań. Popularnie jest ono wyrażane: „Nigdy nie rozmawiaj z obcymi".
Stosowanie prawa Demeter, ułatwia pielęgnację, zmniejsza współczynnik błędów, ogranicza powiązania między obiektami, wzmacnia hermetyzację i abstrakcję obiektów, a także powoduje przeniesienie odpowiedzialności za dostęp do metod i atrybutów klasy z obiektu odwołującego się do nich na ich właściciela.
Poniższy przykład ilustruje naruszenia prawa Demeter (za PMD – sekcja 5.2.10):

```
  public class Example {
      public void example(Bar b) {
          // poprawne wywołanie, bo b jest parametrem metody example
          C c = b.getC();
          //naruszenie LOD - używamy obiektu c otrzymanego od b
          // powinniśmy zrobić wywołanie przez b np. „b.doitonC();”
          c.doIt();
          //to jest również naruszenie wyrażone łańcuchem wywołań
          b.getC().doIt();
      }
}
```

Kod źródłowy 1. Przykład naruszeń prawa Demeter.

3.3.2 Zasady SOLID

Kolejne trzy zasady zostały zebrane z propozycji różnych autorów i przedefiniowane przez Roberta Martina i w formie tutaj podanej zaprezentowane przez niego w roku 2000 [6]. Do tych trzech zasad Martin dodał jeszcze dwie swoje i razem określił wszystkie mnemoniką SOLID od pierwszych liter zasad, do których się odwołują. Wszystkie dotyczą poziomu klas, ponadto zasada otwarte/zamknięte oraz zasada odwróconych zależności ma zastosowanie na wszystkich poziomach abstrakcji.

Zasada jednej odpowiedzialności - SRP *(ang. single responsibility principle).* Jest to zasada, która odnosi się do spójności klasy i została sformułowana przez Tom’a DeMarco w roku 1979 [7]. Zasada ta mówi, że nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy. Jeśli klasa spełnia dwie funkcjonalności dla różnych aplikacji, to istnieje ryzyko, że zmiana jednej funkcjonalności dla jednej aplikacji spowoduje błąd w drugiej.

Zasada otwarte/zamknięte - OCP *(ang. open/closed principle).* Zasada zaproponowana przez Bertranda Meyer’a w roku 1988 [8]. Według Martina jest to najważniejsza zasada programowania obiektowego [6]. Zasada ta mówi, że różnego rodzaju typy (np. klasy albo pakiety) powinny być otwarte na rozszerzanie, ale zamknięte na modyfikacje. Oznacza to, że można zmienić zachowanie takiego komponentu bez zmiany kodu. Zasada OCP jest realizowana przy użyciu typów abstrakcyjnych.

Zasada podstawienia Liskov - LSP *(ang. Liskov substitution principle).* Zasada zaproponowana przez Barbarę Liskov [9]. Zasada ta została zdefiniowana w następujący

sposób: funkcje, które używają wskaźników bądź referencji do klas bazowych, muszą być w stanie używać obiektów klas dziedziczących po klasach bazowych bez dokładnej znajomości tych obiektów. Oznacza to w praktyce, że jeśli obiekt S dziedziczy po obiekcie T, to możliwe jest podstawienie za obiekt T obiektu S. Wiąże się to również z tym, że obiekt S musi „zachowywać się" tak samo jak obiekt T. Jeśli zasada LSP jest łamana, to wtedy łamana jest też zasada OCP ponieważ taka funkcja musi „wiedzieć" o wszystkich klasach dziedziczących po klasie bazowej i musi być modyfikowana za każdym razem, gdy dodawana jest nowa klasa dziedzicząca.

Zasada segregacji interfejsów - ISP *(ang. interface segregation principle).* Zasada ta mówi, że, klient powinien mieć dostęp przez interfejs jedynie do metod klasy, które używa. Inaczej mówiąc klient nie powinien zależeć od metod, których nie używa. W praktyce oznacza to często wprowadzenie kilku interfejsów dla jednej klasy.

Stosowanie zasady segregacji interfejsów zmniejsza liczbę powiązań w systemie, a co za tym idzie ułatwia późniejsze wprowadzanie koniecznych zmian bez konieczności propagacji tych zmian w obiektach nie zainteresowanych daną funkcjonalnością.

Zasada odwrotnych zależności - DIP *(ang. dependency inversion principle).* Zasada sformułowana przez Roberta Martina [6]. Zasada ta wyrażona jest w dwóch sformułowaniach:

- moduły wysokiego poziomu nie powinny zależeć od modułów niższego poziomu, obydwa powinny zależeć od typów abstrakcyjnych,
- typy abstrakcyjne nie powinny zależeć od szczegółowych. Typy szczegółowe powinny zależeć od typów abstrakcyjnych.

Jeśli moduły wysokiego poziomu zależą od modułów niskiego poziomu, wtedy zmiany w modułach niskiego poziomu mogą mieć bezpośredni wpływ na moduły wysokiego poziomu co jest sytuacja niepożądaną. Ponadto w takim wypadku ograniczamy możliwość użycia modułów wysokiego poziomu w innych kontekstach, czyli ograniczamy możliwość ich rozszerzania.

Na następnej stronie (kod źródłowy 2) zaprezentowano przykład implementacji bez użycia zasady odwróconych zależności (po lewej) i z użyciem zasady odwróconych zależności (po prawej) – przykład za [10].

Mamy klasę wysokiego poziomu Manager (załóżmy, że jest to bardzo skomplikowana klasa) i klasę niskiego poziomu Worker. Chcemy dodać nowego specyficznego pracownika SuperWorker. W przypadku pierwszym napotkamy na szereg trudności. Musimy zmienić

klasę Manager, a ponieważ jest skomplikowana, to będzie wymagało od nas dużego wysiłku. Ponadto nowe zmiany mogą mieć wpływ na aktualną funkcjonalność klasy Manager.
Przy użyciu zasady odwróconych zależności nie ma takich problemów. Musimy jedynie wprowadzić interfejs IWorker, który będzie implementowany przez klasę Worker i SuperWorker. W ten sposób wprowadziliśmy warstwę abstrakcyjną do naszych zależności.

Stosowanie zasady DIP pozwala uniknąć następujących cech programu:

- sztywności: trudny do zmiany ze względu na wpływ na inne części programu,
- kruchości: zmiany w programie powodują nieprzewidywalne błędy,
- nieelastyczności: trudny do powtórnego użycia.

```
class Worker {
	public void work() {
		// ....working
	}
}
class Manager {
	Worker m_worker;
	public void setWorker(Worker w) {
		m_worker=w;
	}

	public void manage() {
		m_worker.work();
	}
}
class SuperWorker {
	public void work() {
		//.... working much more
	}
}
```

```
interface IWorker {
	public void work();
}
class Worker implements IWorker{
	public void work() {
		// ....working
	}
}
class SuperWorker  implements IWorker{
	public void work() {
		//.... working much more
	}
}
class Manager {
	IWorker m_worker;
	public void setWorker(IWorker w) {
		m_worker=w;
	}
	public void manage() {
		m_worker.work();
	}
}
```

Kod źródłowy 2. Przykład użycia zasady odwróconych zależności DIP ilustrujący plusy takiego rozwiązania na tle implementacji bez stosowania DIP za [10].

3.3.3 Zasady dotyczące modułów

Kolejne sześć zasad dotyczy sposobu organizacji pakietów, ale mogą być też używane dla większych modułów. Pierwsze trzy dotyczą spójności pakietów, kolejne trzy powiązań

między pakietami. Wszystkie zaprezentowane tutaj zasady zostały przedstawione przez Roberta Martina w roku 2000 [6]. Reguły te stanowią wytyczne do konstruowania pakietów, które są pewnego rodzaju kontenerami dla klas.

3.3.3.1 Zasady związane ze spójnością

Zasada równoważności użycia i wydania - REP (*ang. reuse/release equivalency principle).* Zasada ta mówi, że należy używać jedynie takich pakietów, które zostały wydane przez ich twórcę i mają formalny numer wydania. Użycie jest tu rozumiane jako użycie gotowego pakietu.

Zasada ta wynika z faktu, że użytkownicy gotowych pakietów nie muszą znać ich zawartości, ale każdorazowo powinni być informowani o zmianach w nich zachodzących, tak aby móc efektywnie korzystać z ich zawartości. Wydanie gwarantuje też pewność jakości pakietu.

Zasada wspólnego użycia - CRP *(ang. common reuse principle)*. Klasy w jednym pakiecie są używane wspólnie. Jeśli używasz jednej klasy z pakietu, używasz ich wszystkich.

Zasada ta pomaga w podjęciu decyzji jakie klasy należy umieszczać w obrębie tego samego pakietu.

Zasada wspólnego zamknięcia - CCP *(ang. common closure principle).* Klasy w jednym pakiecie powinny być wspólnie zamknięte na zmiany. Zmiana, która dotyczy pakietu, dotyczy wszystkich klas w pakiecie.

Zasada ta powiązana jest z pielęgnowalnością. Dążmy do tego, aby zmiany jakie chcemy dokonać w systemie zawierały się w obrębie jednego pakietu, a nie były rozproszone pomiędzy różne pakiety. Dzięki temu inne pakiety, których zmiana nie dotyczy nie wymagają rewalidacji i redystrybucji.

3.3.3.2 Zasady dotyczące powiązań

Zasada acyklicznych zależności ADP *(ang. the acyclic dependency principle).* Przy projektowaniu pakietów, a także zbioru pakietów należy unikać cyklicznych zależności. Struktura zależności powinna stanowić skierowany graf acykliczny DAG (*ang. directed acyclic graph*).

Cykliczne zależności sprawiają, że kod jest trudny w testowaniu. Różne fragmenty systemu nie są od siebie w odpowiedni sposób odizolowane. Powoduje, to że pojawiają nam się w systemie zależności pomiędzy pakietami, których można by uniknąć. Skutkuje to również problemami z redystrybucją pakietów, bo w takim wypadku wszystkie pakiety w cyklu powinny być wydawane w jednym czasie. Oznacza to, że właściwie powstaje nam

jeden duży pakiet. Pojawia się też problem z pielęgnowalnością ponieważ zmiany w jednym pakiecie mogą mieć wpływ na inne pakiety w cyklu.

Są dwie możliwości aby przerwać cykl:

- Stworzyć nowy pakiet i przenieść do niego klasy, które doprowadziły do zamknięcia cyklu.
- Korzystając z zasady odwrotnych zależności DIP zachowujemy jedną zależność, a drugą realizujemy przy pomocy interfejsu.

Przykład cyklu na diagramie zależności można znaleźć na Rysunek 5 w rozdziale 6 dotyczącym zastosowania metryk w praktyce – czerwona krawędź wskazuje gdzie jest cykl.

Zasada stabilnych zależności - SDP *(ang. stable dependency principle).* Zależności pomiędzy pakietami powinny być ułożone w kierunku bardziej stabilnych pakietów. Pakiet powinien zależeć jedynie od pakietów, które są bardziej stabilne od niego.

Projektowany system powinien być podatny na zmiany zgodnie z zasadą wspólnego zamknięcia CCP. Oznacza to, że część pakietów powinna być podatna na określone zmiany. Zasada SDP mówi o tym, że pakiety, które zostały zaprojektowane tak, aby być podatnymi na zmiany powinny być konkretne i zależeć od pakietów bardziej stabilnych (typy abstrakcyjne), czyli mniej podatnych na zmiany.

Zasada stabilne abstrakcyjne - SAP *(ang. stable asbstractions principle).* Pakiety, które są najbardziej stabilne powinny być też najbardziej abstrakcyjne (składać się z interfejsów i klas abstrakcyjnych). Niestabilne pakiety powinny się składać z klas konkretnych. Abstrakcyjność pakietu powinna być proporcjonalna do jego stabilności.

Zasada ta ustala relację pomiędzy stabilnością i abstrakcyjnością. Mówi ona, że stabilny pakiet powinien być także abstrakcyjny, dzięki temu, mimo, że jest stabilny, ma możliwość rozszerzenia poprzez dziedziczenie. Z drugiej strony mówi o tym, że niestabilne pakiety powinny być konkretne, dzięki czemu są łatwo modyfikowalne.

3.4 Inne aspekty jakościowe podlegające pomiarom

W tym rozdziale przedstawione zostaną dodatkowe dwa aspekty powiązane z analizą statyczną oprogramowania, które mają przełożenie na jakość. W przypadku testowania jednostkowego przy pomocy metryk można mierzyć stopień pokrycia testami. W przypadku zduplikowanego kodu metryka powiązana z tym zagadnieniem pomaga oszacować skalę tego zjawiska.

3.4.1 Testy jednostkowe

Testowanie jednostkowe stanowi jedną z najważniejszych metod zapewniania jakości pisanego kodu. Pozwala ono zweryfikować poprawność działania poszczególnych elementów programu – metod lub obiektów.

Definicja testu jednostkowego. Test jednostkowy to test pojedynczej, wyizolowanej instancji klasy w różnych przypadkach testowych. Jako przypadek testowy traktujemy sprawdzenie czy dana metoda zachowuje się poprawnie w określonych warunkach. Zazwyczaj więc na metodę może przypadać kilka przypadków testowych [11].

Przypadki testowe dotyczące poszczególnych klas (ich instancji) są zebrane w jednej klasie testującej tak żeby jednej klasie testowanej odpowiadała jedna klasa testująca. Testowanie jednostkowe pozwala programiście na wprowadzenie zmian na późnym etapie rozwoju oprogramowania, dając pewność, że dany moduł nadal działa. Pakiet testów jednostkowych powinien być uruchamiany każdorazowo przy okazji ingerencji w kod źródłowy w celu potwierdzenia, że wprowadzona zmiana nie spowodowała błędu.

Testy jednostkowe powinny mieć następujące cechy:

- Niezależność - dwa dowolne testy nie mogą na siebie wpływać.
- Powtarzalność - możliwość uruchomienia testów w dowolnym momencie, bez konieczności uruchamiania dodatkowych elementów. Powtarzalność oznacza także, że dla tego samego zestawu danych otrzymujemy taki sam rezultat.
- Jednoznaczność - jasna odpowiedź na testowaną funkcjonalność.
- Jednostkowość - w jednym teście sprawdzana może być tylko jedna funkcjonalność.

Testy jednostkowe, jak każdy rodzaj testów oprogramowania mogą pokazać obecność błędów, ale nie dają pewności, że błędów nie ma. Ważnym narzędziem dla testów jednostkowych jest system kontroli wersji, który pozwala w wypadku wystąpienia błędu na porównanie wcześniej działającego poprawnie kodu z obecnym błędnie działającym i na tej podstawie znalezienie miejsca powodującego błąd.

3.4.2 Zduplikowany kod

Zduplikowany kod to fragment kodu, który powstaje poprzez kopiowanie kodu, przy czym mogą pojawić się drobne zmiany np. w nazewnictwie, albo zmiana rodzaju pętli natomiast sposób działania kodu pozostaje ten sam.

Według badań przeprowadzonych przez Bexter'a dla dużych projektów 5-10% kodu to kod zduplikowany [12]. Duplikaty należy usuwać przede wszystkim w sytuacji, gdy dodaje się nowe funkcje do systemu, trzeba poprawić błąd w programie, podczas przeglądów kodu oraz w sytuacji gdy występują ograniczenia co do rozmiaru kodu.

Istnieje wiele metod pozwalających na wykrywanie duplikatów w kodzie. Najprostszym sposobem jest porównywanie ciągu znaków. Zaletą takiego rozwiązania jest niezależność od języka oprogramowania. Wadą jest skuteczność, ponieważ wystarczy zmienić np. nazwę metody, aby duplikat nie został wykryty. Innym rozwiązaniem jest usuwanie komentarzy i zmiana nazw zmiennych na ustalony format. Takie podejście wymaga już uwzględniania standardów języka. Natomiast tutaj również niewielkie zmiany nie mające większego wpływu na działanie programu mogą spowodować niewykrycie duplikatu. Całkiem innym podejściem jest definiowanie drzewa składniowego fragmentu kodu (ang. *syntax tree*). Drzewo takie odzwierciedla strukturę funkcjonalną kodu. Nie uwzględniane są komentarze. Kod dzielony jest na fragmenty, ustalane jest drzewo składniowe, następnie wyliczane są metryki, których wartości są porównywane i na tej podstawie wysuwana jest hipoteza o znalezieniu duplikatu [13]. Przykładem wykorzystywanych metryk są np. metryki Kontogiannisa, a wśród nich opisana w tej pracy złożoność cyklomatyczna McCabe'a (sekcja 4.1.2). Wykorzystywana jest także złożoność danych, złożoność strukturalna, zmodyfikowana metoda punktów funkcyjnych Albrechta oraz metryki jakościowe Henry-Kafura. Cały zestaw został opisany przez Kontogiannisa w [14]. Według badań przeprowadzonych przez Pietrzaka, Nawrockiego i Waltera przy użyciu metryk Kontogiannisa ok. 60% wykrywanych duplikatów to rzeczywiste duplikaty [13].

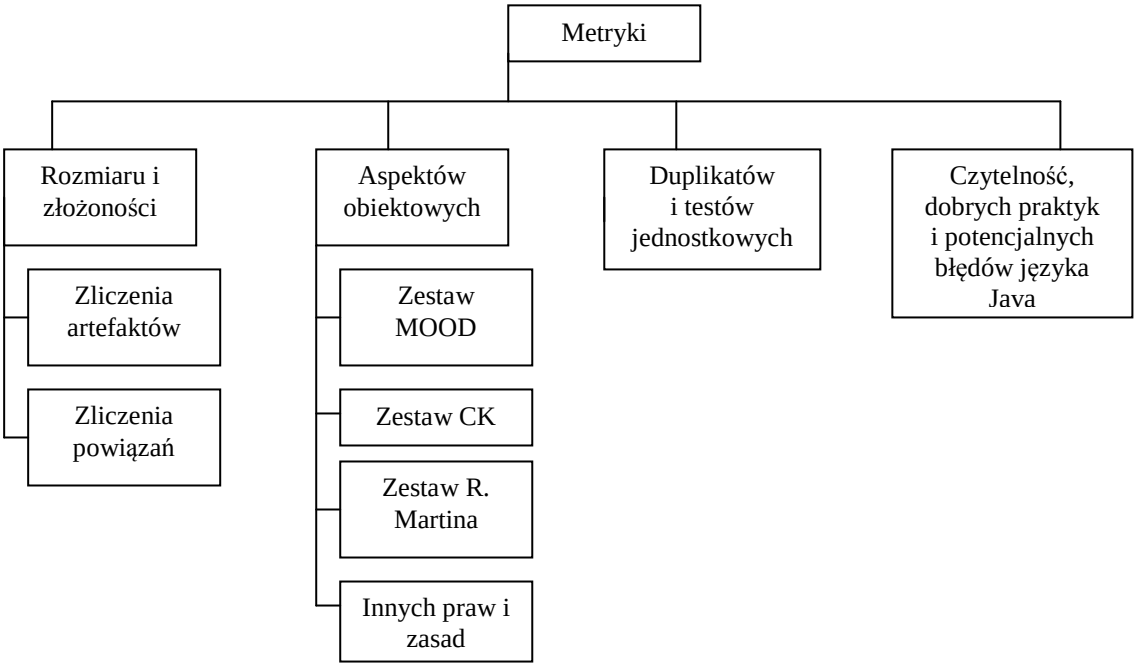
3.5 Podsumowanie

W rozdziale 3 omówione zostały obszary podlegające pomiarom przy użyciu metryk statycznej analizy oprogramowania. W sposób naturalny pierwszym obszarem są zliczenia artefaktów i różnego rodzaju powiązań między nimi na różnych poziomach abstrakcji. Badania tego rodzaju metryk powinny dawać pogląd na temat złożoności artefaktów wszystkich poziomów abstrakcji a także całego systemu. W przypadku języków obiektowych można badać stopień wykorzystania mechanizmów i cech związanych z paradygmatami programowania zorientowanego obiektowo. Ich wykorzystanie stanowi o jakości projektu obiektowego i przekłada się na zewnętrzne cechy jakości oprogramowania takie jak: funkcjonalność, niezawodność, przenośność, efektywność, pielęgnowalność, czy łatwość w

zarządzaniu. Wreszcie omówione zostały prawa i zasady dla programowania zorientowanego obiektowo, których przestrzeganie stanowi dobrą praktykę programistyczną. Nie wszystkie wymienione tutaj prawa i zasady mogą być kontrolowane przy użyciu metryk, omówione zostały jednak wszystkie ze względu na ich komplementarność. Te, które można kontrolować przy użyciu metryk to: prawo Demeter, SRP, OCP, DIP, ADP, SDP, SAP. Omówiono także zalety związane z przestrzeganiem poszczególnych praw, a także ryzyka związane z ich łamaniem. W ramach statycznej analizy oprogramowania można jeszcze badać pokrycie testami jednostkowymi oraz ilość zduplikowanego kodu. Zagadnienia kryjące się za tymi metrykami również zostały tutaj przedstawione. W kolejnym rozdziale można zapoznać się szczegółowo z metrykami analizy statycznej powiązanymi z omówionymi obszarami.

4 Rodzaje metryk kodu w analizie statycznej

W poprzednim rozdziale omówiono obszary związane z kodem źródłowym, które można badać przy udziale metryk statycznej analizy oprogramowania. W tym rozdziale szczegółowo omówione zostaną metryki w kluczu: definicja oraz interpretacja. Podane zostaną również wartości preferowane metryk, które znaleziono w literaturze lub narzędziach je implementujących. Zostanie również zaprezentowane aktualny stan badań nad weryfikacją znalezionych metryk. Struktura tego rozdziału mniej więcej odpowiada strukturze poprzedniego rozdziału. Pierwszy podrozdział dotyczy metryk zajmujących się zliczaniem artefaktów i powiązań między nimi na różnych poziomach abstrakcji. Drugi opisuje kilka zestawów metryk. Zestawy MOOD i CK stanowią głównie odzwierciedlenie użycia paradygmatów programowania zorientowanego obiektowo, a zestaw Martina praw i zasad dotyczących pakietów. Następnie jest podrozdział dotyczący metryk powiązanych z duplikatami i testowaniem jednostkowym. Na koniec pojawia się dodatkowy rozdział związany ze specyficznymi narzędziami języka Java (PMD, Checkstyle i Findbugs – wszystkie opisane w rozdziale 5.2), dzięki którym stworzono zestaw trzech metryk dotyczących dobrych praktyk, czytelności kodu oraz potencjalnych błędów. Na Rysunku 2 zaprezentowano podział metryk statycznej analizy oprogramowania i jednocześnie mapę tego rozdziału.



Rysunek 2. Podział metryk analizy statycznej oprogramowania.

4.1 Metryki rozmiaru i złożoności.

W podrozdziale tym zostaną przedstawione metryki, które dotyczą zliczeń artefaktów, a także zliczeń powiązań między artefaktami na różnych poziomach abstrakcji. Te dwa rodzaje metryk stanowią podstawę do określenia rozmiaru i złożoności wybranych modułów programu bądź nawet całego systemu. Metryki dotyczące metod znajdują zastosowanie również w przypadku procedur z języków proceduralnych – właśnie takie jest ich pochodzenie.

4.1.1 Metryki zliczeń

W rozdziale tym przedstawiono kilka podstawowych metryk zliczeń, które uznano za najbardziej kluczowe. Wiele narzędzi implementuje więcej metryk rozmiaru np. różne wartości średnie, albo nawet mierzy to samo na różnych poziomach abstrakcji.

- **Lines of code (LOC)** – (linie kodu) – prawdopodobnie najstarsza metryka używana jeszcze w języka proceduralnych. Czasem występuje pod nazwą Non commenting

source statement (NCSS) (wyrażenia nie będące komentarzami). Metryka NCSS wraz z narzędziem JavaNCSS została zaproponowana przez Chr. Clemens Lee [15].

Definicja. Mierzy ilość linii kodu w rozpatrywanym typie (metoda, klasa, pakiet). Zazwyczaj nie są zliczane linie komentarzy, puste linie, importy. Dla NCSS w praktyce zliczane są znaki ‘ ;’ i ‘{‘. Nie są zliczane puste wyrażenia, bloki, albo średniki po nawiasach zamykających.

Interpretacja. Metryka ta może być używana jako wskaźnik za dużych metod i klas. Narzędzie Stan (sekcja 5.2.1) proponuje następujące przedziały wartości metryki:

- Dla metod: <60 – wartości dobre, 60-120 wartości niezalecane, >120 zalecane do refaktoryzacji
- Dla klas: <300 – wartości dobre, 300-400 wartości niezalecane, >400 zalecane do refaktoryzacji

- **Number of fields (NOF)** – liczba pól.

  Definicja. Jest to liczba pól w klasie.

  Interpretacja. Liczba pól jest jednym z najprostszych wyznaczników złożoności klasy. Im więcej pól tym klasa bardziej złożona. Odpowiednio dobrane pola stanowią również o spójności klasy. Stan (sekcja 5.2.1) jako optymalne wartości metryki sugeruje wartości mniejsze niż 20, dopuszczalne są również wartości 20-40, powyżej 40 zalecany jest podział klasy.

- **Number of methods (NOM)** – liczba metod.

  Definicja. Jest to liczba metod w klasie.

  Interpretacja. Liczba metod, to obok liczby pól kolejna metryka, która wyznacza złożoność klasy. Im więcej metod, tym klasa bardziej złożona. Stan (sekcja 5.2.1) jako optymalne wartości metryki sugeruje wartości mniejsze niż 50, dopuszczalne są wartości 50-100, powyżej 100 zalecany jest podział klasy.

- **Top level classes (TLC)** - liczba klas z wyłączeniem klas wewnętrznych.

  Definicja. Jest to liczba klas z wyłączeniem klas wewnętrznych zdefiniowana dla pakietów.

  Interpretacja. Najprostsza metryka mówiąca o wielkości, a zarazem złożoności pakietów. Stan (sekcja 5.2.1) jako optymalne wartości metryki zakłada wartości >40. Dla wartości >60 zalecany jest podział pakietu.

4.1.2 Metryki powiązań

- **Cyclomatic complexity (CC)** – *(złożoność cyklomatyczna)* - metryka zaproponowana przez T. Mc Cabe'a w 1976 roku [16] używana do obliczania złożoności metod lub procedur. Wykorzystywana również w celu policzenia metryki WMC z zestawu CK (sekcja 4.2.2).

  Definicja. W grafie reprezentującym przepływ sterowania w metodzie lub procedurze:

  $$CC = e - n + 2 \quad (1)$$

  Gdzie e – liczba krawędzi (możliwych przejść), n – liczba wierzchołków w grafie (instrukcji).

  Używa się też uproszczonej wersji metryki:

  $$CC = d + 1 \quad (2)$$

  Gdzie: d – liczba węzłów decyzyjnych w grafie (w praktyce zlicza się ilość instrukcji `if`, `while`, `do`, `for`, `?:`, `catch`, `switch`, `case` i operatorów `&&`, `||`).

  CC jest liczbą niezależnych ścieżek w grafie reprezentującym przepływ sterowania w metodzie. Ponieważ skoki wstecz mogą powodować nieskończony wzrost liczby takich ścieżek, mierzy się liczbę ścieżek bez uwzględniania cykli.

  Interpretacja. Wysokie wartości metryki CC wskazują na to, że metoda jest zbyt skomplikowana, przez co zwiększa się prawdopodobieństwo wystąpienia błędu. Wartości 1-4 są uważana za dobre, 8-10 jako zalecane do refaktoryzacji, powyżej 11 konieczna jest refaktoryzacja (rekomendacja narzędzia Checkstyle sekcja 5.2.9). Przykład funkcji o niskiej wartości CC pokazano na Rysunku 3.
- **NPathComplexity (NPC)** – zaproponowane w narzędziu Checkstyle (sekcja 5.2.9) dotyczy metod lub procedur.

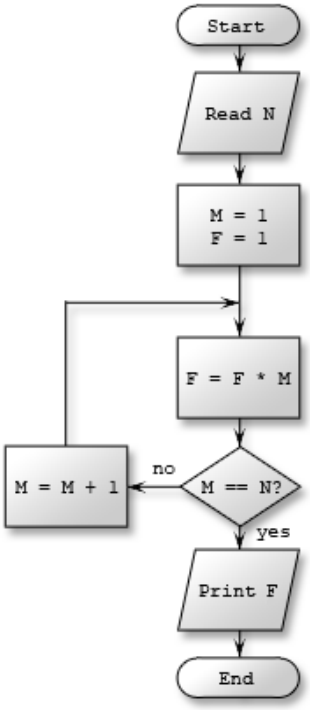
  Definicja. NPC to liczba możliwych do wykonania się ścieżek w ramach metody
  Interpretacja. Checkstyle jako maksimum dozwolonej liczby wykonania się ścieżek metody sugeruje wartość 200 (powyżej zalecane rozbicie metody na mniej skomplikowane).
- **FAT** – metryka zaproponowana przez twórców programu Stan (sekcja 5.2.1) na wzór komercyjnego narzędzia structure101 firmy Headway Software [17], która stanowi uproszczone ugólnienie metryki CC na klasy i pakiety.

Definicja.

Dla klas: FAT jest to liczba zależności pomiędzy metodami i polami w rozpatrywanej klasie.

Dla pakietów: FAT jest to liczba zależności pomiędzy klasami wchodzącymi w skład pakietu.

Interpretacja. Autorzy Stan'a zalecają zarówno dla klas, jak i pakietów wartości FAT < 60. Dopuszczalne są wartości 60-120, powyżej 120 zalecany jest podział klasy lub pakietu. Wyższe wartości metryki mogą sugerować, że rozpatrywany pakiet lub klasa ma zbyt dużą odpowiedzialność i należy rozważyć podział na mniejsze klasy, czy pakiety.



Rysunek 3. Przykład funkcji o niskiej wartości CC: 8 węzłów, 8 krawędzi. CC = 8 – 8 +1 = 1. Źródło za [18].

- **Class Data Abstraction Coupling (DAC)** – źrodło Checkstyle (sekcja 5.2.9)

  Definicja. Jest to liczba obiektów innych klas, które zawiera w sobie rozpatrywana klasa

  Interpretacja. Im większa wartość DAC, tym klasy są bardziej skomplikowane. Wartość progowa zdefiniowana przez Checkstyle to 3.
- **Class Fan Out Complexity (CFOC)** – źródło Checkstyle (sekcja 5.2.9)

Definicja. Jest to liczba innych klas, które odwołują się do danej klasy

Interpretacja. Wartość progowa zdefiniowana przez Checkstyle to 20.

- **Avarage Component Dependency (ACD)** – źródło Stan (sekcja 5.2.1) .

  Definicja. Zależność komponentu (CD) zdefiniowana jest jako liczba innych komponentów, które zależą od rozpatrywanego bezpośrednio lub pośrednio. Jest to zależność bezwzględna. Można też zdefiniować zależność względną - procent komponentów (np. klas w pakiecie albo pakietów w bibliotece pomniejszonych o jeden – rozpatrywany komponent), które zależą od rozpatrywanego komponentu.

  Średnia zależność komponentu (ACD) to średnia względna wartość zależności komponentu CD (klas w pakiecie lub pakietów w bibliotece).

  Interpretacja. Preferowane są niskie wartości. Autorzy jednak nie podają wartości progowych. Dopuszczalne wartości będą się zmieniały w zależności od wielkości modułu, w którym osadzony jest dany komponent.

  Trzy przypadki są warte odnotowania.

  - ACD = 0 jeśli nie ma połączeń pomiędzy komponentami w rozpatrywanym module.
  - ACD = 50% jeśli wszystkie komponenty w module tworzą łańcuch.
  - ACD = 100% jeśli wszystkie komponenty w module tworzą jeden cykl

  Ponadto, jeśli ACD > 50% oznacza to, że w rozpatrywanym module znajduje się cykl (patrz metryka TANGLED 4.2.4).

4.2 Metryki obiektowe

Metryki obiektowe, to metryki, który w sposób ilościowy opisują zastosowania różnych paradygmatów, praw lub zasad w obiektowych językach programowania. Stanowią dobre narzędzie do oceny jakości programów obiektowych, a także pozwalają ocenić wykorzystanie różnych własności oprogramowania zorientowanego obiektowo. Do najbardziej znanych należą metryki z zestawu MOOD, CK oraz Martina. Na koniec omówione zostaną metryki praw i zasad, które nie zostały ujęte w wymienionych zestawach.

4.2.1 Metryki z zestawu MOOD

Jest to zestaw 6 metryk zaproponowanych przez F. B. e Abreu w 1994 roku (MOOD – Metrics of Object Oriented Design)[19]. Część z nich została jeszcze zmodyfikowana i zaprezentowana w roku 1995 przez tych samych autorów i w takiej postaci są tutaj prezentowane[20]. Z pracy tej pochodzą też wartości preferowane metryk, które zostały podane w akapicie dotyczącym interpretacji metryk. Wartości zostały ustalone na podstawie badania pięciu systemów informatycznych napisanych w języku C++: Microsoft Fundation Classes, GNU glib++, ET++ library, NewMat library oraz MotifApp library. Łącznie badano 595 klas. Autorzy starannie dobierali badane oprogramowanie, aby było one dobre jakościowo i mogło posłużyć do zdefiniowania wartości preferowanych.

Celem tych metryk jest ocena stopnia wykorzystania następujących mechanizmów obiektowych: hermetyzacji, dziedziczenia, powiązań i polimorfizmu. Są wyrażone bezwymiarowo jako stopień ich wykorzystania w systemie. Obiektem pomiaru jest tutaj cały system, a nie poszczególne klasy i ich relacje. Metryki te dają więc pogląd na jakość całego projektu, tym bardziej, że dobrze się skalują i nie zależą od wielkości systemu. Nie dają natomiast miarodajnego rezultatu w przypadku systemów opartych o formularze, generację kodu i moduły współdzielone przez wszystkie elementy systemu, w których przydział odpowiedzialności do klas zachodzi w oparciu o różne kryteria [21].

4.2.1.1 Definicja i interpretacja metryk z zestawu MOOD

- **Attribute Hiding Factor (AHF)** – (*współczynnik ukrycia atrybutów)*

  Definicja. AHF określa stopień hermetyzacji atrybutów w klasie:

$$AHF = \frac{\sum_{i=1}^{TC} a_h(c_i)}{\sum_{i=1}^{TC} a_d(c_i)} \tag{3}$$

  Gdzie: *TC* – liczba wszystkich klas, c_i – kolejne klasy, a_h, a_d – atrybuty kolejno niepubliczne i wszystkie zdefiniowane w c_i.

  Interpretacja. Współczynnik ten powinien być bliski 100%, gdyż jest dobrze znanym postulatem, aby dostęp do atrybutów w klasie odbywał się poprzez metody.

- **Method Hiding Factor (MHF) -** (*współczynnik ukrycia metod)*

  Definicja. MHF określa stopień hermetyzacji metod w klasie.

$$MHF = \frac{\sum_{i=1}^{TC} m_h(c_i)}{\sum_{i=1}^{TC} m_d(c_i)} \quad (4)$$

Gdzie: *TC* – liczba wszystkich klas, m_d- metody zdefiniowane w c_i, m_h – metody niepubliczne w c_i

Interpretacja. Pożądane wartości znajdują się w przedziale 10-25%. Część metod wewnątrz klasy nie powinna być ogólnie dostępna.

- **Attribute Inheritance Factor (AIF)** – *(współczynnik dziedziczenia atrybutów)*

  Definicja. AIF określa stopień wykorzystania dziedziczenia dla atrybutów. Stanowi stosunek atrybutów odziedziczonych do wszystkich obecnych.

$$AIF = \frac{\sum_{i=1}^{TC} a_i(c_i)}{\sum_{i=1}^{TC} a_a(c_a)} \quad (5)$$

  Gdzie: *TC* – liczba wszystkich klas, $a_a(c_i) = a_d(c_i) + a_i(c_i)$ - atrybuty dostępne w klasie c_i, $a_i(c_i)$ – atrybuty odziedziczone (i nie zmienione) w klasie c_i, $a_d(c_i) = a_n(c_i) + a_o(c_i)$ - atrybuty zdefiniowane w klasie c_i, gdzie $a_n(c_i)$ – atrybuty nowe, $a_o(c_i)$ – atrybuty odziedziczone i zmienione.

  Należy zwrócić uwagę, że dziedziczenie jest jednym z narzędzi dostępnych w obiektowych językach oprogramowania, stopień jego wykorzystania nie stanowi jednak bezpośrednio o jakości projektu (dotyczy się także MIF).

  Interpretacja. Typowo współczynnik zyskuje wartości 50-60%. Można się spodziewać, że w przypadku metod dziedziczenie będzie wykorzystywane w większym stopniu.

- **Method Inheritance Factor (MIF)** – *(współczynnik dziedziczenia metod)*

  Definicja. MIF określa stopień wykorzystania dziedziczenia dla metod. Stanowi stosunek metod odziedziczonych do wszystkich obecnych.

$$MIF = \frac{\sum_{i=1}^{TC} m_i(c_i)}{\sum_{i=1}^{TC} m_a(c_a)} \quad (6)$$

  Gdzie: *TC* – liczba wszystkich klas, $m_a(c_i) = m_d(c_i) + m_i(c_i)$ - metody dostępne w klasie c_i, $m_i(c_i)$ – metody odziedziczone (i nie przesłonięte) w klasie c_i,

$m_d(c_i) = m_n(c_i) + m_o(c_i)$ - metody zdefiniowane w klasie c_i, gdzie $m_n(c_i)$ – metody nowe, $m_o(c_i)$ – metody odziedziczone i przesłonięte (zredefiniowane).

Interpretacja. Typowe wartości to 65-80%. Wartości poniżej 65% wskazują na zbyt słabe wykorzystanie mechanizmu dziedziczenia natomiast powyżej 80% na zbyt skomplikowane hierarchie dziedziczenia i nadużycie powtórnego wykorzystania kodu przez dziedziczenie.

- **Polymorphism Factor (PF)** – *(współczynnik polimorficzności)*

  Definicja. PF to procent metod pokrytych w podklasach.

$$PF = \frac{\sum_{i=1}^{TC} m_o(c_i)}{\sum_{i=1}^{TC} \left[m_n(c_i) \times dc(c_i) \right]} \tag{7}$$

  Gdzie: TC – liczba wszystkich klas, $m_n(c_i)$ – metody nowe, $m_o(c_i)$ – metody odziedziczone, $dc(c_i)$ – potomkowie klasy c_i.

  Licznik reprezentuje aktualną liczbę możliwych powiązań przez polimorficzność. Mianownik maksymalną możliwą liczbę różnych relacji polimorficzności, czyli przypadek, że każda nowa metoda klasy c_i jest przesłonięta we wszystkich jej podklasach.

  Interpretacja. Polimorfizm pozwala związać wspólnym wywołaniem metody wiele instancji klas dzięki czemu system zbudowany w ten sposób jest elastyczniejszy, a klasy w większym stopniu ukrywają szczegóły swojej implementacji. Typowe wartości wynoszą 4-18%. Zbyt duże wartości wskazują na skomplikowane hierarchie dziedziczenia, a co za tym idzie duże koszty testowania i pielęgnacji.

- **Coupling Factor (COF)** – *(współczynnik powiązań)* Współczynnik ten służy do określenia stopnia powiązań innych niż przez dziedziczenie pomiędzy klasami. **Definicja**. Jest to ilości powiązań innych niż dziedziczenie między klasami (nie ma rozróżnienia między naturą tego powiązania: asocjacje, kompozycje czy agregacje są traktowane tak samo), liczy się tylko jedno powiązanie między dana parą klas, do maksymalnej liczby powiązań nie związanych z dziedziczeniem, jakie mogłyby zaistnieć w systemie.

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} jest\_klientem(c_i, c_j) \right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} dc(c_i)} \tag{8}$$

Gdzie: *TC* – liczba wszystkich klas, $dc(c_i)$ – potomkowie klasy c_i,

$TC^2 - TC$ – maksymalna ilość możliwych powiązań w systemie o liczbie klas *TC*,

$2\times\sum_{i=1}^{TC} dc(c_i)$ - maksymalna ilość możliwych powiązań ze względu na dziedziczenie,

$$jest\_klientem(c_i, c_j) = \begin{vmatrix} 1 & jesli & c_c \Rightarrow c_s \wedge c_c \neq c_s \wedge \neg(c_c \rightarrow c_s) \\ 0 & pozostale & \end{vmatrix}$$, gdzie $c_c \Rightarrow c_s$ oznacza, że klasa c_c ma przynajmniej jedno odniesienie do klasy c_s, a $c_c \rightarrow c_s$ oznacza, że klasa c_c dziedziczy po klasie c_s.

Interpretacja. Optymalna wartość tej metryki mieści się w przedziale 5-15%. Wysoka wartość tego wskaźnika oznacza, że system jest mało elastyczny i jest trudny w modyfikacji i pielęgnacji, a zmiana w jednej części systemu oznacza konieczność zmian w innych jego częściach. Za niskie wartości mogą wskazywać, że klasy są zbyt złożone i niespójne i same realizują zbyt dużo funkcjonalności.

4.2.1.2 Weryfikacja metryk z zestawu MOOD

Abreu, Goulão i Stevensa, jak wspomniano we wstępie do rozdziału o metrykach MOOD, w roku 1995 zaprezentowali wyniki badań nad swoim zestawem metryk [20]. Na podstawie tych badań zaprezentowano wartości preferowane metryk. W pracy tej autorzy sprawdzali również zależność od rozmiaru programu przy pomocy korelacji Pearsona. Wykryto, że od rozmiaru programu zależy jedynie AHF i MIF, natomiast badano jedynie 5 programów, dlatego wymagane są badania na większej próbie. Badano również korelacje między metrykami. Okazało się, że zależne od siebie były metryki PF i COF oraz AHF i MIF. W pierwszym przypadku jako wyjaśnienie podano fakt, że jedna z metryk COF znacznie odbiegała od pozostałych. Generalnie sytuację podsumowano, że konieczne są dalsze badania na większych próbkach w celu weryfikacji uzyskanych danych.

Harrison, Counsell i Nithi przeprowadzili badania nad zestawem MOOD dla kilku systemów informatycznych [22]. Badano kilka projektów akademickich: 3 wersje systemu *Electronic Retail System*, oraz pakiet programów służących do obróbki obrazu EFOOP2, a także dziewięć systemów komercyjnych (nie podano jakich). Programy składały się z około tysiąca do 47 tysięcy linii kodu.

Dla trzech systemów ERS i pakietu EFOOP2 metryka AHF wyniosła 100%, czyli optymalną wartość. MHF dla tych czterech systemów wahało się od 6,3% do 20,4%. Wartości te w większości mieszczą się w przedziale ustalonym przez Abreu, niemniej jednak

autorzy uważają, że są zbyt niskie - informacje powinny być lepiej ukrywane. Wykryto również, że w systemach tych (oprócz pierwszej wersji ERS) nie wykorzystywano mechanizmu dziedziczenia. Dla dwóch wersji ERS i pakietu EFOOP2 MIF oraz AIF wyniosły 0%, a PF był nieokreślony (mianownik wyniósł 0 – żadna z klas nie miała swojej podklasy, więc musiało tak być zgodnie z definicją PF). COF oscylowało w granicach 0% (pierwsza wersja ERS) i 5,8% (druga i trzecia wersja ERS). Autorzy badania sugerują, że tak mała liczba powiązań pomiędzy klasami, może wskazywać że drugie i trzecie wydanie ERS oraz EFOOP2 były dobrze zaprojektowane. Natomiast wartość COF równa 0 % dla pierwszej wersji ERS sugeruje, że w systemie tym klasy komunikują się jedynie przez dziedziczenie lub duża część kodu jest zduplikowana, co w każdym przypadku jest sytuacją niepożądaną.
Dla systemów komercyjnych wartości AHF oscylowały pomiędzy 44% i 66%, a MHF 8% i 25%. Wartości AHF są zbyt niskie, atrybuty powinny być bardziej ukryte. Autorzy wysuwają wniosek, że abstrakcja implementacji jest niewystarczająca. Wartości AIF i MIF we wszystkich dziewięciu systemach komercyjnych przyjmowały wartości znacznie niższe, niż dolne progi optymalnych przedziałów zdefiniowane przez Abreu [20]. Projektanci tych systemów nie w pełni wykorzystali więc możliwości jakie daje dziedziczenie. PF dla wszystkich systemów przyjmowało stosunkowo niskie wartości (od 3% do 9%), a więc w zalecanym przedziale. Według Harrisona wartości PF mogłyby być nieco wyższe, gdyby w większym stopniu wykorzystane było dziedziczenie. Wartości COF dla wszystkich dziewięciu systemów znajdowały się w optymalnym przedziale od 4% do 18%.
Praca Harrisona pokazuje w jaki sposób można wyciągać wnioski o strukturze systemu bazując na wartościach metryk z zestawu MOOD. Autor sugeruje jednak dalsze badania w celu weryfikacji powiązań wartości metryk z zewnętrznymi atrybutami oprogramowania takim jak łatwość w utrzymaniu, czy podatność na błędy.

Mayer i Hall podjęli się krytyki podstaw teoretycznych metryk z zestawu MOOD [23].
W celu policzenia wartości metryki AIF użyto koncepcji nadpisywania atrybutów. Według autorów analizy, atrybuty klasy nie mogą być nadpisane. Można w klasie A zdefiniować jakiś atrybut x, następnie w klasie B, która dziedziczy po A również zdefiniować taki sam atrybut x. Wtedy jednak dla klasy B taki atrybut należy liczyć dwukrotnie: raz jako atrybut odziedziczony z klasy A, bo klasa A nadal używa atrybutu A.x, a raz jako atrybut nowy ponieważ klasa B używa atrybutu B.x. Mayer i Hall uważają również, że w metryce AIF powinno się uwzględniać fakt, że podklasy mają dostęp do atrybutów prywatnych nadklas poprzez ich publiczne metody.

Metryki MIF i AIF również nigdy nie przyjmują maksymalnej wartości jeden, jak to zakładali ich autorzy. Dzieje się tak dlatego, że mianownik w definicji tych dwóch metryk zlicza odpowiednio metody i atrybuty dostępne w klasach po których nikt nie dziedziczy np. klasy z samego dołu hierarchii. Oznacza to, że mianownik w tych definicjach przyjmuje wartość większą niż maksymalny możliwy poziom dziedziczenia w systemie.
Następnie metryki PF oraz MIF zakładają, że jeśli dana metoda została nadpisana, to nie nie jest zliczana do metod odziedziczonych. Takie założenie nie jest zawsze prawdziwe, gdyż metoda nadpisana może używać metody, którą nadpisuje. Ponadto PF może przyjmować wartości większe niż jeden (założono, że dostępny jest zakres od 0 do 1). Wynika to z faktu, że pominięte zostały przypadki, gdy klasy nadpisują metody spoza systemu.
Mayer i Hall stwierdzają jednak, że zestaw metryk MOOD to jeden z lepszych jakie opracowano, wymaga jednak dalszego dopracowania i weryfikacji praktycznej.

4.2.2 Metryki Chidambera i Kemerera

Zestaw sześciu metryk zaproponowane przez S.R. Chidambera i C.F. Kemerara w roku 1994 [24] popularnie zwanych metrykami CK, badających różne aspekty obiektowości: złożoność klasy, dziedziczenie, spójność, powiązania między klasami. Po raz pierwszy jeszcze w prototypowej formie metryki te zostały przedstawione na konferencji OOPSLA w roku 1991 [25]. Metryki te badają klasy i powiązania pomiędzy klasami.

4.2.2.1 Definicja i interpretacja metryk z zestawu CK

- **Weighted Methods per Class (WMC)** - *(ważona liczba metod w klasie)*
  Definicja. WMC to suma ważona metod w klasie gdzie wagę stanowi złożoność cyklomatyczna McCabe'a CC (sekcja 4.1.2). WMC jest sumą współczynników CC w danej klasie. Stosuje się także uproszczoną wersję, gdzie wszystkie metody mają taką samą wagę. Wówczas jest to po prostu suma metod w danej klasie.

$$WMC = \sum_{i=1}^{TM} CC_i \tag{9}$$

  Gdzie: *TM* – liczba metod w klasie, CC_i – złożoność cyklomatyczna metody *i.*
  Interpretacja. Zalecany próg maksymalny wartości tej metryki wynosi 20 [26]. Dr Linda Rosenberg z SATC (Software Assurance Technology Center) w NASA zaleca,

bazując na doświadczeniu wynikającym z projektów obiektowych przeprowadzonych przez NASA, aby w szczególności przeprowadzić rewizję klas o WMC większym niż 100 [27]. W projektach analizowanych przez L. Rosenberg około 60% klas miało WMC < 20. Około 36% klas zawierało się w przedziale od 20 do 100. Jedynie 4% klas miało WMC wyższe niż 100.

- **Response for a Class (RFC)** – *(odpowiedź klasy)*

  Definicja. RFC jest to liczba metod, które mogą być wykonane w odpowiedzi na wiadomość odebraną przez obiekt rozpatrywanej klasy (a więc zliczane są wszystkie metody danej klasy oraz metody innych klas, które są wywoływane bezpośrednio przez metody rozpatrywanej klasy)

  Interpretacja. Im większa wartość RFC, tym większa funkcjonalność i złożoność klasy, ale co za tym idzie większe trudności w testowaniu i weryfikacji poprawności klasy. Zbyt duże RFC może oznaczać też za dużą odpowiedzialność klasy. W badaniach L. Rosenberg z NASA 60% klas przyjmowało RFC < 40. Około 35% klas wartości od 40 do 140 i 5% powyżej 140 [27]. Autorka zaleca w szczególności rewizję klas, których RFC przekracza wartość 200.

- **Depth of Inheritance Tree (DIT)** – *(głębokość drzewa dziedziczenia)*

  Definicja. Jest to maksymalna liczba poziomów nadklas, z których rozpatrywana klasa dziedziczy.

  Interpretacja. Im większy współczynnik DIT, tym więcej metod dana klasa dziedziczy - oznacza to większy stopień specjalizacji. Drzewa dziedziczenia o dużej głębokości charakteryzują systemy o dużej złożoności, w których uczestniczy wiele spokrewnionych ze sobą klas i metod. Oznacza to wyższe koszty późniejszej pielęgnacji. Z drugiej strony dziedziczenie zwiększa stopień powtórnego użycia kodu, co ogranicza ilość zduplikowanego kodu.

- **Number of Children of Class (NOC)** – *(liczba bezpośrednich potomków klasy)*

  Definicja. NOC jest to suma bezpośrednich potomków klasy.

  Interpretacja. Za duże wartości NOC mogą wskazywać na niewłaściwe użycie mechanizmu dziedziczenia i mogą skutkować w trudnościach z testowaniem takiej klasy.

- **Coupling Between Objects (CBO)** - *(powiązania między klasami)*

Definicja. CBO jest liczbą klas, które związane są z rozpatrywaną klasą w relacjach innych niż poprzez dziedziczenie. W praktyce implementacyjnej zlicza się powiązania odśrodkowe, choć nie wynika to wprost z definicji.

Interpretacja. Niski stopień powiązań klas między sobą wskazuje na ścisłe określenie granic pomiędzy klasami i sposobu ich komunikowania się. Ponadto zwiększa stopień abstrakcji projektu, ponieważ łatwiej modyfikować fragmenty kodu w nieznacznym stopniu zależne od pozostałych klas. Wysoka wartość CBO sprawia, że klasa jest bardziej czuła na zmiany w innych częściach systemu i przez to jest trudniejsza w pielęgnacji. Powiązania między klasami powinny być utrzymywane na jak najniższym poziomie, co zmniejsza złożoność systemu. Niskie wartości CBO wspierają lepszą hermetyzację. W projektach NASA prawie 1/3 klas była samowystarczalna, czyli wartość metryki CBO = 0 (w obrębie badanej klasy brak wywołań do metod innych klas niezwiązanych z badaną klasą przez dziedziczenie) [27]. Kolejne 60% klas miało wartości CBO < 10, a pozostałe około 8% wartości większe niż 10.

- **Lack of Cohesion in Methods (LCOM)** – *(brak spójności metod)* – jest to metryka mierząca spójność metod wewnątrz klasy i w odróżnieniu od poprzednich mierzy brak cechy, a nie jej obecność. Zaprezentowano tutaj kilka wersji tej metryki. Pierwsza z nich zaprezentowana została przez Chidambera i Kemerera w roku 1991 na koferencji OPPSLA [25].

- **Definicja**. LCOM1 to różnica pomiędzy liczbą par metod odwołujących się do różnych atrybutów a liczbą par metod odwołujących się do przynajmniej jednego wspólnego atrybutu. W sposób formalny można ją wyrazić w następujący sposób:

  Rozważmy klasę *C*, która ma *n* metod m_1, m_2,…,m_n, gdzie $\{I_j\}$ zbiór atrybutów używanych przez metodę m_j. Jest n takich zbiorów. Definiujemy $P = \{(I_i, I_j) : I_i \cap I_j = 0\}$ oraz $Q = \{(I_i, I_j) : I_i \cap I_j \neq 0\}$, wtedy

$$LCOM1 = \begin{cases} P - Q & jesli \quad P > Q \\ 0 & pozostale \end{cases} \tag{10}$$

  Interpretacja. Wysoka wartość metryki może wskazywać klasy, które mają przydzieloną zbyt dużą ilość obowiązków i są kandydatem do podziału na mniejsze. Klasa o wysokiej wartości LCOM1 może być bardziej narażona na błędy, jest trudniejsza w testowaniu.

  Metryka ta spotkała się z dużą krytyką m.in. dlatego, że przyjmuje wartość zero w różnych okolicznościach, a także bardzo różne klasy o ewidentnie różnej spójności

mogą przyjmować tę samą wartość LCOM [32]. Także niektóre języki programowania jako mechanizmu dostępu do pól klasy wykorzystują własności (*properities*).

B.Henderson-Sellers, L.Constatine oraz Graham [28] zaproponowali inne wersje metryki LCOM, które zyskały popularną nazwę LCOM2 i LCOM3 (propozycja Chidambera i Kemerera określana jest jako LCOM1).

- **Definicja**. LCOM2 to procent metod, które nie mają dostępu do poszczególnych atrybutów klasy. Jeśli liczba metod lub atrybutów jest równa zero , wtedy LCOM2 jest niezdefiniowane i przyjmuje wartość zero.

$$LCOM2 = 1 - \frac{\sum_{i=1}^{TM} Ma_i}{TM \times TA} \tag{11}$$

Gdzie: *TM* – ilość metod w klasie, *TA* – ilość atrybutów w klasie, Ma_i – liczba metod odwołujących się do atrybutu a_i.

Interpretacja. Wartości tej metryki mieszczą się w zakresie [0;1], im niższa wartość metryki, tym lepiej.

- **Definicja**. LCOM3 wyraża względną liczbę metod, które nie odwołują się do poszczególnych atrybutów klasy, ale przyjmuje wartości od 0 do 2. Gdy liczba atrybutów jest równa 0, albo liczba metod równa jest 1 to metryka jest niezdefiniowana.

$$LCOM3 = \frac{TM - \frac{\sum_{i=1}^{TM} Ma_i}{TA}}{TM - 1} \tag{12}$$

Gdzie: *TM* – ilość metod w klasie, *TA* – ilość atrybutów w klasie, Ma_i – liczba metod odwołujących się do atrybutu a_i.

Interpretacja. Wartości większe niż 1 wskazują na brak spójności i prawdopodobną konieczność podziału klasy w przyszłości. Tak dzieje się np. gdy mamy atrybuty, które nie są wykorzystywane przez żadną z metod. A to z kolei oznacza, że atrybut jest w ogóle niewykorzystywany (*dead code*) albo dostęp innych klas do niego jest bezpośredni, czyli nie jest zachowana odpowiednia hermetyzacja.

Jeszcze inną propozycję do obliczenia braku spójności wewnątrz klasy zaprezentowali W.Li i S.Henry [29]. Była to kontrpropozycja do prototypu metryki LCOM zaprezentowanej przez Chidambera i Kemerara na konferencji OOPSLA[25]. W przypadku narzędzia Sonar (sekcja 5.2.3 jest jako LCOM4).

- **Definicja**. LCOM4 to liczba zbiorów rozłącznych lokalnych metod; żadne dwa zbiory nie przecinają się; każde dwie metody w tym samym zbiorze mają dostęp do przynajmniej jednego wspólnego argumentu.

  Interpretacja. Wartość metryki wynosi od 0 do N, gdzie N to liczba naturalna. Optymalna wartość metryki to jeden, dla wyższych wartości należy rozważyć podział klasy na mniejsze. Bazując na pomyśle W.Li i S.Henry'ego koncepcja metryki LCOM4 została rozwinięta przez M. Hitz'a i B.Montazeri [30] i właśnie w tej postaci jest implementowana w różnych narzędziach.

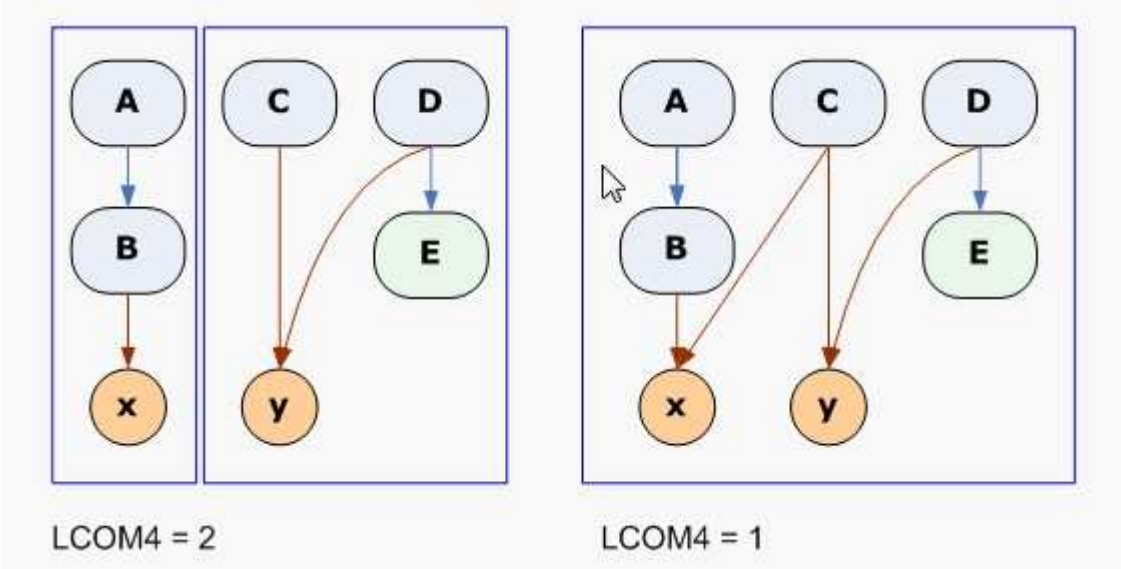
- **Definicja**. Rozważmy *C* jako badaną klasę. A_C – zbiór argumentów klasy *C* i M_C zbiór metod klasy *C*. Stwórzmy graf nieskierowany $G_C(V,E)$, taki, że $V=M_C$ oraz $E = \{< m,n >\in V \times V \mid \exists i \in A_C : (m \quad ma\_dostęp\_do \quad i) \wedge (n \quad ma\_dostęp\_do \quad i)\}$,

  wtedy:

  LCOM4 będzie liczbą komponentów grafu G_C , które powstały w wyniku połączeń w grafie G_C $(1 \le LCOM4 \le M_C)$. W tym wypadku już nie każda para metod w podzbiorze (komponencie) musi mieć dostęp do wspólnego argumentu.

  Interpretacja. Optymalna wartość metryki wynosi jeden. Dla wyższych wartości należy rozważyć podział klasy na mniejsze.

  Na rysunku 1 pokazano przykład niespójnej LCOM4 = 2 i spójnej klasy LCOM4 = 1. A,B,C,D,E to metody w klasie, a x, y to atrybuty.



Rysunek 3. Przykłady spójnej i niespójnej klasy według LCOM4 za [31]

4.2.2.2 Weryfikacja metryk CK i pochodnych

V.Basili, L.Briand i W.Melo podjęli próbę oceny metryk z zestawu CK jako wskaźników jakości kodu [32]. Przeprowadzili oni eksperyment na Uniwersytecie Maryland. Badano 8 projektów – w sumie składających się ze 180 klas – napisanych przez studentów. Jako zmienne niezależne wzięto metryki CK, a jako zależne liczbę źle wykonanych klas i liczbę usterek wykrytych w czasie testów. Najważniejsze wnioski jakie płyną z pracy to:

- Metryki CK są stosunkowo niezależne. Najwyższy współczynnik korelacji osiągnięto dla metryk RFC i CBO ($R^2=0,38$).
- Na podstawie regresji logistycznej wielu zmiennych wykazano, że metryki RFC, CBO, DIT i WMC są silnie skorelowane z liczbą błędów. Tak więc metryki te dobrze nadają się do przewidywania błędów w systemie.
- Metryka LCOM nie umożliwia wykrywania źle napisanych klas. Główne wady tej metryki zasugerowane przez autorów: metryka przyjmuje wartość zero w różnych okolicznościach, a także dla klas o różnym stopniu spójności podaje te same wartości metryki.
- Badania pokazały również, że metryki z zestawu CK lepiej nadają się do przewidywania błędów niż metryki rozmiaru (np. liczba deklaracji i wywołań funkcji, maksymalny poziom zagnieżdżenia instrukcji w klasie).
- Niepełne wykorzystanie mechanizmu dziedziczenia zostało potwierdzone niskimi wartościami metryk DIT oraz NOC.

Autorzy zwracają uwagę, że projekty realizowane przez studentów nie były duże i wykazywały się niską złożonością. Są to czynniki, które mogły mieć wpływ na weryfikację metryk.

W roku 1998 Chidamber, Darcy i Kemerer opisali wyniki zastosowania metryk z zestawu CK do trzech komercyjnych systemów oprogramowania finansowego opracowanych przez europejski bank [33].

Metryki NOC i DIT przyjmowały niskie wartości, z czego wysnuto wniosek, że dziedziczenie nie było wykorzystane w wystarczającym stopniu. Stwierdzono również silną korelację pomiędzy metrykami WMC, CBO i RFC. Wynik ten jest sprzeczny z uzyskanym przez Basili'ego i współpracowników [32], gdzie zmienne te okazały się być stosunkowo niezależne. Aspekt ten wymaga dalszych badań. Autorzy sugerują, że powodem silnej

korelacji w tych konkretnie badanych systemach mogą być wysokie wartości metryki WMC, które pociągają za sobą wysokie wartości metryk CBO i RFC.

Opracowano również model regresji liniowej dla badanych systemów. Za zmienną zależną przyjmowano kolejno:

- produktywność – liczba linii kodu w klasie do liczby godzin potrzebnych do jej zaimplementowania,
- koszt ponownego użycia – liczba godzin potrzebna do przystosowania klasy do nowego systemu,
- koszt projektu - liczba godzin potrzebna do napisania klasy.

Najlepszymi zmiennymi niezależnymi okazały się metryki zdefiniowane jako wysoka wartość CBO i wysoka wartość LCOM (wartość zero zmienna niezależna przyjmuje jeśli wartość metryki rozpatrywanej klasy znajduje się w 80% dolnych wartości tej metryki w systemie i jeden w przeciwnym przypadku). Znaleziono więc korelację pomiędzy wysokim wartościami metryk CBO i LCOM, a takim zewnętrznymi aspektami oprogramowania jak niższa produktywność, większy koszt ponownego użycia, czy większy koszt zaprojektowania klasy. Autorzy zastrzegają jednak, że dla innych systemów można uzyskać inne modele regresji liniowej (z pewnością inne będą progi 80%/20% dla zmiennych niezależnych) i zalecają przeprowadzenie podobnej analizy w każdej organizacji.

L.Etzkorn, C.Davis, and W.Li przeprowadzili analizę, która z metryk LCOM1, czy LCOM4 lepiej mierzy spójność klasy[25]. Zestawiając przy użyciu regresji liniowej otrzymane wyniki metryk z danymi przygotowanymi na temat spójności klas przez siedmiu ekspertów wykazali, że znacznie lepiej spójność mierzy LCOM4 (współczynnik korelacji wyniósł $R^2=0,66$). Dodatkowo badania obejmowały również sprawdzenie, w jakiej konfiguracji tych metryk osiąga się najlepsze wartości, gdzie zmiennymi było uwzględnianie bądź nie funkcji konstruktora klasy i metod dziedziczenia. Z ich badań wynikło, że dokładniejszy rezultat otrzymuje się jeśli w obliczeniach nie uwzględnia się dziedziczenia oraz uwzględnia się metodę konstruktora klasy.

Mayer i Hall poddali pod analizę podstawy teoretyczne metryk z zestawu CK [34]. Według autorów metryka WMC nie powinna odnosić się do dwóch atrybutów klasy: liczby metod i złożoności metod. Dodanie do definicji wag metod niepotrzebnie komplikuje ich interpretację. Skrytykowano również metrykę DIT za to, że nie uwzględnia liczby odziedziczonych metod, ani liczby przodków danej klasy (jeśli chodzi o języka Java, to ten drugi argument akurat nie ma znaczenia, ponieważ w Javie nie jest używane wielodziedziczenie). Generalnie jednak metryka odnosząca się do dziedziczenia powinna

uwzględniać ten aspekt ponieważ nadmierne użycie tego mechanizmu znacznie zwiększa złożoność. Metryka NOC została zdefiniowana jako liczba bezpośrednich podklas klasy. Według Mayer'a i Hall'a powinno się zliczać wszystkich potomków klasy, nie tylko bezpośrednich. Wtedy zgodnie z intencją Chidambera i Kemerera metryka ta będzie lepiej mierzyła stopień ponownego użycia kodu w systemie.
Skrytykowano także niejasną definicję metryki CBO. Nie zostało określone, czy metryka ta ma mierzyć powiązania jednostronne czy dwustronne. Została też odnotowana nieścisłość w definicji metryki RFC. Ze względów praktycznych zliczane są jedynie metody w bezpośrednio badanej klasie natomiast w praktyce wykonane mogą być metody z kolejnych poziomów zagnieżdżenia. Autorzy zgadzają się również z wcześniejszymi opracowaniami przywoływanymi w tej pracy [28, 29], że metryka LCOM1 jest źle zdefiniowana.
Wreszcie Chidamber i Kemerer pominęli przy definiowaniu swojego zestawu metryk tak ważne aspekty związane z programowaniem obiektowym jak: polimorfizm, czy enkapsulacja.

4.2.3 Metryki Martina

Kolejny zestaw 5 metryk obiektowych został zaproponowany przez Roberta Martina w roku 1994 [35]. Martin skupił się na zagadnieniu zależności pomiędzy pakietami, a nie na badaniu poszczególnych atrybutów projektu obiektowego jak to jest w przypadku metryk z zestawu MOOD oraz CK. Z omówionymi tutaj metrykami wiążą się dwie zasady, które Martin sformułował w roku 2000 [6]. Jest to zasada stabilne abstrakcyjne SAP oraz zasada stabilnych zależności SDP, obydwie zostały opisane we sekcji 3.3.3. Do innych zasad zaproponowanych przez Martina też można stworzyć metryki – sekcja 4.2.4 jednak w literaturze i w narzędziach implementujących metryki, do metryk Martina zalicza się jedynie 5 zaprezentowanych w tym rozdziale metryk.

- **Efferent coupling (Ce)** – *(powiązania do zewnątrz - odśrodkowe)*

  Definicja – Ce to liczba klas z rozpatrywanego pakietu, które zależą od klas w innych pakietach. Mierzy zatem podatność rozpatrywanego pakietu na zmiany w pakietach, od których zależy.

  Interpretacja. Wysokie wartości tej metryki wskazują na niestabilność pakietu – duża zależność od zewnętrznych klas. Z drugiej strony w takim przypadku rozpatrywany pakiet nie jest odpowiedzialny przed innymi komponentami, co oznacza, że zmiany, które w nim zachodzą nie są propagowane.

Preferowane wartości to od 0 do 20 (RefactorIT sekcja 5.2.2). Wartości wyższe powodują problemy z pielęgnowalnością i rozwojem kodu. Typowym przykładem o wysokiej wartości Ce są elementy GUI [21].

- **Afferent coupling (Ca)** – *(powiązania do wewnątrz - dośrodkowe)*

  Definicja. Ca to liczba klas spoza rozpatrywanego pakietu, które zależą od klas w rozpatrywanym pakiecie.

  Interpretacja. Wysoka wartość tej metryki sugeruje w niejawny sposób dużą stabilność komponentu, ale także dużą odpowiedzialność względem zależnych typów. Dopuszczalne są znacznie większe wartości niż w przypadku Ce (RefactorIT sekcja 5.2.2 dopuszcza wartości do 500) , co wynika z trudności nad kontrolą nad pakietami, które zależą od analizowanego pakietu. Przykładem klasy o dużej wartości Ca jest kontroler z wzorca MVC [21].

- **Instability (I)** – *(niestabilność)*

  Definicja. I jest to względna podatność na zmiany. Metryka ta jest zdefiniowana za pomocą Ce i Ca i jej wartość wynosi:

$$I = \frac{Ce}{Ce + Ca} \tag{13}$$

  Interpretacja. Pakiety powinny się dzielić na dwie grupy [21]:

  - stabilne o wartości metryki I bliskiej 0, które ponoszą dużą odpowiedzialność wobec innych komponentów.
  - niestabilne o wartości metryki I bliskiej 1, które w dużej mierze zależą od innych komponentów

  Podane powyżej przedziały to wartości optymalne, które powinny być otrzymywane w projekcie. Należy unikać komponentów o pośredniej stabilności. W praktyce jednak około połowa pakietów przyjmuje wartości pośrednie [35]. W takim wypadku sugeruje się, aby metryka znajdowała się możliwie blisko ciągu głównego (patrz metryka Dn).

- **Abstractness (A)** – *(abstrakcyjność)* – mierzy stopień abstrakcji pakietu.

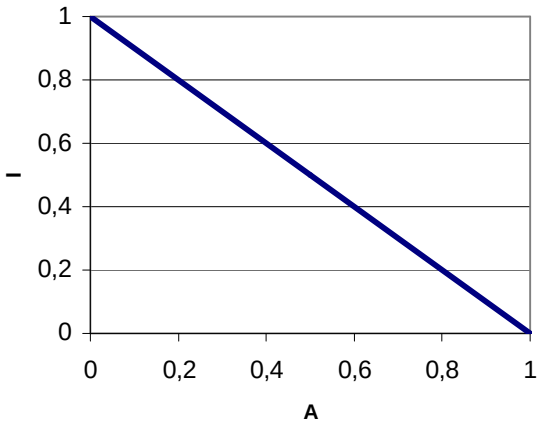
  Definicja. A to stosunek jednostek abstrakcyjnych w pakiecie (w języku Java są to interfejsy i klasy abstrakcyjne) do wszystkich jednostek w rozpatrywanym pakiecie.

  Interpretacja. A powinno przyjmować wartości skrajne bliskie 0 lub 1. Pakiety, które są stabilne - metryka I bliska zeru, czyli w niskim stopniu zależne od innych pakietów, ale ponoszące dużą odpowiedzialność wobec innych pakietów, powinny być jednocześnie abstrakcyjne. Klasy abstrakcyjne są bardzo dobrymi kandydatami na

klasy stabilne, dlatego, że przy użyciu dziedziczenia można implementować przez nie różne funkcjonalności, a z drugiej strony nie trzeba modyfikować tych klas. Ich niezmienność zapewnia, że są stabilne.

Z kolei pakiety bardzo niestabilne – metryka I bliska jeden, powinny się składać z klas konkretnych. Zależy nam na tym, żeby ograniczyć zależności od typów niestabilnych. Użycie klas abstrakcyjnych jako typów niestabilnych warunkowałoby nam zwiększenie zależności od nich, ponieważ w przypadku klas abstrakcyjnych potrzebujemy klas, które będą po nich dziedziczyły (nie da się tworzyć instancji klas abstrakcyjnych).

- **Normalized distance from main sequence (Dn**) – *(znormalizowana odległość od ciągu głównego).* R. Martin wprowadził również pojęcie ciągu głównego na wykresie A(I). Równanie ciągu głównego ma postać A + I = 1.



Rysunek 4. Wykres ciągu głównego. Według Martina pakiety powinny mieć takie wartość metryk I oraz A aby koncentrowały się głównie na końcach ciągu głównego.

Najbardziej pożądaną sytuacją jest gdy pakiety znajdują się na końcach ciągu głównego, a więc albo są konkretne i niestabilne albo stabilne i abstrakcyjne. W praktyce jednak około połowa pakietów przyjmuje wartości pośrednie [35] . Martin wysnuł hipotezę, że w takim wypadku najlepiej aby pakiety miały takie wartości metryk I oraz A aby znajdowały się możliwie blisko ciągu głównego. Wtedy abstrakcyjność będzie zbalansowana ze stabilnością takiego pakietu, a liczba klas abstrakcyjnych i konkretnych będzie w dobrej proporcji do powiązań odśrodkowych i dośrodkowych tego pakietu.

Definicja. Metryka jest obliczana w następujący sposób:

$$Dn = |A + I - 1| \tag{14}$$

Dzięki normalizacji wartość metryki ma zakres od [0;1], bez normalizacji sama odległość od ciągu głównego ma zakres [0;~0,707]. Przyjęło się używanie wersji znormalizowanej.

Interpretacja. Wartość Dn powinna być jak najniższa, tak, aby komponenty znajdowały się możliwie blisko ciągu głównego. Rozważmy skrajne - niepożądane przypadki:

- A=0 i I=0, czyli pakiet, który jest skrajnie stabilny i konkretny. Sytuacja bardzo niepożądana, ponieważ pakiet jest bardzo sztywny i nie można go rozszerzać
- A=1 i I=1 – taka sytuacja właściwie jest niemożliwa ponieważ pakiet całkowicie abstrakcyjny musi mieć jakieś powiązania do zewnątrz, żeby mogła powstać instancja implementująca funkcjonalności zdefiniowane w klasach abstrakcyjnych lub interfejsach znajdujących się w tym pakiecie.

4.2.4 Metryki powiązane z innymi prawami i zasadami programowania obiektowego

W rozdziale tym zaprezentowano metryki powiązane z prawami i zasadami programowania zorientowanego obiektowo, które nie zostały ujęte w zestawie MOOD, CK albo Martina. Prawa i zasady powiązane z tymi metrykami zostały szczegółowo omówione w sekcji 3.3. Tam też znaleźć można zalety stosowania się do tych praw jak i również konsekwencje ich łamania. Automatycznie więc znaleźć tam można szczegółową interpretację poniższych metryk, która tutaj nie będzie powtarzana.

- **Violations of Demeter (VOD)** – *(liczba naruszeń prawa Demeter)* (prawo Demeter opisane w sekcji 3.3) .

  Definicja metryki. Jest to liczba naruszeń prawa Demeter w rozpatrywanej części systemu.

  Interpretacja metryki. Wartość optymalna to 0.
- **TANGLED** - Metryka, która powstała w oparciu o zasadę acyklicznych zależności ADP (sekcja 3.3.3). Wyliczana jest w oparciu o graf zależności pomiędzy pakietami lub zbiorami pakietów. W tym celu wykorzystywana jest zasada minimalnej informacji zwrotnej MFS (ang. *minimum feedback set*). Jeśli założymy, że każda

krawędź grafu ma swoją wagę wyrażoną liczbą powiązań to MFS pozwala nam wybrać krawędzie, których rozbicie powinno być najłatwiejsze. Splot (*ang. tangle)* w tym wypadku, to na grafie zależności taki graf o przynajmniej dwóch węzłach, w którym każdy z węzłów jest osiągalny z drugiego węzła.

Definicja. Tangled wyliczana jest jako stosunek MFS do wagi wszystkich krawędzi w grafie (dla pakietu lub zbioru pakietów).

Interpretacja. Zaleca się aby eliminować wszystkie cykle, stąd pożądana wartość metryki wynosi 0.

- **Cyclic dependency CYC** *(zależności cykliczne)*. Kolejna metryka opierająca się o zasadę acyklicznych zależności ADP (sekcja 3.3.3) i jest wyliczana dla pakietu.

  Definicja. CYC to liczba cykli w jakie zaangażowany jest pakiet.

  Interpretacja. Zaleca się aby eliminować wszystkie cykle, stąd pożądana wartość metryki wynosi 0. W im więcej cykli zaangażowany jest pakiet, tym problemy wynikające z naruszenia zasady ADP są bardziej nasilone.

- **Dependency inversion principle metric DIPM** *(metryka zasady odwróconych zależności).* Odwołuje się do zasady DIP z zestawu SOLID (sekcja 3.3.2). Obliczana dla klas.

  Definicja. DIPM to stosunek liczby zależności, które mają za cel klasę abstrakcyjną albo interfejs, do wszystkich zależności od innych klas.

  Interpretacja. Generalnie zalecane są wysokie wartości zgodnie z zasadą DIP. Rekomendacja programu RefactorIT (sekcja 5.2.2) mówi o tym, że preferowane są wartości od 0,3 do 1. Tak duże rozpiętości wartości metryki DIPM wynikają z tego, że czasem konieczne jest stworzenie konkretnych instancji, a następnie modułów, które będą od nich zależały. Zależności od klasy konkretnej, która nie będzie się bardzo zmieniać nie są szkodliwe.

- **Encapsulation principle EP** (*zasada hermetyczności).* Metryka ta odnosi się do jednego z paradygmatów programowania obiektowego – hermetyczności (opis w sekcji 3.2). Metryka tutaj zaprezentowana dotyczy hermetyczności pakietów.

  Definicja. EP dla pakietów to stosunek klas z rozpatrywanego pakietu, które są używane na zewnątrz pakietu, do wszystkich klas w pakiecie.

  Interpretacja. Preferowane są niskie wartości, bo właśnie takie wyrażają ideę hermetyczności. Natomiast RefactorIT (sekcja 5.2.2) rekomenduje dość szeroki zakres dopuszczalnych wartości od 0 do 0,6.

4.3 Metryki duplikatów i testów jednostkowych

4.3.1 Zduplikowany kod

Zagadnienie zduplikowanego kodu zostało omówione w sekcji 3.4. Opisane zostały tam sposoby wykrywania duplikatów.

Jako metrykę można zaproponować procent zduplikowanego kodu – DC (Duplicated Code).

Definicja metryki:

$$DC = \frac{DLOC}{TLOC} \tag{15}$$

Gdzie: DLOC – liczba zduplikowanych linii kodu, TLOC – całkowita liczba linii kodu w systemie.

Interpretacja metryki. Wartość metryki powinna być jak najbliższa zeru. Usuwanie duplikatów niesie z sobą następujące korzyści [36]:

- zmniejsza rozmiary kodu, zwiększa jego czytelność, przez co ułatwia zrozumienie i przyspiesza refaktoryzację,
- zmniejsza liczbę błędów w systemie, ponieważ usuwane są kopie błędów powstałe w wyniku powielania źle zaimplementowanej struktury kodu.

4.3.2 Pokrycie testami jednostkowymi

Testy jednostkowe to testy pojedynczej instancji klasy w różnych przypadkach testowych. W sposób bardziej szczegółowy zagadnienie zostały opisane w sekcji 3.4.

Definicja metryki. Zwykle przedstawia się dwie metryki związane z pokryciem testami

- Line Covarage (LC) – pokrycie testami linii kodu – jest to procent linii kodu, które zostały wykonane w trakcie przeprowadzania testów.
- Branch Covarage (BC) – pokrycie testami gałęzi – jest to procent gałęzi kodu, które zostały wykonane w trakcie przeprowadzania testów.

Obydwie metryki można wyliczać dla klas, pakietów jak i całego systemu.

Interpretacja. Przy pomocy narzędzi wyliczających pokrycie testami można osiągnąć wskaźnik pokrycia testami rzędu 90% i to jest wartość metryki do której należy zmierzać [37]. 100% jest niemożliwe, dlatego, że część kodu ma prawo nie wykonać się nigdy, np.:

- Kod, który wykonuje się tylko pod konkretnym systemem operacyjnym.

- Metody w klasach niepublicznych dziedziczące po klasie typu `interface`, które muszą być zaimplementowane, a nie mają szansy się nigdy wykonać.
- Bloki kodu, które obsługują błędy maszyny wirtualnej.

Według Elliote Rusty Harold z IBM [37] testować należy nawet najprostsze metody typu `get()` lub `set()`gdyż bardzo trudno ocenić jest kiedy metoda jest zbyt prosta, żeby mogła wykonać się błędnie. Napisanie prostego testu wiele nie kosztuje, a daje pewność poprawnego wykonania się metody. Wysoki wskaźnik pokrycia testami daje więc nam większą pewność poprawnie działającego kodu.

4.4 Metryki błędów, czytelności i dobrych praktyk języka Java

Wraz z rozwojem języków obiektowych zaczęły pojawiać się narzędzia, które wspierają pisanie kodu dobrego jakościowo. Jeśli chodzi o język Java, to w pewną osobną kategorię wpisują się trzy narzędzia Checkstyle, FindBugs i PMD. Narzędzia te – szczegółowo opisane w rozdziale 5.2 - implementują część z przedstawionych w tej pracy metryk, niemniej jednak głównie testują kod pod kątem specyficznych przypadków błędów, naruszeń stylu lub standardu języka Java. Każde z tych narzędzi zawiera po kilkaset różnego rodzaju testów. Ponieważ są to narzędzia *opensource* stosowane obecnie na szeroką skalę również w firmach informatycznych zdecydowano się na wykorzystanie ich w niniejszej pracy. Narzędzia te stanowią więc ważne wsparcie statycznej analizy oprogramowania, nie koniecznie jednak implementują metryki.

Wspomniane narzędzia służą do bieżącej pracy przy rozwoju kodu, skierowane są więc głównie do developerów. Należy tutaj podkreślić, że każdy negatywny wynik poszczególnego przypadku testowego należy traktować osobno. Dokumentacja do FindBugs mówi, że nawet do 50% zgłaszanych błędów może zostać uznane przez programistę jako niekoniecznych do refaktoryzacji.

Stwierdzono jednak, że na potrzeby tej pracy jako ciekawą metrykę jakości kodu można uznać liczbę naruszeń dla danego zestawu reguł. Opierając się o artykuł napisany przez Oliviera Gaudina jednego z twórców aplikacji Sonar[38] zdecydowano się na skonstruowanie 3 metryk na bazie każdego ze wspomnianych na początku tego rozdziału narzędzi. Obecnie Checkstyle, Findbugs i PMD w dość dużym stopniu przenikają się zakresem swoich testów niemniej jednak według Gaudina każde z tych narzędzi ma swoją specjalizację wynikającą z pierwotnego zamiaru autorów. Zdecydowano się aby stworzyć trzy metryki, dla których

został subiektywnie dobrany zestaw testów w oparciu o kategorie proponowane przez twórców narzędzi:

- **Potential Bugs (PB)** – (*metryka potencjalnych błędów*). Wyliczana w oparciu o Findbugs (sekcja 5.2.8).

  Definicja. Jest to liczba naruszeń reguł zdefiniowanych przez program Findbugs. Wykorzystane zostały wszystkie kategorie reguł zdefiniowanych przez Findbugs: *Correctness Bug, Multithreaded correctness, Malicious code vulnerability, Security, Dodgy code,* za wyjątkiem *Bad practise* i *Performance* (objaśnienie kategorii wraz z przykładami testów zostało zaprezentowane w rozdziale dotyczącym Findbugs). Zdecydowano się na usunięcie kategorii *Bad practise*, gdyż testy tutaj zawarte przynależą do kolejnej metryki zaproponowanej w tym rozdziale – GP, oraz *Performance* - testy tutaj zdefiniowane pomagają napisać bardziej wydajny kod natomiast, to, że coś jest mniej wydajne nie oznacza, że jest błędne.

  Interpretacja. Pożądane są jak najniższe wartości. Każde naruszenie powinno być jednak w miarę możliwości analizowane osobno.

- **Good Practise (GP)** – (*metryka przestrzegania dobrych praktyk programowania obiektowego oraz stosowania się do standardu języka zalecanego przez producenta*). Wyliczana w oparciu o PMD (sekcja 5.2.10).

  Definicja. Jest to liczba naruszeń wybranych kategorii testów w oparciu o PMD. Wykorzystana zostały następujące kategorie reguł zdefiniowanych w PMD: *Basic, Clone Implementation, Controversial, Design, Empty code, Finalizer, Import Statements, Java Logging, JUnit, Migration, Security Code Guidelines Optimization, Strict Exceptions, String and StringBuffer, Unnecessary* – opis kategorii wraz z przykładami znajduje się w sekcji dotyczącej PMD (wybrane tutaj kategorie zostały oznaczone pogrubioną czcionką w opisie narzędzia).

  Interpretacja. Pożądane są jak najniższe wartości. Każde naruszenie powinno być jednak w miarę możliwości analizowane osobno.

- **Readability (R)** – metryka czytelności, czyli stosowania się do konwencji i stylu pisania w języku Java według standardu producenta języka – Sun. Metryka wyliczana w oparciu o Checkstyle (sekcja 5.2.9).

  Dość powszechnie zdarza się, że przestrzeganie stylu i konwencji zalecanego przez producentów oprogramowania jest marginalizowana przez programistów. To dlatego, że konwencja pisania kodu i styl często nie mają bezpośredniego przełożenia na stabilność, czy wydajność pisanego oprogramowania. Niemniej jednak stosowanie się

do dobrych zasad w tym zakresie poprawia czytelność kodu, ułatwia zrozumienie, przez co przyczynia się do zwiększenia wydajności pracy zespołu.

Definicja. Zestaw kategorii testów użytych do wyliczenia tej metryki zaznaczono pogrubioną czcionką w kolejnym rozdziale, gdzie opisany jest Checkstyle.

Interpretacja. Pożądane są jak najniższe wartości. Każde naruszenie powinno być jednak w miarę możliwości analizowane osobno.

4.5 Podsumowanie

Rozdział 4 zawiera szczegółowe opisy metryk statycznej analizy oprogramowania, które znaleziono w literaturze a także dostępnych narzędziach wspierających metryki dla języka Java. Metryki rozmiaru i złożoności zaczerpnięto głównie z dostępnych narzędzi. Oprócz metryki CC żadna z tych metryk nie doczekała się publikacji naukowej. Także podane tutaj preferowane wartości metryk pochodzą z rekomendacji narzędzi i wynikają zapewne z doświadczeń twórców narzędzi. Najlepiej opisane w literaturze, a także najlepiej zweryfikowane w praktyce są metryki powiązane z mechanizmami i cechami obiektowymi – metryki z zestawu MOOD i CK. Oba zestawy częściowo pokrywają się jeśli chodzi o zakres pomiarów związanych z dziedziczeniem i powiązaniami (np. metryka COF z MOOD i CBO z CK), ale są też nawzajem komplementarne – zestaw MOOD porusza aspekt hermetyzacji i polimorfizmu, a zestaw CK spójności. Metryki te były kilkukrotnie redefiniowane i usprawniane przez samych autorów pomimo to jednak spotkały się w swojej finalnej wersji z konstruktywną krytyką ze strony Mayer'a [23, 34] i wymagają dalszych usprawnień. Pomimo nieścisłości w definicji praktyczna weryfikacja obydwu zestawów pokazuje jednak, że obydwa zestawy stanowią dobre wskaźniki zewnętrznych cech jakości oprogramowania takich jak niezawodność [26, 32, 33]. Jeśli chodzi o organizację systemów na poziomie pakietów to wyłączność ma Robert Martin, który zaproponował metryki mierzące stabilność pakietów. Metryki te stanowią implementację zasad, które zaproponował: SDP, SAP. Dla kolejnych zasad: ADP, DIP znaleziono odpowiednio metryki Tangled, CYC oraz DIPM w narzędziach Stan i RefactorIT (sekcja 5.2). Niestety nie znaleziono dla metryk Martina, ani dla jego praw, prób weryfikacji jego propozycji w praktyce. Omówiono także metryki związane z duplikatami oraz testami jednostkowymi. W ostatnim podrozdziale zaproponowane zostały metryki dobrych praktyk, czytelności i potencjalnych błędów języka Java w oparciu o narzędzia dedykowane do badania kodu właśnie w tych aspektach. Dokonano także wyboru kategorii testów dostępnych w tych narzędziach, tak aby

proponowane metryki PB, GP oraz R jak najlepiej mierzyły zaproponowane obszary. W Tabeli 1 zestawiono wszystkie metryki opisane w tym rozdziale.

Tabela 1. Zestawienie metryk statycznej analizy oprogramowania. Objaśnienia kolumn: Grupa – odnosi się do grupy metryk zdefiniowanych w pracy; skrót – oznaczenie metryki; poziom – jakiego poziomu abstrakcji dotyczy metryka: m – metoda, k – klasa, p – pakiet, s - system; obszar – odnosi się do szczegółowego obszaru pomiaru metryki (nawiązanie do rozdziału 3); nor. – normalizacja: N – nie, T – tak; wartości zalecane - tam gdzie znaleziono sugerowane wartości metryk podano ich wartości, w większości przypadków jednak w literaturze mówi się o wysokich (↑) , bądź niskich (↓) wartościach metryk, użyto tutaj też skrótu komp. od kompromis, gdyż są przypadki gdzie zarówno wartości niskie jak i wyższe mają zalety; odwołanie – zawiera odwołanie do rozdziału w pracy opisującego daną metrykę.

Grupa	Skrót	Poziom	Definicja	Obszar	Nor.	Wartości zalecane	Odwołanie
Zliczenia	LOC	-	Liczba linii kodu	Rozmiar, złożoność	N	m. < 60 k. < 300	4.1.1
Zliczenia	NOF	k.	Liczba pól	Rozmiar, złożoność	N	< 20	4.1.1
Zliczenia	NOM	k.	Liczba metod	Rozmiar, złożoność	N	< 50	4.1.1
Zliczenia	TLC	p.	Liczba klas, bez klas wewnętrznych	Rozmiar, złożoność	N	< 40	4.1.1
Powiązania	CC	m.	Złożoność cyklomatyczna	Złożoność	N	< 10	4.1.2
Powiązania	NPC	m.	Liczba możliwych do wykonania się ścieżek	Złożoność	N	< 200	4.1.2
Powiązania	FAT	k. p.	Liczba zależności między metodami i polami Liczba zależności pomiędzy klasami	Złożoność	N N	↓ ↓	4.1.2
Powiązania	DAC	k.	Liczba obiektów innych klas, które zawiera w sobie rozpatrywana klasa	Złożoność	N	< 3	4.1.2
Powiązania	CFOC	k.	Liczba innych klas, które odwołują się do rozpatrywanej	Złożoność	N	< 20	4.1.2
Powiązania	ACD	k. p.	Średnia, względna zależność komponentu od innych komponentów	Złożoność	N	↓	4.1.2
Obiektowe/ MOOD	AHF	s.	Stopień hermetyzacji atrybutów w klasach	Hermetyzacja, OCP	T	100%	4.2.1
	MHF	s.	Stopień hermetyzacji metod w klasach	Hermetyzacja, OCP	T	10-25%	
	AIF	s.	Procent odziedziczonych atrybutów w klasach	Dziedziczenie	T	50-60%	
	MIF	s.	Procent odziedziczonych metod w klasach	Dziedziczenie	T	65-80%	
	PF	s.	Procent metod pokrytych w podklasach	Polimorfizm	T	4-18%	
	COF	s.	Stopień powiązań między klasami	Złożoność	T	5-15%	
	WMC	k.	Ważona liczba metod w klasie	Złożoność	N	↓	4.2.2
	RFC	k.	Liczba metod, które mogą być	Złożoność	N	↓	

			wykonane w odpowiedzi na wiadomość wysłaną do tej klasy				
Obiektowe/ CK i pochodne	DIT	k.	Głębokość drzewa dziedziczenia	Dziedziczenie	N	↓, komp.	
	NOC	k.	Liczba podklas	Dziedziczenie	N	↓, komp.	
	CBO	k.	Liczba powiązań danej klasy z innymi klasami	Złożoność	N	↓	
	LCOM1	k.	Brak spójności metod	Spójność, SRP	N	↓	
	LCOM2	k.	Brak spójności metod	Spójność, SRP	Y	↓	
	LCOM3	k.	Brak spójności metod	Spójność, SRP	N	<1	
	LCOM4	k.	Brak spójności metod	Spójność, SRP	N	0	
Obiektowe/ Martina	Ce	p.	Powiązania do zewnątrz	Złożoność	N	↓	4.2.3
	Ca	p.	Powiązania do wewnątrz	Złożoność	N	↓	
	I	p.	Względna podatność na zmiany	Niestabilność	Y	↑ lub ↓	
	A	p.	Stopień abstrakcji pakietu	Abstrakcyjność, SAP	Y	↑ lub ↓	
	Dn	p.	Znormalizowana odległość od ciągu głównego	SDP, SAP	Y	↓	
Obiektowe/ Metryki zasad i praw	VOD	k.	Naruszenia prawa Demeter	Prawo Demeter	N	0	4.2.4
	Tangled	p.s.	Stosunek wagi krawędzi MFS do wagi wszystkich krawędzi w grafie zależności	ADP	N	0	
	CYC	p.	Liczba cykli, w których bierze udział pakiet	ADP	T	0	
	DIPM	k.	Stosunek zależności od klas abstrakcyjnych i interfejsów do wszystkich zależności	DIP	T	↑	
	EP	p.	Stosunek klas mających powiązania na zewnątrz pakietu do wszystkich klas	Hermetyczność	T	↓	
Zduplikowany kod	DC	-	Procent zduplikowanego kodu w systemie	Unikanie duplikatów	T	0%	4.3.1
Pokrycie testami	LC	-	Pokrycie testami linii kodu	Testy jednostkowe	T	90%	4.3.2
	BC	-	Pokrycie testami gałęzi	Testy jednostkowe	T	↑	4.3.2
Metryki potencjalnych błędów, dobrych praktyk oraz czytelności języka Java	PB	k.p.s.	Liczba potencjalnych błędów według zbioru reguł Findbugs	Poprawne zrozumienie działania języka	N	↓	4.4
	GP	k.p.s.	Liczba potencjalnych naruszeń dobrych praktyk	Stosowanie się do dobrych praktyk programowania	N	↓	
	R	k.p.s.	Liczba naruszeń utrudniających czytelność kodu	Czytelność kodu	N	↓	

5 Narzędzia statycznej analizy oprogramowania

W rozdziale tym zaprezentowano przegląd narzędzi statycznej analizy oprogramowania. W pierwszym podrozdziale omówiono wykorzystane środowiska programowania wspierające analizę statyczną. Środowiska te poprzez możliwość dodawania wtyczek, umożliwiają analizowanie wybranych projektów przy pomocy narzędzi wspierających metryki statycznej analizy oprogramowania. Narzędzia wspierające metryki opisane zostały w drugim podrozdziale.

5.1 Środowiska programowania wspierające analizę statyczną oprogramowania

W celu przeprowadzanie analizy kodu podobnie jak programiści, którzy są tak naprawdę pierwszym inżynierami jakości, trzeba się zdecydować na pracę w konkretnych środowiskach. W przypadku języka Java wygodnie jest pracować używając Eclipse'a [39] ze względu na dużą dostępność wtyczek. Do całościowego budowania projektów można używać Maven'a [41]. Z tych dwóch środowisk korzystają też twórcy GridSpace2.

5.1.1 Eclipse

Eclipse [39] to platforma programistyczna obejmujące zintegrowane środowisko programistyczne (IDE) i rozszerzalny system wtyczek. Może być używany do pisania aplikacji w języku Java, a także dzięki rozbudowanemu systemowi wtyczek w innych

językach takich jak Ada, C, C++, COBOL, Haskell, Perl, PHP, Pyton, R, Ruby, Scala, Clojure, Groomy, Android i Scheme. Jeśli chodzi o język Java to środowisko programistyczne zawiera Java Development Tools (JDT) – zestaw programów użytkowych integrujących się z platformą. Aplikacje te korzystają z mechanizmu rozszerzenia funkcjonalności dostępnego w ramach platformy. JDT jest to pakiet wtyczek wspomagających tworzenie aplikacji w Javie. Zawiera w sobie między innymi interfejs programistyczny pozwalający na operowanie na zasobach projektu Javy (plikach źródłowych, pakietach, bibliotekach), a także kompilator, czy debugger. To co jednak ważne z punktu widzenia analizy statycznej, to fakt rozszerzalności o wtyczki. Miejscem, gdzie można odszukać wszystkie wtyczki Eclipse'a jest Eclipse Marketplace [40]. W tym momencie dostępnych jest prawie 1500 różnego rodzaju wtyczek wspierających tworzenie oprogramowania, a wśród nich wiele, które wspierają statyczną analizę oprogramowania. Z przedstawionych w dalszej części tego rozdziału narzędzi implementujących metryki analizy statycznej 7 na 10 posiada wtyczki do Eclipse'a.

Eclipse jest środowiskiem *open source'owym*.

5.1.2 Apache Maven

Apache Maven [41] (dalej Maven) jest narzędziem służącym do automatycznego budowania oprogramowania dla języka Java tworzonym na licencji Apache (jest to projekt *open source*). Sposób budowania projektu określony jest w pliku XML-owym POM (ang. *Project Object Model*), który ponadto zawiera informacje o zespole programistów, czy zastosowanych systemach wspierających rozwój oprogramowania. Funkcjonalności Maven'a wprowadzane są za pomocą wtyczek. Każda z wtyczek posiada specyficzne cele do realizacji. Wtyczki dzielą się na używane w czasie budowania i w czasie tworzenia raportów. Z przedstawionych w dalszej części tego rozdziału narzędzi implementujących metryki aż 8 na 10 można używać jako wtyczkę do Maven'a.

Cykl życia (budowania) składa się z następujących faz:

- walidacja - sprawdzenie, czy projekt jest poprawny i wszystkie wymagane informacje są dostępne,
- kompilacja – kompilacja kodu źródłowego,
- testowanie – testuje kod źródłowy używając odpowiedniego środowiska testowania jednostkowego,

- pakowanie – skompilowany kod jest pakowany do formatu dystrybucyjnego takiego jak JAR,
- testowanie integracyjne – zbudowany projekt jest umieszczany w środowisku wykonującym testy integracyjne,
- weryfikacja – następuje sprawdzenie czy pakiet dystrybucyjny spełnia kryteria jakościowe,
- instalacja – pakiet dystrybucyjny umieszczany jest w lokalnym repozytorium do użycia jako zależność w innych projektach,
- wdrożenie - finalny pakiet dystrybucyjny jest publikowany w repozytorium zdalnym

Wykorzystuje się też bardzo często funkcjonalność Maven'a polegającą na tworzeniu strony internetowej z raportem.

Innym narzędziem służącym do budowania projektu jest Ant również na licencji Apache [42]. Ant jest tutaj wspomniane ponieważ narzędzia implementujące metryki często wspierają zarówno Maven'a jak i dla Ant'a. Sposób budowy projektu w przypadku Ant'a opisany jest przy pomocy skryptu *buildfile.* W skrypcie tym opisuje się krok po kroku sposób budowy programu. Natomiast jeśli chodzi o POM Maven'owy, to tutaj kolejność nie ma znaczenia, ponieważ Maven sam ustala poszczególne fazy budowy projektu.

5.1.3 Narzędzia Continuous Integration

Omawiając narzędzia wspierające analizę statyczną oprogramowania należy wspomnieć też o narzędziach ciągłej integracji. Ciągła integracja to praktyka stosowana w inżynierii oprogramowania polegająca na regularnej integracji bieżących zmian w kodzie źródłowym do głównego repozytorium. Martin Fowler sugeruje, aby taka integracja odbywała się nawet każdego dnia po zakończeniu pracy oczywiście po uprzedniej weryfikacji poprawności [43]. Integracja projektu do zdalnego repozytorium odbywa się w ostatniej fazie cyklu życia Maven'a – w fazie wdrożenia. W tym celu można używać np. aplikacji Apache Continuum [44].

5.2 Narzędzia implementujące metryki

Metryki opisane w rozdziale 4 znajdują swoje implementacje w szeregu narzędzi *open source* (narzędzia komercyjne nie są przedmiotem tego przeglądu). Część z tych narzędzi działa jako wtyczki do Eclipse'a, część integruje się z Maven'em. W rozdziale tym

zaprezentowano przegląd dostępnych narzędzi opisując ich główne funkcjonalności skupiając się na metrykach, które są w nich zaimplementowane.

5.2.1 Stan – Structure Analysis for Java

Stan to narzędzie do analizy struktury kodu napisanego w języku Java [45]. Do niekomercyjnych projektów można uzyskać darmową licencję. Aktualnie można też używać Stan'a za darmo dla projektów o liczbie klas mniejszej niż 500.

Strukturę kodu rozumiemy tutaj dwojako:

- jak artefakty (typy) budują artefakty wyższego poziomu
- jakie są zależności pomiędzy artefaktami

Struktura odzwierciedla więc nasz projekt. Oprócz wartości metryk Stan wspiera wizualizacje struktury. Jak twierdzą autorzy, wizualizacje stanowią dobre wsparcie dla metryk, pozwalają lepiej zrozumieć zależności. W codziennej pracy z kodem takie wizualizacje są więc bardzo pomocne. Stan pozwala nam obejrzeć diagram zależności na różnych poziomach np. między klasami, lub między pakietami. Klikając w linie powiązań możemy podejrzeć ich listę. Dostępne są dwa widoki: widok kompozycji, który pozwala podejrzeć zależności wewnątrz danego typu, oraz widok powiązań, pozwalający na przyjrzenie się zależnościom na zewnątrz typu.

Bardzo wygodne są również histogramy wartości poszczególnych metryk. Są one najlepszą formą prezentowania wartości metryk. Niestety nie są dostępne na każdym poziomie abstrakcji i nie dla wszystkich metryk.

Stan proponuje również wartości progowe metryk dzieląc je na trzy kategorie: zielona – wartości zalecane, żółta – wartości pośrednie, czerwona – wartości złe, czyli zalecana refaktoryzacja. Wartości złe i pośrednie raportowane są w widoku naruszeń. Zakres przedziałów może być redefiniowany przez użytkownika.

Do badania jakości kodu Stan używa następujących metryk:

- metryki zliczeń: liczba klas, liczba metod na klasę, liczba pól na klasę, liczba linii kodu, liczba linii kodu na jednostkę (sekcja 4.1.1)
- metryki powiązań: CC (sekcja 4.1), FAT, ACD (sekcja 4.1.2), Tangled (sekcja 4.2.4). Stan oprócz tego, że wylicza metrykę Tangled, to także na diagramach zależności wskazuje krawędź MFS (minimum informacji zwrotnej) oznaczając ją kolorem czerwonym.
- metryki Martina (sekcja 4.2.3)

- metryki z zestawu CK (sekcja 4.2.2)

Stan pozwala również na wygenerowanie raportu o ogólnej jakości kodu, co może być przydatnym narzędziem dla audytorów kodu, czy menedżerów projektu. W raporcie zawarte są wartości metryk dla całego projektu (w niektórych przypadkach średnie wartości np. dla metryk z poziomu klasy).
Stan nie daje natomiast możliwości wyeksportowania wartości metryk dla całego projektu, przez co analiza poza standardowo zaimplementowanymi opcjami jest utrudniona lub nawet niemożliwa.

5.2.2 RefactorIT

Komercyjne narzędzie rozwijane do roku 2008 przez firmę Aqris. Od tego czasu jest dostępne za darmo, ale nie jest już dłużej rozwijane [46]. Działa jako wtyczka do Eclipse'a. Jest to narzędzie, które ogólnie wspiera proces refaktoryzacji. Implementuje szereg różnych funkcjonalności, które służą wsparciu procesu refaktoryzacji takich jak np.:

- Audyty kodu – szereg różnego rodzaju testów wykrywających potencjalne błędy, łamanie standardów języka Java (między innymi używa do tego wbudowanego tutaj narzędzia PMD opisanego w dalszej części pracy 5.2.10).
- Usuwanie zbędnych importów z kodu.
- Znajdywanie zależności dla typów wraz z rysowaniem diagramów zależności (diagramy są jednak słabo czytelne).
- Automatyczne tworzenie konstruktorów i metody `factory`.
- I wiele innych.

Wśród funkcjonalności implementowanych przez program jest też wsparcie dla bardzo wielu metryk:

- metryki zliczeń: LOC, CLOC (liczba linii kodu będących komentarzami) , DC (gęstość komentarzy), EXEC (liczba wyrażeń wykonywalnych), NOT (liczba klas, interfejsów), NOTa (liczba klas abstrakcyjnych i interfejsów), NOTc (liczba klas konkretnych), NP (liczba parametrów) , NOF (liczba pól zdefiniowanych w metodzie), NOA (liczba atrybutów klasy). Część z wymienionych opisana w sekcji 4.1.1.
- metryki powiązań: CC (sekcja 4.1.2)

- metryki Martina (sekcja 4.2.3)
- metryki z zestawu CK bez LCOM i CBO (sekcja 4.2.2). Do wyliczania metryk z zestawu CK nie są brane pod uwagę interfejsy.
- metryki praw i zasad: EP, DIPM, CYC (sekcja 4.2.4).

RefactorIT generuje raport w formacie (txt, html, xml) z wartościami wszystkich wybranych metryk dla wszystkich typów analizowanego poziomu, przez co daje możliwość dalszej analizy metryk w oparciu o wszystkie wartości zbierane z całego projektu bądź wybranego modułu.

5.2.3 Sonar

Sonar [47] to *open source'owa* platforma służąca do zarządzania jakością oprogramowania od strony analizy statycznej oprogramowania. Pierwotnie zaprojektowana do analizy jakości programów napisanych w języku Java. Obecnie dostępne są wtyczki, które pozwalają na analizę kodu napisanego w takich językach C, C#, Flex, Natural, PHP, PL/SQL, Cobol i Visual Basic 6.

Sonar jest też aktualnie najprężniej rozwijanym narzędziem *open source* wspierającym metryki. Wciągu 2 lat miał 15 wydań i odnotował 100 000 ściągnięć. Sonar może być używany jako wtyczka do Meven'a, bądź można go używać bezpośrednio z linii poleceń przy pomocy aplikacji Sonar runner. Wymagana jest instalacja serwera i bazy danych dla Sonar. Wygenerowane raporty dostępne są przez zainstalowany serwer.

Narzędzie oferuje kompleksową analizę kodu, dlatego, że wspiera metryki w takich obszarach jak: architektura i design, duplikaty, testowanie jednostkowe, złożoność, potencjalne błędy, reguły kodowania, komentarze. Dokładnie dla języka Java oferowane jest wsparcie dla następującego zestawu metryk:

- Metryki zliczeń: liczba wszystkich linii w kodzie, liczba linii komentarzy, LOC, gęstość komentarzy, liczba pakietów, liczba klas, gęstość komentarzy, liczba metod, liczba metod typu get i set, liczba wyrażeń wykonywalnych, część opisana w sekcji 4.1.1.
- Metryki powiązań: CC, FAT dla pakietów (tutaj nazwana file edges weight) sekcja 4.1.2.
- Pokrycie testami: LC, BC (sekcja 4.3.2), a także kilka metryk, które mogą być użyte w trakcie wykonywania testów takich jak, liczba testów, czas trwania testów, testy wykonane poprawnie, testy zakończone błędem.

- Duplikaty: DC (sekcja 4.3.1), a także liczbę zduplikowanych linii kodu, liczbę zduplikowanych bloków linii kodu, liczbę plików zawierających duplikaty.
- Metryki CK: z metryk spójności tylko LCOM4, brak metryki WMC (sekcja 4.2.2).
- Metryki Martina: metryka Ca i Ce (sekcja 4.2.3)
- Metryki praw i zasad: metryki wspierające zasadę acyklicznych zależności ACD (sekcja 3.3.3). Sonar proponuje tutaj szereg różnych metryk podobnych do metryki Tangled i CYC (sekcja 4.2.4) pomagających wykryć cykle takich jak np. liczba cykli w pakiecie, liczba zależności do usunięcia, aby zlikwidować cykl.

5.2.4 JDepend

JDepend to narzędzie służące do analizy pakietów i ich zależności [48]. Program uruchamiany jest przy pomocy Ant. Istnieje również wtyczka dla Maven'a. Służy do analizy pakietów. Wylicza następujące metryki:

- metryki Martina (sekcja 4.2.3)
- liczbę klas i interfejsów
- zależności cykliczne między pakietami (metryka CYC sekcja 4.2.4).

5.2.5 CKJM – Chidamber Kemerer Java Metrics

Program służący do analizy klas i pakietów [49]. Rozwinięcie nazwy to: Chidamber and Kemerer Java Metrics. Program oblicza metryki z zestawu CK (sekcja 4.2.2), oprócz tego jedynie metrykę *Ca* (sekcja 4.2.3) oraz metrykę wyliczającą liczbę publicznych metod. Używany jako wtyczka do Maven'a.

5.2.6 Simian – Similiarity analyser

Simian [50] znajduje zduplikowany kod (sekcja 4.3.1) dla takich języków programowania jak: Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic (sekcja 4.3.1). Istnieje wtyczka do Checkstyle'a, Maven'a, a także może być uruchamiany jako zadanie Ant. Program porównuje ciągi znaków i na tej podstawie raportuje duplikaty. 6 powtórzonych linii kodu to domyślna wartość przy której raportowany jest duplikat.

5.2.7 Cobertura

Cobertura to narzędzie, które sprawdza pokrycie testami (sekcja 4.3.1) [51]. Do tego celu wykorzystuje pliki Java class po skompilowaniu (sprawdza, które linie kodu, były osiągnięte w trakcie wykonywania). Następnie generuje raport XML lub HTML, który pokazuje, jakie pakiety, klasy, metody nie zostały przetestowane.

Przeglądając raport widzimy wynik procentowy pokrycia testami na różnych poziomach (pakiety, klasy, metody, linie kodu). Cobertura oblicza pokrycie testami dla linii kodu, ale także dla gałęzi. Na najniższym poziomie możemy podejrzeć ile razy poszczególna linia kodu została osiągnięta w trakcie wykonywania testów.

Cobertura implementuje także metrykę złożoności metod CC (sekcja 4.1.2).

Narzędzia tego można używać jako weryfikatora poprawności testów. Może się zdarzyć tak, że ze względu na jakiś błąd metoda testująca się nie wykona. Np. programista zapomni dodać jakąś metodę testującą do metody uruchamiającej testy jednostkowe, albo pojawi się literówka w nazwie metody. Cobertura pokaże wtedy, że dana część kodu nie została osiągnięta przez metody testujące.

Cobertura uruchamiana jest przez Ant, istnieje także wtyczka dla Meven'a. Wtyczka dla Eclipse'a, nazywa się eCobertura.

5.2.8 Findbugs

Narzędzie służące do wykrywania potencjalnych błędów [52]. Do analizy wykorzystywane są pliki Java class. Może raportować nieprawdziwe ostrzeżenia, zwykle jest ich mniej niż 50%, co wiąże się z tym, że nie wszystkie testy są precyzyjne. Findbugs uruchamiany jest przez Ant, ale istnieją też wtyczki dla Eclipse'a i Maven'a. . Pogrubioną czcionką zaznaczono zestawy reguł, które wytypowano do policzenia metryki potencjalnych błędów języka Java PB (sekcja 4.4). Findbugs dzieli błędy na kilka kategorii :

- ***Correctness bug*** – potencjalny błąd wynikający z prawdopodobnie innego zamiaru programisty (142 testy) np.:
    - DMI_COLLECTIONS_SHOULD_NOT_CONTAIN_THEMSELVES – kolekcja nie powinna zawierać samej siebie
    - SE_METHOD_MUST_BE_PRIVATE – metoda musi być prywatna aby zadziałała serializacja
- *Bad practise* – naruszenie istotnej i zalecanej praktyki kodowania (85 testów) np.:

    - NM_WRONG_PACKAGE – metoda nie przesłania metody z klasy, po której dziedziczy, ponieważ parametr ma inny typ – został zaimportowany z innego pakietu
    - ES_COMPARING_STRINGS_WITH_EQ – używanie operatorów `==` lub `!=` zamiast metody `equals(Object)` dla obiektów typu String
- ***Multithreaded correctness*** – potencjalne błędy związane z wielowątkowością (45 testów) np.:
    - NP_SYNC_AND_NULL_CHECK_FIELD – synchronizacja na polu, które jednocześnie przyjmuje wartość `null`, powoduje wyrzucenie wyjątku NullPointerException
- ***Malicious code vulnerability*** – kod potencjalnie narażony na złośliwe oprogramowanie (15 testów) np.:
    - MS_SHOULD_BE_FINAL – publiczna zmienna statyczna powinna mieć również modyfikator `final`, w przeciwnym razie może być zmieniona przez złośliwe oprogramowanie albo inny pakiet.
- ***Security*** – kod z potencjalnymi błędami dotyczący bezpieczeństwa (11 testów) np.:
    - DMI_CONSTANT_DB_PASSWORD – hasło do bazy danych zapisane w kodzie zamiast pobierane z zewnętrznego źródła
- *Performance* – kod, który można zrefaktoryzować na bardziej wydajny (27 testów) np.:
    - UUF_UNUSED_FIELD – pole nigdy nie jest używane, należy je usunąć
- ***Dodgy code*** – kod niepewny, budzący niepokój, albo napisany w taki sposób, że sam z siebie prowadzi do błędów (72 testy) np.:
    - RI_REDUNDANT_INTERFACES – klasa implementuje ten sam interfejs, co klasa, po której dziedziczy, jest to zbędne, z drugiej strony sugeruje, że mogła się zmienić hierarchia dziedziczenia

Poza tym każdy z wychwyconych błędów otrzymuje ranking: wysoki, średni i niski priorytet. W rekomendacji do programu znajdziemy informację, że w pierwszej kolejności należy zająć się błędami typu *correctness*. W pozostałych kategoriach dopuszczalne jest ignorowanie większej ilości ostrzeżeń.

5.2.9 Checkstyle

Checkstyle to narzędzie, które pomaga programistom pisać kod zgodnie ze standardami obowiązującymi dla języka Java [53]. Pierwotnie narzędzie sprawdzało kod pod kątem potencjalnych problemów z układem kodu, w tym momencie ma jednak znacznie szersze możliwości – wykrywa potencjalne błędy, czy naruszenie dobrych praktyk i standardów języka Java. W Checkstyle programista ma możliwość definiowania wartości granicznych, przy których raportowany jest błąd. Może być uruchamiany przez Maven'a. Istnieje także wtyczka dla Eclipse'a. Pogrubioną czcionką zaznaczono zestawy testów, które wytypowano do policzenia metryki czytelności kodu R (sekcja 4.4).

Narzędzie implementuje szereg testów podzielonych na następujące kategorie:

- ***Annotations*** - poprawność użycia adnotacji, 5 testów, np.:
  - MissingOverride – sprawdza, czy jest obecna adnotacja `Java.lang.Override`, gdy obecny jest komentarz Javadoc `{@inheritDoc}`
- ***Block Checks*** - np. wyszukuje puste bloki, spawdza czy bloki `if`, `else`, `while` otoczone są nawiasami klamrowymi, wyszukuje zagnieżdżone bloki kodu, które sprawiają, że kod jest nieczytelny, 5 testów
- *Class design* – weryfikuje poprawność użycia klas i interfejsów, 8 testów np.
  - InterfaceIsType - interfejs powinien zawierać metody, a nie tylko zmienne.
- ***Coding*** – 44 testy różnego rodzaju np.:
  - ArrayTrailingComma - Sprawdzenie, czy deklaracja tablicy zawiera końcowe przecinki – pozwala na późniejsze łatwiejsze dodanie elementów, bądź zmianę ich kolejności
  - AvoidInlineConditionals - Pisanie warunków w jednej linii np.: *String b = (a==null || a.length<1) ? null : a.substring(1);* Przez wiele firm uważana za uciążliwe do odczytania, przez to zakazane
  - Różne przypadki użycia metody *equals()*.
- *Duplicate code* - zduplikowany kod

  Metoda zaproponowana w Checkstyle porównuje kod linię po linii i raportuje zduplikowanie kodu jeśli sekwencje kodu różnią się tylko wcięciami. Ignorowane są tylko wyrażenia zawierające słowo kluczowe import. Domyślna ilość powtórzonych linii, powyżej której występuje ostrzeżenie wynosi 12.

- ***Header*** - Sprawdza, czy plik z kodem źródłowym rozpoczyna się w odpowiedni sposób np. zawiera informację o projekcie, autorze.
- *Imports* – poprawność użycia importów, 7 testów np.:
    - AvoidStarImport - unikanie importów z użyciem * - import wszystkich klas z pakietu. Przy aktualizacji biblioteki może prowadzić do konfliktów nazw.
- ***Javadoc Comments*** – poprawność użycia komentarzy Javadoc dla pakietów, klas metod i zmiennych, 6 testów np.:
    - JavadocPackage - sprawdzenie, czy każdy pakiet ma swój plik Javadoc
- *Metrics* – metryki złożoności i rozmiaru, większość była przytoczona w poprzednim rozdziale, 6 testów
- ***Miscellaneous*** – zestaw 12 różnych testów, np.:
    - NewlineAtEndOfFile – każdy plik kodu źródłowego powinien się kończyć znakiem nowej linii – ważne w przypadku systemów kontroli wersji np. CVS
    - UpperEll – sprawdza, czy stałe typu long są opisane z dużą literą L. Mała litera może być pomylona z jedynką.
- *Modifiers* – użycie modyfikatorów dostępu zgodnie ze standardem 2 testy, np.:
    - ModifierOrder - Sprawdzenie, czy modyfikatory dostępu występują w kolejności zalecanej przez specyfikację Javy.
- ***Naming Conventions*** - konwencje w nazewnictwie – szereg różnych testów, które sprawdzają, czy zachowane są zgodne ze standardem Javy konwencje w nazewnictwie
    - MemberName – sprawdza konwencje w nazewnictwie dla klas, interfejsów, metod, stałych, pakietów, klas abstrakcyjnych
- ***Regexp*** - wyrażenia regularne, 3 testy np.:
    - RegexpSingleline – pozwala na zdefiniowanie testu, czy w danej linii nie występują wyrażenia zaliczane do złych praktyk np. : `System.out.println()`,`System.exit()` itp. – do zdefiniowania samodzielnego
- *Size Violations* - zaburzenia wielkości – testy wykrywające za duże metody, klasy, pliki, 8 testów np.:
    - ExecutableStatementCount – maksymalna ilość wyrażeń wykonywalnych można zdefiniować np. dla metody, konstruktorów klas (domyślnie 30).

- MethodLength – wyszukuje za długie metody i konstruktory klas (domyślnie 150, gdzie wliczane są puste linie)

- ***Whitespace*** – sprawdza użycie znaków niedrukowalnych – 12 testów np.:
  - GenericWhitespace – w wyrażeniach zwierających znaki < > np. w przypadku list nie powinno być spacji
  - EmptyForInitializerPad – definiuje standard gdy brak inicjalizacji indeksu pętli – domyślnie brak znaku.

5.2.10 PMD

Kolejnym narzędziem, które definiuje zestaw testów do przeprowadzenia w celu wyszukania potencjalnych problemów jest PMD [54]. Może być uruchamiany przez Maven'a. Istnieje także wtyczka dla Eclipse. Pogrubioną czcionką zaznaczono zestawy reguł, które wytypowano do policzenia metryki dobrych praktyk języka Java GP (sekcja 4.4). Program dzieli zestaw testów na następujące kategorie:

- ***Basic*** – podstawowe reguły naruszenia dobrych praktyk kodowania, 22 testy np.:
  - ReturnFromFinallyBlock – należy unikać zwracania wartości w blokach final – może wyrzucić wyjątek
- *Braces* – zestaw 4 testów sprawdzających, czy w wyrażeniach if, while, for, if…else
- ***Clone Implementation*** – zestaw 3 reguł znajdujących niewłaściwe użycie metody `clone()` np.:
  - CloneMethodMustImplementCloneable – metoda clone() powinna być implementowana tylko wtedy, gdy klasa implemntuje interfejs Cloneable
- *Code size* – zestaw 10 reguł – metryk rozmiaru i złożoności podobnych do tych opisanych w poprzednim rozdziale i do tych, które implementuje Checkstyle
- *Comments* – 3 testy dotyczące komentarzy np.:
  - CommentSize – sprawdza, czy rozmiar komentarza mieści się w ustalonym zakresie
- ***Controversial*** – Zestaw 22 reguł z różnych powodów uważanych za kontrowersyjne np.:
  - DontImportSun – należy unikać importowania pakietów 'sun.*', gdyż pakiety te często się zmieniają

- *Coupling* – 5 reguł powiązań pomiędzy obiektami i pakietami, znacznie mniej precyzyjne niż te opisane w poprzednim rozdziale, a wśród nich jest prawo Demeter
    - LowOfDemeter
- ***Design*** – 51 reguł dotyczących designu, wyszukujących nieoptymalne implementacje np.:
    - ConstructorCallsOverridableMethod – konstruktor nie powinine wywoływać metody przesłoniętej, może powodować problemy z późniejszym wywołaniem metody `super()`.
    - EqualsNull – test sprawdzający wartość null nie powinien używać metody `equals()` tylko operatora porównania.
- *Empty code* - zestaw reguł wyszukujących puste wyrażenia różnego rodzaju (puste metody, bloki, puste bloki `try` lub `catch`) 10 testów
- ***Finalizer*** – 6 reguł związanych z metodą `finalize()`, wychwytujące różne problemy, które mogą wystąpić przy okazji użycia tej metody np.:
    - FinalizeShouldBeProtected – metoda, która przesłania metodę `finalize()` powinna mieć atrybut protected
- ***Import Statments*** – 6 reguł związanych z importem np.:
    - ImportFromSamePackage – nie ma potrzeby, żeby importować typ z tego samego pakietu
- ***Java Logging*** - Logowanie – zestaw 2 reguł weryfikujących użycie klasy Logger np.:
    - LoggerIsNotStaticFinal – w większości przypadków Logger powinien być deklarowany jako static i final
- ***JUnit*** – zestaw 11 reguł, eliminujących potencjalne problemy z testami jednostkowymi JUnit np.:
    - JUnitTestsShouldIncludeAssert – JUnit test powinien zawierać przynajmniej jedną asercję
- ***Migration*** – zestaw 14 reguł związanych z migracją pomiędzy kolejnymi wersjami JDK np.:
    - ReplaceVectorWithList – rozważyć zamianę Victor na nowszą ArrayList
- *Naming* – zestaw 20 testów, które dotyczą standardów nazewnictwa np.:
    - AvoidFieldNameMatchingMethodName – należy unikać stosowania tego samego nazewnictwa metody i zmiennej, zmniejsza to czytelność

- ***Optimization*** - reguły optymalizujące kod, zaliczane też do dobrych praktyk, 11 testów np.:
    - AvoidArrayLoops – do kopiowania list należy używać metody `arraycopy()`
- ***Security Code Guidelines*** - reguły bezpieczeństwa – weryfikacja, czy przestrzegany jest standard proponowany przez firmę Sun – 2 testy
- ***Strict Exceptions*** - reguły sprawdzające obsługiwanie wyjątków, 21 testów np.:
    - AvoidCatchingGenericException – należy unikać wyłapywania ogólnych wyjątków takich jak: NullPointerException, RuntimeException
- ***String and StringBuffer*** - operacje na ciągach znaków – reguły dotyczące operacji na klasach String, StringBuffer i StringBuilder, 14 testów np.:
    - UseEqualsToCompareStrings – należy unikać operatorów == oraz != do porównywania obietków String, a używać metody `equals()`
- ***Unnecessary*** - 7 reguł pozwalających wychwycić zbędny kod np.:
    - UnnecessaryReturn – zbędne użycie wyrażenia `return()`
- ***Unused code*** - zestaw 5 reguł wynajdujących kod, który nie jest wykorzystany np.:
    - UnusedPrivateMethod – wynajduje nieużywane metody prywatne
- Oprócz wyżej wymienionych kategorii PMD implementuje też szereg reguł związanych z użyciem różnych platform i technologii takich jak: Android, Ecmascript, JSF, JSP, XML, J2EE, Jakarta, JavaBeans, XPath in XSL

5.3 Podsumowanie

W rozdziale 5 przedstawione zostały narzędzia konieczne do statycznej analizy oprogramowania. Podzielone zostały na dwie kategorie: środowiska programowania oraz narzędzia, które implementują metryki.

Wśród narzędzi implementujących metryki znalazły się też 3 narzędzia oparte o system – reguł testów, które badają kod pod kątem dobrych praktyk, czytelności oraz potencjalnych błędów. Pozostałe 7 narzędzi oblicza większość metryk opisanych w rozdziale 4. Wszystkie opisane narzędzia są darmowe. Za szczególnie użyteczne uznano 3 narzędzia:

- Stan – przede wszystkim za bardzo dobrą wizualizację zależności na wszystkich poziomach abstrakcji oraz generowane histogramy wartości niektórych metryk.

- RefactorIT – największą ilość zaimplementowanych metryk oraz generowanie raportu z wartościami dla każdego typu do którego odnosi się metryka, co umożliwia później własną analizę na kompletnych danych dla całego systemu.
- Sonar – najszybciej rozwijające się narzędzie z największą liczbą wydań, powszechnie stosowane również w projektach komercyjnych, oferujące jednocześnie kompleksową analizę statyczną.

Nie udało się odnaleźć narzędzia *open source* dla języka Java, które implementowałoby metryki z zestawu MOOD (4.2.1). Komercyjne narzędzie, które implementuje ten zestaw metryk to np. Essentials Metrics firmy Powersoft [55]. Nie znaleziono też narzędzia, które implementowałoby metryki LCOM2 i LCOM3.

Tabela 2 zawiera zestawienie narzędzi implementujących metryki wraz z podstawowymi informacjami na ich temat.

Tabela 2. Narzędzia *open source* implementujące metryki analizy statycznej języka Java. Tabela zestawia narzędzia wraz z metrykami, które implementują. Ponadto zwiera informacje o tym, czy dane narzędzie zawiera wsparcie dla Maven'a i Eclipse'a, sposób prezentacji metryki w programie oraz najważniejsze zalety analizowanego narzędzia.

	Metryki								Wsparcie		Sposób prezentacji	Inne zalety
Narzędzie	Rozmiaru i Złożoności	Duplikaty	Pokrycie testami	MOOD	Zestaw CK	Martina	Prawa, zasady	Dobre praktyki, potencjalne błędy, czytelność	Maven	Eclipse		
Stan	13 metryk zliczeń, CC, FAT, ACD	-	-	-	WMC, DIT, NOC, RFC, LCOM1	+	Tangled	-	N	T	Histogramy, Naruszenia, Wartości średnie	Wizualizacje struktury
RefactorIT	11 mertryk zliczeń, CC	+	-	-	WMC, DIT, NOC, RFC	+	CYC, EP, DIPM	-	N	T	Wartości, Naruszenia	Wsparcie dla procesów refkatoryzacji
Sonar	14 metryk zliczeń, CC, FAT	+	+	-	DIT, NOC, RFC, LCOM4	Ca, Ce	Tangled CYC	+	T	T	Histogramy	Zawiera w sobie wtyczki do innych narzędzi
JDepend	2 metryki zliczeń	-	-	-	-	+	CYC	-	T	N	Wartości	-
Ckjm	-	-	-	-	+	Ca	-	-	T	N	Wartości	-
Simian	-	+	-	-	-	-	-	-	T	N	Wartości	-
Cobertura	CC	-	+	-	-	-	-	-	T	T	Wartości	-
Findbugs	-	-	-	-	-	-	-	+	T	T	Naruszenia	-
Checkstyle	8 metryk zliczeń, CC, NPC, DAC, CFO,	+	-	-	-	-	-	+	T	T	Naruszenia	-
PMD	9 metryk zliczeń, CC, NPC	+	-	-	-	-	VOD	+	T	T	Naruszenia	-

6 Zastosowanie metryk w praktyce

W tej części pracy użyto wybranych narzędzi w celu obliczenia metryk dla systemu GridSpace2 (GS2) rozwijanego przez zespół Distributed Computing Environments Team [3] działający w Akademickim Centrum Komputerowym CYFRONET AGH. GS2 stanowi część wirtualnego laboratorium projektu PL-Grid [56].

W ramach projektu PL-Grid powstała naukowa infrastruktura informatyczna oparta na klastrach komputerów. Umożliwia ona geograficznie rozproszonym zespołom integrację danych doświadczalnych i wyników symulacji komputerowych. Daje możliwość prowadzenia badań naukowych polskim naukowcom i studentom w oparciu o symulacje i obliczenia dużej skali z wykorzystaniem klastrów komputerów, a także zapewnia wygodny dostęp do rozproszonych zasobów komputerowych [57].

GS2 jest typem oprogramowania, które podlega ciągłym zmianom w związku z potrzebą jego adaptacji do wciąż zmieniającego się kontekstu i środowiska jego działania:

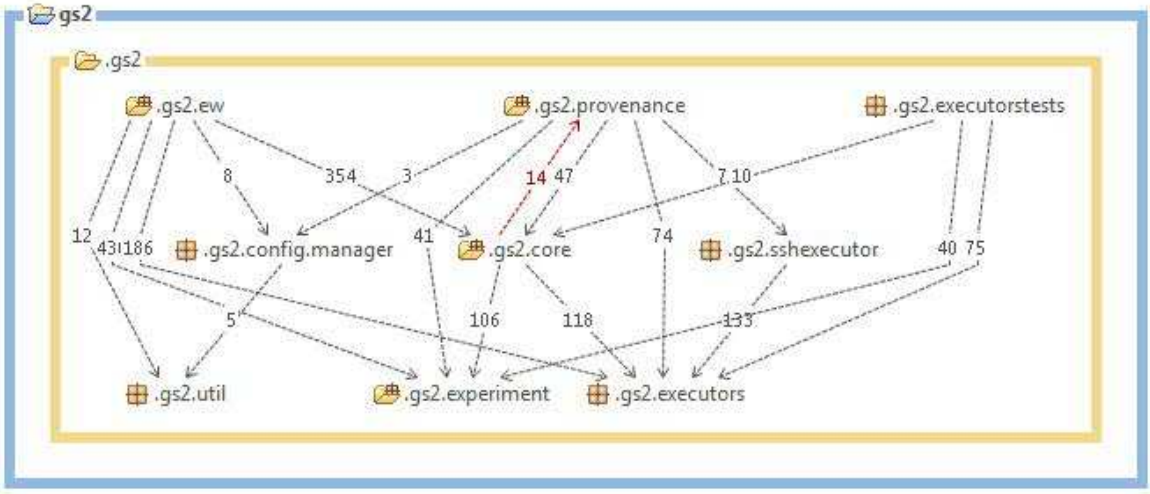
- powinny być wspierane wymagania specyficznych modelów obliczeniowych zarówno tych już istniejących jak i nowo powstających,
- wspierane winny być także różnego rodzaju e-infrastruktury, które w sposób ciągły ewoluują,
- oprogramowanie, które jest wykorzystywane do specyficznych symulacji obliczeniowych jest wciąż rozwijane i podlega nowym wydaniom.

Z tego względu kluczowym elementem w efektywnym rozwoju GS2 jest jego architektura i projekt, która pozwoliłaby na zminimalizowanie ryzyk i nakładu prac przy ciągłym adaptowaniu się do zmian w takim środowisku, przy jednoczesnym zapewnieniu stabilności i

niezawodności rozwiązania. Dodatkowa możliwość weryfikacji jakości kodu tego systemu przy użyciu metryk statycznej analizy będzie więc cenną wartością dodaną.

Analizowana część kodu GS2 to 8 modułów rozwijanych jako projekty Maven'owe. Kod jest analizowany z wyłączeniem klas testujących.

Przy pomocy wtyczki do Eclipse'a aplikacji Stan (sekcja 5.2.1) udało się rozczytać zależności pomiędzy modułami, co jest niewątpliwą zaletą tej aplikacji. Aplikacja ta pozwala analizować jednocześnie kilka projektów. Diagram zależności pomiędzy analizowanymi modułami przedstawia Rysunek 5. Formalnie pakiety `gs2.config.manager` i `gs2.util` stanowią część modułu `gs2.spi`. Na krawędziach umieszczona jest liczba powiązań między modułami (np. wywołanie metody, dziedziczenie itp.) i kierunek zależności.



Rysunek 5. Diagram zależności GS2 stworzony przez aplikację Stan. Powiązania między modułami oznaczone są poprzez strzałkę wskazującą kierunek zależności, dodatkowo nad strzałką zaznaczono liczbę powiązań. Pomiędzy modułami provanance i core widoczny jest splot, co oznacza łamanie zasady ADP.

W Tabeli 3 podano wartości sumy podstawowych typów dla całego systemu GS2 (wartości odczytane z programu Stan). Pokazują one ilościowo z jak dużym projektem mamy do czynienia. Jeśli chodzi o liczbę klas zliczane są klasy najwyższego poziomu, bez klas wewnętrznych. Metryka LOC nie uwzględnia importów, komentarzy i pustych linii.

Tabela 3. Podstawowe metryki rozmiaru dla GS2. Analizowany system jest średniej wielkości.

Typ	Wartość
LOC	27256
Liczba klas	413
Liczba pakietów	68

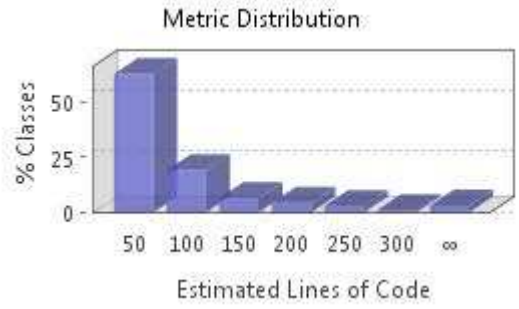
W dalszej części rozdziału prezentowane będą wartości metryk omówione w rozdziale 4 z zachowaniem kolejności w jakiej tam występowały. Wartości metryk najlepiej prezentować przy użyciu histogramu. Taka forma pozwala najlepiej ocenić jakość badanych klas, pakietów i relacji między nimi, daje możliwość szybkiej weryfikacji jaki procent analizowanych typów należy przejrzeć i ewentualnie poddać refaktoryzacji. Wraz histogramem prezentowana będzie obok tabela z 5 typami (metody, klasy lub pakiety), które mają najwyższą wartość badanej metryki. W przypadku naruszenia zalecanej wartości jest to rekomendacja pierwszych kandydatów do przeglądu i potencjalnej zmiany.

6.1 Metryki zliczeń

Metryki zliczeń (sekcja 4.1.1) zaprezentowano przy użyciu aplikacji Stan (sekcja 5.2.1).

Na Rysunek 6 oraz 8 pokazane są podstawowe metryki zliczeń (LOC oraz NOF) wewnątrz klasy dla wszystkich klas systemu. Metryka NOM nie jest prezentowana, gdyż w dalszej części znaleźć można metrykę WMC z zestawu CK, która lepiej oddaje wartość metod w klasie, gdyż jest to suma ważona, gdzie wagę stanowi metryka CC dla metod. Bazując na rekomendacji Stan'a mamy 11 klas, które przekraczają zalecaną wartość progową równą 400 LOC/klasę oraz jedną klasę, której wartość przekracza 40 – wartość progową zdefiniowaną dla metryki NOF. Patrząc na tabele z największą liczbą naruszeń dla metryk NOF, LOC i WMC widać, że część klas się powtarza dla tych trzech metryk – można przypuszczać, że metryki te są skorelowane. Wydaje się to być dość naturalne ponieważ im klasa ma więcej linii kodu, tym przypuszczalnie więcej metod, a te z kolei zapewne odwołują się do większej liczby pól.

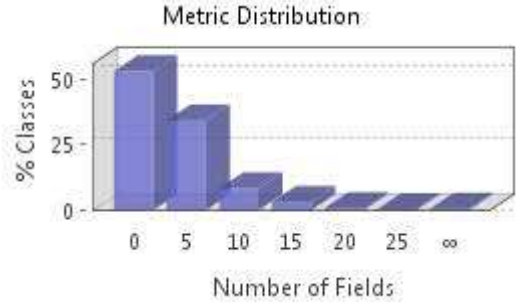
Rysunek 8 przedstawia liczbę klas w pakietach dla wszystkich pakietów. Ponad 70% pakietów ma poniżej 5 klas, oznacza to, że pakiety mają wąską specjalizację i nie są zbyt złożone. Żaden z pakietów nie narusza zalecanej przez Stan maksymalnej wartości klas w pakiecie – 40.



Rysunek 6. Metryka LOC dla klas. 11 klas przekracza rekomendację Stan'a dla LOC równą 400. 80% klas ma LOC poniżej 100.

Tabela 4. 5 klas o najwyższych wartościach LOC. Zalecany podział klas.

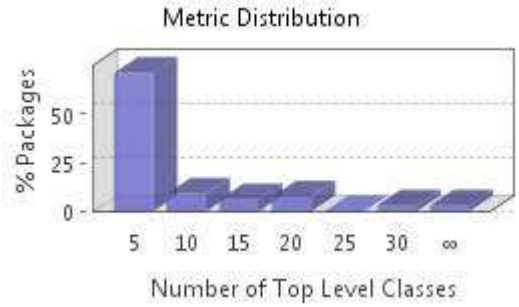
Klasy	LOC
ew.workbench.client.ui.SnippetPanel	951
ew.workbench.client.ui.FileBrowserPanel	771
sshexecutor.SSHFileManager	725
core.execution.ExperimentSessionBase	614
sshexecutor.ModifiedSCPClient	512



Rysunek 7. Metryka NOF. 80% klas posiada mniej niż 5 pól co ułatwia spójność klas i ogranicza złożoność.

Tabela 5. Klasy o najwyższych wartościach NOF. Jedynie klasa SnippetPanel przekracza wartość progową rekomendowaną przez Stan'a na 40.

Klasy	NOF
ew.workbench.client.ui.SnippetPanel	45
ew.workbench.client.ui.FileBrowserPanel	32
ew.webgui.client.richtoolbar.RichTextToolbar	32
ew.workbench.client.ui.MainSitePanel	27
ew.workbench.client.ui.ExperimentPanel	24



Rysunek 8. Metryka TLC dla pakietów. Ponad 70% pakietów ma poniżej 5 klas, co oznacza, że pakiety nie są zbyt złożone i mają wąską specjalizację

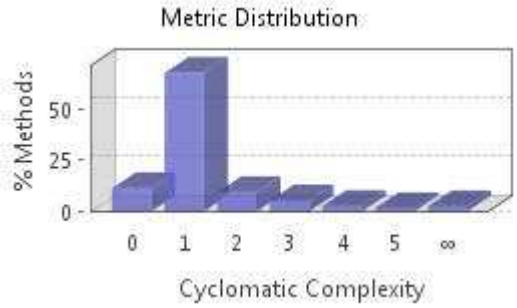
Tabela 6. Pakiety z najwyższą wartością TLC. Brak naruszeń rekomendacji Stan'a równej dla TLC max 40.

Klasy	TLC
ew.base.service	33
executors	31
core.execution	30
ew.workbench.client.ui	27
experiment.client	19

6.2 Metryki powiązań

W tym podrozdziale zostaną opisane wartości metryk powiązań (sekcja 4.1.2).

Metryka CC. Na Rysunek 9 przedstawiono złożoność cyklomatyczną dla metod - CC. Jak widać na histogramie metody nie są zbyt skomplikowane. Jedynie ok. 2,5% metod ma CC wyższe niż 5, podczas gdy wartości poniżej 10 uważane są jako dobre. Część metod przyjmuje CC równe zero, gdyż w Stan'ie metryka CC jest zdefiniowana jako liczba punktów decyzyjnych takich jak if/for/while/case/catch, czy innych gdzie przepływ nie jest liniowy.



Rysunek 9. Złożoność cyklomatyczna CC dla metod. Jedynie 2,5% metod przyjmuje wartości CC wyższe niż 5, co jest bardzo dobrym rezultatem.

W Tabeli 7 pokazano 5 metod z największą wartością metryki CC – kandydaci do refaktoryzacji. W całym systemie wykryto jedynie 11 metod powyżej zalecanego progu CC = 10.

Tabela 7. 5 metod z najwyższą wartością metryki CC. Zaleca się aby wymienione tutaj metody rozbić na mniejsze. W całym systemie wykryto tylko 11 metod przekraczających zalecany próg równy dla CC 10.

Metoda	CC
`core.execution.ExperimentSessionBase.runSnippet(SnippetRun, String)`	36
`ew.webgui.client.richtoolbar.RichTextToolbar$EventHandler.onClick(ClickEvent)`	22
`ew.webgui.client.WebGuiDisplay.createField(FieldType, String, String, Map<String, String>)`	20
`cyfronet.gs2.ew.base.server.OpenerController.getFileContent(HttpServletRequest, WebRequest, String, String, String, HttpServletResponse)`	18
`core.Populator.populate(String, Variables)`	16

Metryka FAT. Metryka ta stanowi uproszczone uogólnienie metryki CC na poziom klas i pakietów i informuje o liczbie wszystkich powiązań wewnątrz typu. Niestety Stan, gdzie zaimplementowana jest ta metryka nie udostępnia jej histogramu. Dostępny jest jedynie

widok naruszeń, gdzie możemy zobaczyć klasy i pakiety przekraczające wartość FAT = 60, a więc próg (taki sam dla klas i pakietów), dla którego raportowane jest ostrzeżenie. Natomiast dopiero dla FAT > 120 zalecana jest refaktoryzacja. I tak w GS2 mamy tylko jeden pakiet przekraczający wartość 60 i jest to `core.execution` o wartości FAT = 66, a więc wartość dopuszczalna. Brak naruszeń metryki FAT dla pakietów wiąże się na pewno mocno z faktem, że pakiety są stosunkowo niewielkie jak pokazały wartości metryki TLC. Tabela 8 prezentuje najwyższe wartości metryki FAT dla klas. 3 klasy przekraczają wartość metryki FAT zalecaną do refaktoryzacji, czyli 120. W sumie jednak jedynie 13 klas przekracza wartość 60 – próg „ostrożności". Klasy o najwyższej wartości FAT, znajdują się też wśród klas o wysokich wartościach takich metryk jak LOC, NOF, WMC. Metryki zliczeń są wyraźnie skorelowane z metrykami powiązań – taki wniosek można wysnuć z poziomu klas. Wydaje się to być naturalne, ponieważ jeśli dodajemy pola w klasie, to chcemy mieć do nich dostęp poprzez metody.

Tabela 8. 5 klas o najwyższej wartości metryki FAT. 3 klasy w GS2 przekraczają wartość FAT zalecaną do refaktoryzacji.

Klasa	FAT
ew.workbench.client.ui.SnippetPanel	237
ew.workbench.client.ui.FileBrowserPanel	174
ew.workbench.client.ui.ExperimentPanel	130
sshexecutor.SSHFileManager	105
ew.base.client.ui.FileUploadPresenter	93

Metryka ACD. Metryka zaproponowana przez aplikację Stan. Niestety przez to, że wartości progowe nie są w Stanie definiowane i nie jest dostępny histogram dla tej metryki, znana jest tylko ogólna średnia jej wartość dla pakietów i klas w całym systemie. Aby wyświetlić wartości dla poszczególnych pakietów i klas, należy je wybrać każdy z osobna. Podane zostaną więc tylko średnie wartości metryki na poziomie klas i pakietów. I tak na poziomie pakietów metryka wynosi średnio 10,64% a na poziomie klas w systemie 5,68%. Wartości te wydają się być niskie. Metryka ACD to średnia względna zależność komponentów od innych komponentów. Metryka ta jest ciekawa z tego względu, że liczą się zależności nie tylko bezpośrednie, ale również pośrednie, czyli niejako uwzględniane są łańcuchy wywołań. Warto się zapoznawać z jej wartością ponieważ daje ona informację np. ile procent klas w pakiecie może mieć bezpośredni bądź pośredni wpływ na rozpatrywaną klasę.

6.3 Metryki z zestawu CK

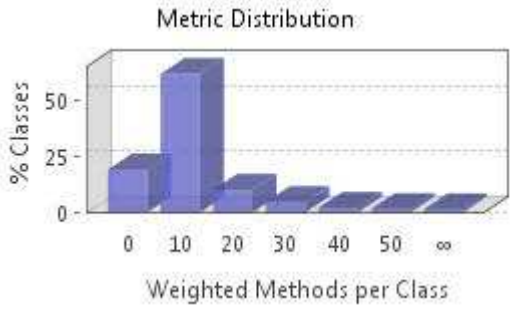
Oprócz metryki DIT wszystkie metryki z zestawu CK wygenerowano przy pomocy aplikacji Stan (sekcja 5.2.1). Zestaw CK został opisany w sekcji 4.2.2.

Na Rysunku 11, 12 i 13 przedstawiono odpowiednio wartości metryk WMC, RFC, CBO. Około 5% klas przekracza zalecaną przez Kan'a wartość progową metryki WMC równą 20 [26]. Jednak jeśli odniesiemy się do zaleceń opartych o projekty obiektowe NASA to jedynie 4 klasy przekraczają wartość WMC = 100 zalecaną do przeglądu i ewentualnej refaktoryzacji [27]. Około 75% klas ma WMC mniejsze niż 10, co trzeba uznać za dobry rezultat, oznacza to, że klasy nie są zbyt złożone.

Jeśli chodzi o metrykę RFC to uzyskane wartości są również bardzo dobre, bo niskie. Prawie 80% klas ma odpowiedź klasy poniżej 10, a jedynie około 6% klas ma wartości RFC przekraczające 30. W zestawieniu z projektami NASA można powiedzieć, że klasy w GS2 mają wartość metryki RFC średnio dwa razy niższą. Kilka klas ma wartości wyższe niż 100 (Tabela 10). Żadna klasa nie przekracza jednak progu 200 zdefiniowanego przez Rosenberg dla systemów NASA jako próg do refektoryzacji [27].

Metryka CBO przyjmuje wartość zero dla około 25% klas. To jedynie kilka procent mniej niż w projektach NASA. Oznacza to, że ¼ klas jest samowystarczalna i nie zawiera odwołań do metod klas zewnętrznych innych niż poprzez dziedziczenie. Są to najbardziej stabilne klasy. 70% klas ma wartości CBO z przedziału od 1 do 15 (podobnie sytuacja wyglądała w projektach NASA). Jedynie 1,2% klas przekracza wartość metryki CBO = 25 (wartość progowa definiowana przez Stan) i to są klasy, które należy przejrzeć i potencjalnie zrefaktoryzować.

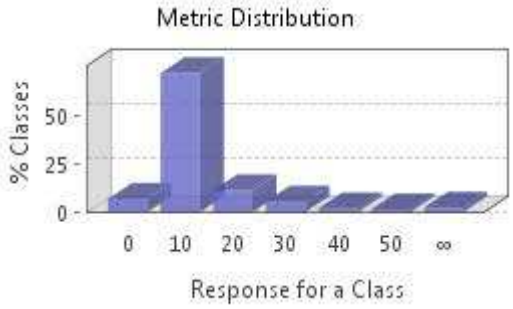
Analizując tabele z najwyższymi wartościami WMC, RFC i CBO widzimy, że dla tych trzech metryk powtarza się część klas w pierwszej piątce klas z najwyższymi wartościami. Zgadza się to z obserwacją Chidambera i Kemerera dla 3 aplikacji komercyjnych [32], w którym zauważono, że wysokie wartości metryk WMC pociągają za sobą wysokie wartości metryk RFC, CBO. Autorzy wykryli również, że te trzy metryki są skorelowane, ale nie znalazło to potwierdzenia w pracy Basili'ego [32].



Rysunek 10. Metryka WMC. 75% klas ma wartość WMC < 10, co oznacza, że klasy nie są zbyt złożone.

Tabela 9. WMC najwyższe wartości dla 5 klas. Jedynie 4 klasy łamią rekomendację NASA dla WMC równą 100, jako zalecaną do podziału.

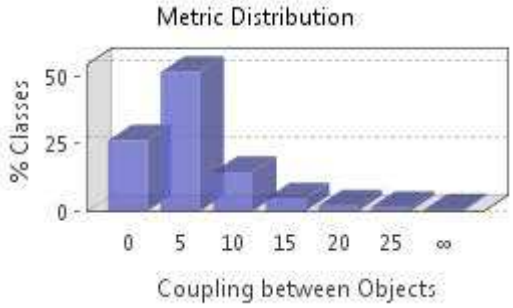
Klasy	WMC
`ew.workbench.client.ui.SnippetPanel`	136
`sshexecutor.SSHFileManager`	120
`sshexecutor.ModifiedSCPClient`	118
`core.execution.ExperimentSessionBase`	109
`ew.workbench.client.ui.ExperimentPanel`	78



Rysunek 11. Metryka RFC. 80% klas ma RFC poniżej 10, a jedynie 6% przekracza próg 20. Wartości średnio 2 razy niższe niż w NASA.

Tabela 10. RFC najwyższe wartości dla 5 klas. Brak naruszeń wartości progowej NASA ustalonej na 200.

Klasy	RFC
`ew.workbench.client.ui.SnippetPanel`	195
`ew.workbench.client.ui.FileBrowserPanel`	159
`ew.workbench.client.ui.ExperimentPanel`	140
`core.execution.ExperimentSessionBase`	137
`ew.workbench.client.ui.MainSitePanel`	126



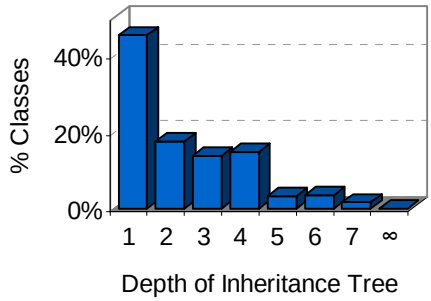
Rysunek 12. Metryka CBO. 25% klas nie zawiera odwołań do metod innych klas. Pozostałe za wyjątkiem 4 klas nie przekraczają wartości progowej.

Tabela 11. CBO najwyższe wartości dla 5 klas. Jedynie 4 klasy naruszają próg 25 zdefiniowany w Stan'ie jako zalecany do refakotryzacji.

Klasy	CBO
`ew.workbench.client.ui.ExperimentPanel`	28
`ew.workbench.client.ui.SnippetPanel`	26
`ew.workbench.service.ExperimentSession.`	26
`ew.base.client.BaseEntryPoint`	26
`core.execution.ExperimentSessionBase`	24

Rysunek 13 oraz Rysunek 14 prezentują wartości metryk z zestawu CK, które są powiązane z dziedziczeniem – DIT i NOC. W przypadku metryki DIT jej wartości zostały pobrane z programu RefactorIT, gdyż wartości podawane przez Stan'a nie były zgodne z definicją. Głębokość drzewa dziedziczenia powinna sumować wszystkie poziomy klas, aż do

klasy `java.lang.Object`, uwzględniając również zewnętrzne biblioteki wykorzystywane w programie. DIT w Stan'ie natomiast uwzględnia jedynie klasy napisane w projekcie (jest to zapewne błąd implementacyjny, gdyż w instrukcji Stan'a DIT jest poprawnie zdefiniowany).



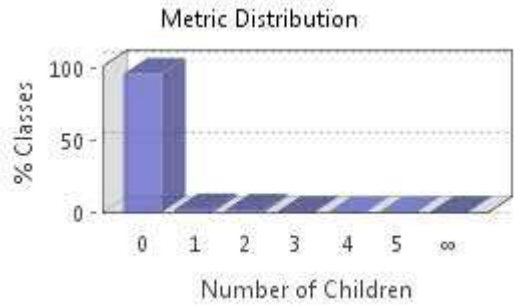
Rysunek 13. Metryka DIT. Ponad połowa klas dziedziczy po jakiejś innej klasie, głównie z wykorzystaniem zewnętrznych bibliotek.

Tabela 12. 5 klas o najwyższej wartości DIT. Wszystkie mają wartość DIT równą 7. Stan zaleca ograniczenie wartości DIT do max 8.

Klasy	DIT
`ew.webgui.client.WebGuiDisplay`	7
`ew.workbench.client.ui.UserWalletManagerPanel$ActionsPanel`	7
`ew.base.client.ui.util.DirectionsTextBox`	7
`ew.workbench.client.timeout.PopupTimeoutAlertWidget`	7
`ew.workbench.client.ui.InterpreterConfigPanel$VarTextBox`	7

Analizując histogramy obydwu metryk możemy zauważyć, ze mechanizm dziedziczenia w analizowanym projekcie wykorzystywany jest głównie poprzez zewnętrzne biblioteki. Ponad połowa klas ma metrykę DIT większą niż 1 (każda klasa w Javie implementuje klasę Object, więc minimalna wartość to 1). Natomiast w sumie jedynie 25 klas ma bezpośrednich potomków (NOC > 1), co oznacza, że w samym projekcie rzadko pisano klasy, które by później były rozszerzane z wykorzystaniem dziedziczenia. Może to wynikać z charakteru systemu. Nie wszędzie jest to konieczne, szczególnie jeśli korzysta się z zewnętrznych pakietów, a implementuje jedynie finalne rozwiązanie.

Rysunek 15 pokazuje rozkład braku spójności metod wewnątrz klas. Widzimy, że znaczna część klas wydaje się być spójna. Jedynie ok. 10% klas ma LCOM1 większe niż 10. Natomiast 42 klasy przyjmują niepokojące wartości z LCOM1 > 50. I to są klasy, które są kandydatami do przeglądu pod kątem spójności i ewentualnie dalszego podziału. Co prawda sama metryka LCOM1 była mocno krytykowana i w pracy Basili'ego [32] nie wykryto korelacji pomiędzy wysoką wartością metryki a liczbą błędów, to jednak Chidamber i Kemerer [33] zaobserwowali powiązania tej metryki z takim zewnętrznymi metrykami jak np. produktywność. Produktywność programisty piszącego klasę jest niższa im wyższa jest wartość LCOM1. Chociażby z tego względu warto unikać klas o wysokiej wartości metryki LCOM1.

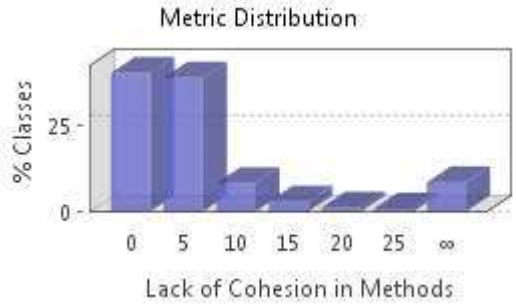


Rysunek 14. Metryka NOC. 95% klas GS2 nie posiada potomków. Oznacza to, że implementowane klasy nie są rozszerzane przez dziedziczenie.

Tabela 13. 5 klas on najwyższych wartościach metryki NOC.

Klasy	NOC
ew.base.service.eventing.impl.BasicEvent	11
provenance.AsyncProvenanceCollectorTask	3
ew.base.client.ui.util.OutputPanel	2
ew.widget.client.ui.BaseWidget	2
ew.widget.shared.WidgetInfo	2

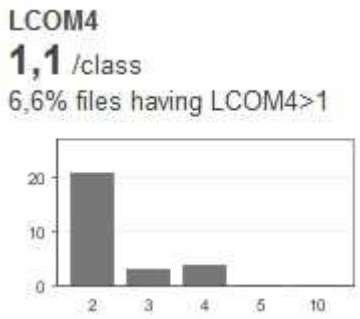
Pewniejszej informacji na temat spójności klasy dostarczy nam metryka LCOM4 - Rysunek 16. Wartości metryki i rysunek wygenerowane zostały przez Sonar. Zgodnie z badaniami Elzkorn [25] metryka ta dużo lepiej sprawdza się w weryfikacji spójności klas niż LCOM1. 6,6% klas posiada LCOM4 większe niż jeden i te klasy należy podzielić. Ponieważ zgodnie z definicją LCOM4 to liczba zbiorów rozłącznych lokalnych metod, to klasę należy podzielić na tyle klas, ile jest tych zbiorów. Sprawę ułatwia tutaj Sonar, ponieważ daje możliwość podejrzenia każdej klasy z LCOM4 >1 i grupuje metody i pola klasy właśnie w rozłączne zbiory. Warto odnotować, że wśród 5 klas o najwyższych wartościach LCOM1 znalazła się tylko jedna, która spełniała podobny warunek dla LCOM4. Prawdopodobnie metryki te nie są skorelowane.



Rysunek 15. Metryka LCOM1. Około 10% klas przyjmuje niepokojące wartości LCOM1 > 50, są to klasy, które trzeba przejrzeć pod kątem spójności. Z drugiej strony LCOM1 miała negatywną praktyczną weryfikację.

Tabela 14. LCOM1 najwyższe wartości dla 5 klas. Dla 5 klas metryka LCOM1 przyjmuje bardzo duże wartości – ponad 300 do prawie 700. Należy je zweryfikować przy pomocy np. LCOM4.

Klasy	LCOM1
ew.base.service.ExperimentWorkbenchConfig	695
sshexecutor.SSHFileManager	634
ew.workbench.client.ui.SnippetPanel	585
ew.workbench.client.ui.ExperimentPanel	575
ew.widget.client.ui.InputOutputWidget	319



Rysunek 16. Metryka LCOM4. 6,6% klas ma LCOM4 > 1. Są to klasy, które trzeba podzielić na taką liczbę klas, na jaką wskazuje wartość metryki. Na osi poziomej mamy wartość metryki, a na pionowej liczbę klas o zadanej wartości.

Tabela 15. 5 klas o najwyższej wartości LCOM4. Maksymalnie metryka osiąga wartość 4.

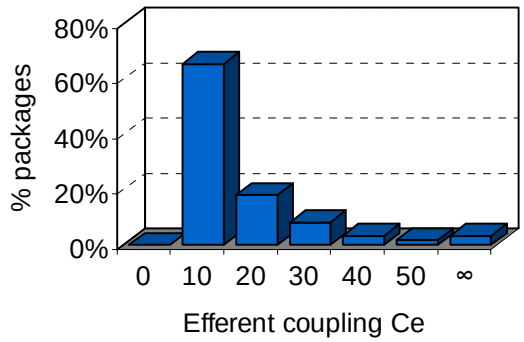
Klasy	LCOM4
ew.widget.client.ui.BaseWidget	4
ew.widget.client.ui.InputOutputWidget	4
ew.workbench.client.ui.InputOutputForm	4
executors.AbstractExecutor	4
ew.widget.client.ui.MasterWidget	3

6.4 Metryki Martina

Zestaw metryk Martina został opisany w sekcji 4.2.3. Histogramy metryk zaprezentowano przy użyciu danych wygenerowanych przez program RefactorIT (sekcja 5.2.2). Jedynie ostatni wykres (Rysunek 21.) wygenerowany został przez program Stan (sekcja 5.2.1).

Rysunek 17 i 19 prezentują wartości powiązań do zewnątrz Ce i powiązań do wewnątrz Ca dla pakietów. Jeśli chodzi o powiązania do zewnątrz, a więc klasy z zewnętrznych pakietów, które mają wpływ na klasy w rozpatrywanym pakiecie to ok. 80% pakietów nie przekracza zalecanej przez RefactorIT wartości 20. Dla pozostałych należy rozważyć podział na mniejsze pakiety. Pakiety o wyższych wartościach metryki Ce są niestabilne, przez to bardziej narażone na błędne działanie. Zgodnie z przewidywaniami najwyższa wartość Ce przypada dla pakietu UI.

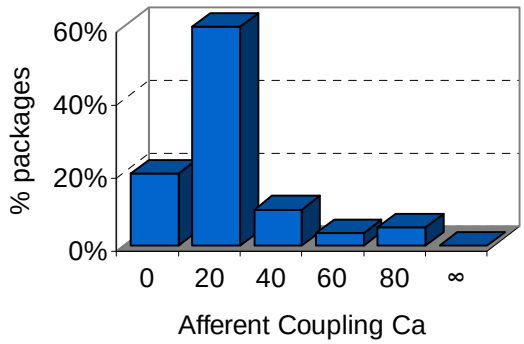
Jeśli chodzi o powiązania do wewnątrz, to żaden z pakietów nie ma metryki Ca większej niż 90, co jest wartością dużo niższą niż zdefiniowany przez RefactorIT próg 500. Dla metryki Ca również preferowane są niższe wartości, gdyż im wyższa wartość metryki Ca tym większa odpowiedzialność rozpatrywanego pakietu przed innymi pakietami i większe prawdopodobieństwo propagowania błędu, gdyby taki wystąpił.



Rysunek 17. Metryka Ce. 20% pakietów przekracza bezpieczny poziom powiązań do zewnątrz ustalony w RefactorIT na 20.

Tabela 16. Metryka Ce - najwyższe wartości dla 5 pakietów. Zgodnie z przewidywaniami największą wartość Ce osiągnęła klasa interfejsu użytkownika, przekracza jednak znacznie zalecany próg 20.

Pakiety	Ce
ew.workbench.client.ui	185
ew.base.service	54
ew.base.client.ui.util	42
core.execution	34
executors	31

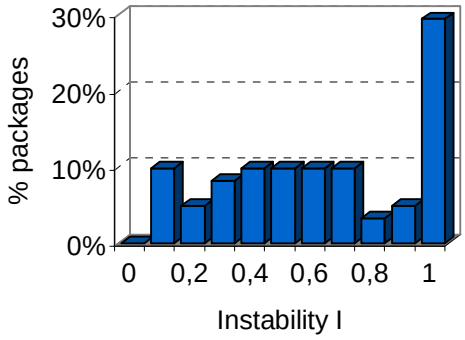


Rysunek 18. Metryka Ca. 80% pakietów nie przekracza wartości 20, co jest bardzo dobrym rezultatem.

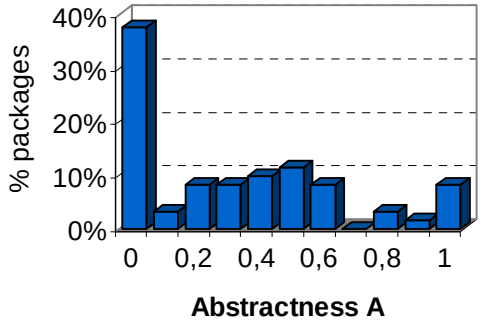
Tabela 17. Ca najwyższe wartości dla 5 pakietów. Pakiety o najwyższej wartości Ca nawet nie zbliżają się do progu 500 podanego przez RefactorIT.

Pakiety	Ca
ew.base.shared	89
ew.base.client.resources	68
ew.base.client	65
experiment.client	63
ew.workbench.client	54

Rysunek 19 i 21 pokazują znormalizowane metryki niestabilności i abstrakcyjności pakietów. Najlepiej jeśli te dwie metryki przyjmują skrajne wartości. W praktyce jednak w dobrym systemie około połowa pakietów przyjmuje wartości pośrednie [35]. Jeśli założymy, że wartości skrajne stanowią przedziały ok. 10% wartości metryk najniższych i najwyższych, to sytuacja gdzie połowa pakietów przyjmuje wartości pośrednie ma miejsce w analizowanym systemie. Nie prezentowano tutaj nazw pakietów, ponieważ jeśli chodzi o niepożądane wartości metryk I oraz A to najistotniejsza jest relacja między tymi metrykami oraz odległość od ciągu głównego Dn.



Rysunek 19. Metryka I. Brak zalecanej, wyraźnej koncentracji pakietów w obszarze stabilnym (niskie wartości I). Widoczna, zalecana koncentracja w obszarze niestabilnym (wysokie wartości I).



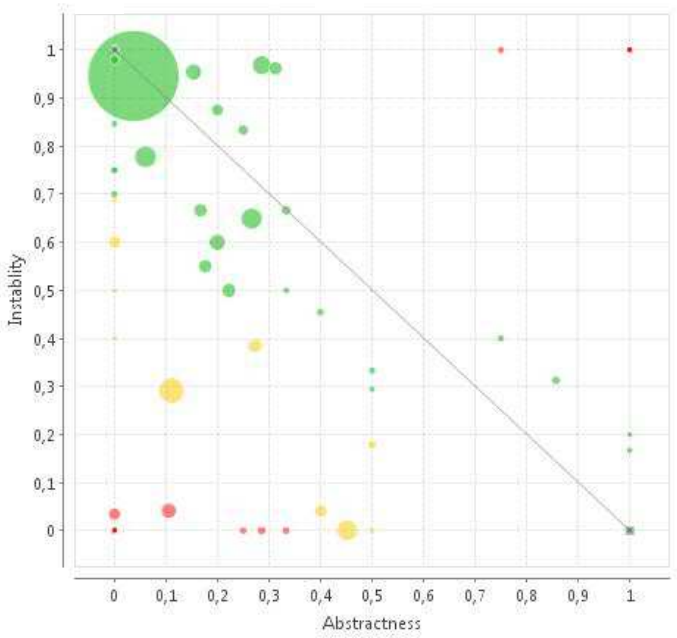
Rysunek 20. Metryka A. Widoczna, zalecana koncentracja w obszarze konkretnym (niskie wartości A). Brak zalecanej koncentracji w obszarze abstrakcyjnym (wysokie wartości A).

Rysunek 21 prezentuje wartości metryk I oraz A dla pakietów na tle ciągu głównego. Wykres ten został wygenerowany przy użyciu programu Stan. Pakiety zostały oznaczone kołami – im większe koło, tym pakiet zawiera więcej klas. Dodatkowo na zielono oznaczono pakiety, których wartość metryki Dn mieści się w przedziale [0 ; 0,3], na żółto [0,3 ; 0,6] oraz na czerwono [0,6 ; 1]. W codziennej pracy z programem Stan wskazując myszką na wybrane koło możemy podejrzeć do jakiego pakietu się ono odnosi.

Analizując rysunek można powiedzieć, że mniej więcej spełniona została zasada, że pakiety niestabilne są konkretne. Widzimy znaczną koncentrację pakietów w obrąbie końca ciągu głównego o I=1 oraz A=0. Natomiast nie ma koncentracji pakietów w obrębie końca ciągu głównego I=0 oraz A=1. Grupa pakietów stabilnych zgodnie z zasadą stabilne abstrakcyjne SAP (sekcja 3.3.3) powinna być jak najbardziej abstrakcyjna. Tracąc abstrakcyjność znacznie utrudniamy rozszerzalność tych pakietów. Okazało się również, że pakiet, który ma wartości I=0 oraz A=1 (wartości nie możliwe do uzyskania) jest pusty i nie zawiera żadnych klas.

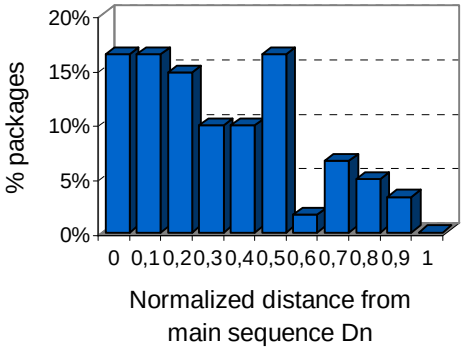
Z kolei Rysunek 22 prezentuje histogram odległości od ciągu głównego. Widzimy wyraźnie, że nie jest tutaj spełniony postulat Martina, aby pakiety znajdowały się możliwie blisko ciągu głównego. Ponad 16% pakietów ma wartość metryki Dn > 0,5. Abstrakcyjność pakietów nie jest więc dobrze balansowana z ich stabilnością, a liczba klas abstrakcyjnych i konkretnych nie jest w dobrej proporcji do powiązań odśrodkowych i dośrodkowych pakietów.

Tabela 18



Rysunek 21. Metryka I oraz A na tle ciągu głównego. Wyraźna, zalecana koncentracja pakietów w niestabilnym i konkretnym obszarze wykresu. Brak koncentracji pakietów w obszarze abstrakcyjnym i stabilnym oznacza łamanie zasady SAP.

Tabela 18 prezentuje najwyższe wartości metryki Dn według programu RefactorIT.



Rysunek 22. Metryka Dn. 45% pakietów ma metrykę Dn większą niż 0,3 co oznacza, że abstrakcyjność pakietów nie jest dobrze balansowana z ich stabilnością.

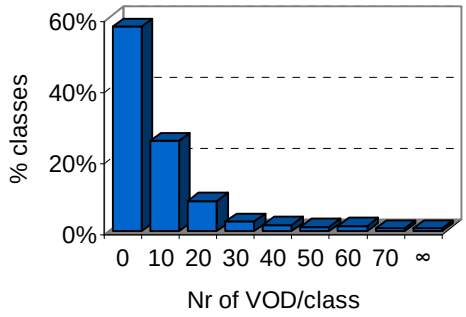
Tabela 18. Najwyższe wartości metryki Dn dla 5 pakietów. Są to głównie pakiety stabilne a nie są abstrakcyjne.

Pakiety	Dn
ew.widget.client	0.857
core.cmdtemplate	0.833
experiment.util	0.778
ew.config.service.tab	0.75
executorstests	0.75

6.5 Inne metryki powiązane z prawami i zasadami programowania obiektowego

Metryki te zostały szczegółowo opisane w sekcji 4.2.4.

Metryka VOD. Metryka została uzyskana przy pomocy PMD - sekcja 5.2.10 (PMD wspiera VOD od wersji 5.0, niestety wersja ta nie jest jeszcze dostępna jako wtyczka dla Maven'a lub Eclipse'a, używano PMD z wiersza poleceń). Prawie 42% klas w analizowanym systemie łamie prawo Demeter. Większość wielokrotnie, co pokazuje Rysunek 23. Można wysnuć wniosek, że autorzy pisanego oprogramowania nie starali się przestrzegać prawa Demeter. Warto o nim pamiętać, gdyż jego stosowanie zmniejsza ilość powiązań między obiektami, przenosi odpowiedzialność wywołania metod do ich właściciela. Tym samym łatwiejsza staje się pielęgnowalność.



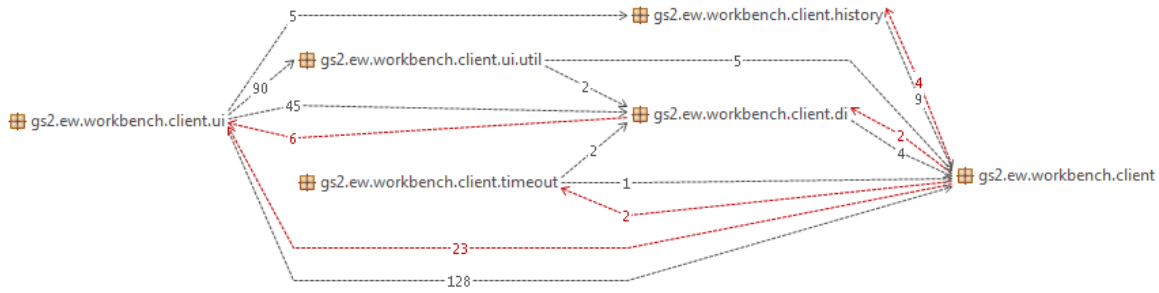
Rysunek 23. Histogram liczby VOD na klasę. 42% klas łamie prawo Demeter, co m.in. zwiększa liczbę powiązań między klasami.

Tabela 19. Największa liczba VOD. Prawo Demeter łamane jest głównie przez klasy o dużej ilości powiązań do zewnątrz.

Klasa	Liczba VOD
ew/webgui/client/WebGuiProcessor.java	141
ew/workbench/client/ui/SnippetPanel.java	99
`sshexecutor/ModifiedSCPClient.java`	79
ew/workbench/client/ui/MainSitePanel.java	67
core/execution/ExperimentSessionBase.java	64

Tangled i CYC. Metryka Tangled realizuje zasadę ADP acyklicznych zależności pomiędzy pakietami lub większymi modułami. W formie zaprezentowanej w tej pracy jest wyliczana w programie Stan. Stan nie oferuje w przypadku tej metryki histogramu wartości dla poszczególnych pakietów. Nawigując po projekcie można podejrzeć wartości dla poszczególnych pakietów. Co ważniejsze na diagramie zależności na czerwono od razu wskazywane są krawędzie MFS, a więc krawędzie o najniższej wadze pozwalające rozbić splot. Po kliknięciu na krawędź można podejrzeć wszystkie zależności dedykowane do refaktoryzacji.

Przykład splotu można już zaobserwować na najwyższym poziomie organizacji naszego systemu w prezentowanym już w pracy diagramie zależności - Rysunek 5. Obserwujemy tam splot pomiędzy modułami `provanance` i `core`. Wartość metryki Tangled wynosi tutaj 0,84% (stosunek wagi krawędzi MFS do wagi wszystkich krawędzi na diagramie). Dodatkowo w raporcie generowanym przez Stan możemy podejrzeć wszystkie sploty. Dostępne są widoki z modułami, jak również z płaskich struktur pakietów. Rysunek 24 przedstawia najbardziej skomplikowaną sieć splotów wychwyconych przez program Stan dla płaskiej struktury pakietów. W sumie narzędzie Sonar znalazło 18 cykli pomiędzy pakietami.

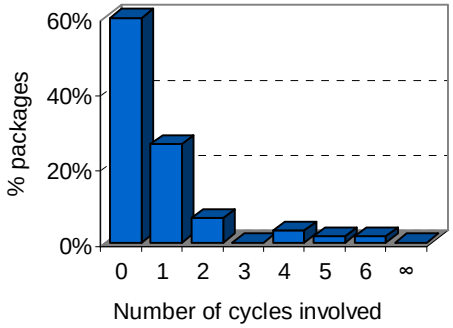


Rysunek 24. Przykład splotów z płaskiej struktury pakietów dla 6 wybranych pakietów GS2 (najgorszy przypadek w całym GS2). W sumie widzimy na rysunku 5 splotów, dla których krawędzie MFS zostały oznaczone kolorem czerwonym.

Ponieważ rekomendacja jest taka, aby likwidować wszystkie sploty (albo dzięki zasadzie odwróconych zależności DIP albo poprzez tworzenie nowych pakietów) warto znać skalę tego zjawiska w systemie. Z pomocą może tu przyjść program RefactorIT, który implementuje inną metrykę powiązaną z zasadą ADP. Metryka CYC wyliczana jest dla pakietu i informuje w ilu cyklach uczestniczy rozpatrywany pakiet. Rysunek 25 pokazuje, że 40% pakietów jest zaangażowanych w cykle, a 8 nawet więcej niż w jeden. Tabela 20 pokazuje z którymi pakietami będzie największy problem jeśli chodzi o ich testowanie, redystrybucję , czy pielęgnowalność ponieważ na te aspekty ma wpływ niestosowanie się do zasady ADP.

Metryka DIPM. Rysunek 26 pokazuje wartości metryki DIPM dla klas i interfejsów, które zależą od innych klas i interfejsów (nie wszystkie klasy zależą od innych, ok. 25% jest samowystarczalna). Metryka ta pokazuje jaki procent z tych zależności to zależności od typów abstrakcyjnych i interfejsów. Zgodnie z zasadą DIP wartości tej metryki powinny być jak najwyższe i bliskie jeden. Stosując rekomendację RefactorIT wartości tej metryki powinny być wyższe niż 0,3. Widzimy, że nawet przy tak liberalnym podejściu około połowa

klas nie spełnia tego warunku. Generalnie można powiedzieć, że brakuje warstwy abstrakcyjnej między klasami. Może to skutkować problemami w późniejszej rozszerzalności takich klas i użycia w innych kontekstach. Ponadto zależność klas wyższej warstwy od klas niższej warstwy może się przełożyć na problemy w tych pierwszych na wypadek zmiany tych drugich.

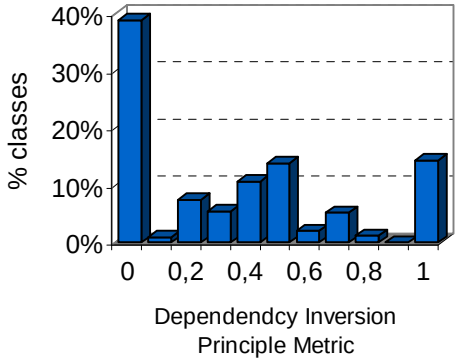


Rysunek 25. Metryka CYC. 40% pakietów uczestniczy przynajmniej w jednym cyklu.

Tabela 20. Najwyższe wartości metryki CYC. Wymienione pakiety uczestniczą w od 2 do 6 cykli. Dla wszystkich zalecane jest rozbicie splotów.

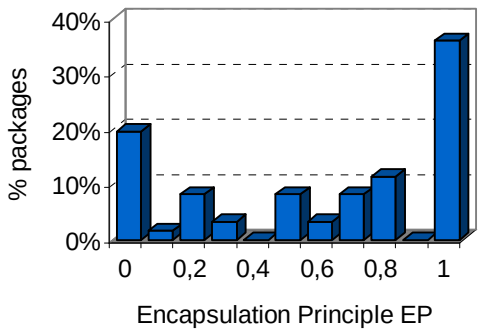
Pakiety	CYC
ew.base.client	6
ew.workbench.client	4
ew.workbench.client.di	4
core.execution	2
ew.workbench.client.history	2

Pytanie oczywiście, czy rozpatrywane klasy będą kiedykolwiek rozszerzane o nowe funkcjonalności, albo czy klasy niższego poziomu modyfikowane. Jeśli nie, to wtedy programista nie odniesie korzyści ze stosowania zasady DIP.



Rysunek 26. Metryka DIPM dla klas, które zależą od innych klas – 75% klas całości klas GS2. Około połowa tych klas nie osiąga rekomendacji RefactorIT, aby wartość DIPM była wyższa niż 0,3. Oznacza to brak warstwy abstrakcyjnej w zależnościach od innych klas.

Metryka EP. Rysunek 27 prezentuje histogram metryki EP wygenerowanej na bazie danych przygotowanych przez RefactorIT. Ponad 50% procent pakietów ma procent klas używanych na zewnątrz w wartości większej niż 60% (próg zdefiniowany przez RefactorIT). Po przyjrzeniu się pakietom, które mają wartość metryki EP = 1 okazało się, że są to pakiety, które jednocześnie mają niską wartość metryki I – niestabilności, czyli są stabilne. Pakiety mające niską wartość metryki I, mają większość powiązań do wewnątrz (inaczej mówiąc klasy z tego pakietu są używane na zewnątrz). Jeśli dołożymy do tego postulat Martina aby pakiety stabilne były abstrakcyjne, wtedy mamy dokładnie sytuację, że EP = 1. Typy abstrakcyjne mają tylko sens, kiedy mają klasy, które je implementują. Z kolei sytuacja, gdy metryka I ma wartość równą 1 daje nam dokładnie EP = 0. Także bazując na tezach Martina (pakiety powinny znajdować się na końcach ciągu głównego sekcja 4.2.3) powinniśmy mieć wartości EP albo bardzo niskie albo bardzo wysokie. W jakimś stopniu oddaje to Rysunek 27. Wydaje się, że metryka EP mogłaby mieć zastosowanie dla pakietu, które ma do wykonania określoną funkcjonalność, ale aby ją wykonać potrzebuje dużej liczby klas specjalistycznych. Metrykę EP należy więc traktować ze szczególną ostrożnością i korzystać z niej z krytycznym podejściem. Należy tutaj wspomnieć, że metrykę tę odnaleziono w programie RefactorIT, natomiast nie znaleziono odwołań do teoretycznej analizy jej zasadności.



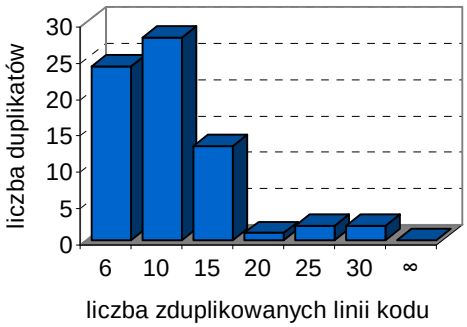
Rysunek 27. Metryka EP. Widać koncentrację pakietów o wysokiej i niskiej wartości EP, co jest zgodne z zasadą SAP.

6.6 Zduplikowany kod

Metryka zduplikowanego kodu została opisana w sekcji 4.3.1.

Zduplikowany kod znaleziono przy pomocy programu Simian (sekcja 5.2.6), który był używany z linii poleceń. Jako wartość progową przyjęto domyślną wartość 6 linii kodu, które

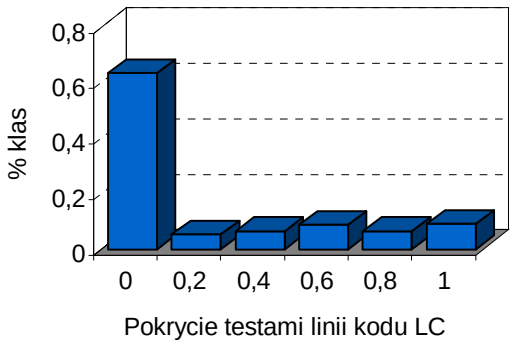
muszą się powtórzyć w dwóch plikach, aby zgłoszony był duplikat. W sumie wykryto 71 duplikatów, a całkowita wartość zduplikowanych LOC wyniosła 1337. Oznacza to, że ok. 4,7% kodu w rozważanym systemie to kod zduplikowany (porównywano wszystkie 7 modułów między sobą), co w świetle badań Baxter'a jest dobrym rezultatem [12]. Rysunek 28 prezentuje rozkład liczby duplikatów w stosunku do liczby zduplikowanych linii kodu które są w nich zawarte. Maksymalny duplikat jaki znaleziono miał 30 linii kodu.



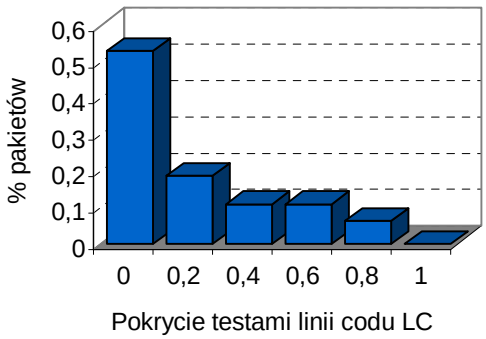
Rysunek 28. Duplikaty. 60% znalezionych duplikatów ma poniżej 15 zduplikowanych linii kodu.

6.7 Pokrycie testami

Metryka pokrycia testami LC została opisane w sekcji 4.3.2. i obliczona przy użyciu programu Cobertura (sekcja 4.3.2), który był używany jako wtyczka do Maven'a. Rysunek 30 i 31 pokazuje pokrycie testami linii kodu odpowiednio dla klas oraz pakietów. Zarówno w pierwszym jak i drugim przypadku rezultat jest podobny: odpowiednio 60% klas i 50% pakietów nie jest testowanych przy użyciu testów jednostkowych. Średnia wynosi odpowiednio 20% dla klas i 15% dla pakietów. Po sprawdzeniu pokrycia testami linii kodu w całym systemie okazało się, że rezultat wynosi 17%. Nie jest to wynik zadowalający, jeśli brać pod uwagę rekomendację Harolda z IBM, który uważa, że testować należy nawet najprostsze metody i osiągać LC na poziomie 90% [37]. Zgodnie z rekomendacją Harolda uzupełnianie testów jednostkowych należy rozpocząć od najgorzej przetestowanych modułów.



Rysunek 29. Metryka LC dla klas. 60% klas nie była testowana przez testy jednostkowe.



Rysunek 30. Metryka LC dla pakietów. 50% pakietów nie było testowanych przy użyciu testów jednostkowych.

6.8 Metryki błędów, czytelności i dobrych praktyk języka Java

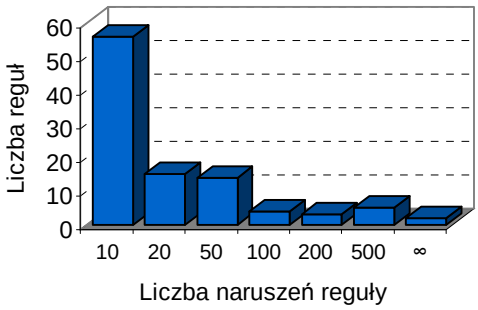
Opis tych metryk znaleźć można w sekcji 4.4.

Metryka GP. Metryka powstała w oparciu o program PMD 5.2.10 wersja 5.0. Program używany był z wiersza poleceń. Do metryki dobrych praktyk wybrano zestawy testów, które w sumie składały się ze 199 przypadków testowych. PMD wykryło w analizowanym systemie naruszenia dla 99 przypadków testowych, czyli dla połowy wszystkich dostępnych testów. W sumie naruszeń było 7129. Rysunek 31 pokazuje liczbę reguł w zależności od ilości naruszeń tych reguł. Dla 13 przypadków testowych odnotowano większą liczbę naruszeń niż 50. W Tabeli 21 pokazano reguły o największej ilości naruszeń. Najwięcej naruszeń mają następujące reguły:

- MethodArgumentCouldBeFinal – argument metody nie jest zmieniany, dlatego może być zadeklarowany jako `final`.
- LocalVariableCouldBeFinal – lokalna zmienna nie jest zmieniana może być deklarowana jako `final`.

Z powyższych dwóch reguł widać już pewne wady oceniania dobrych praktyk na bazie programów typu PMD. Po pierwsze argument metody, który nie jest zmieniany może się powtarzać w wielu metodach w rozpatrywanej klasie zgodnie z zasadą spójności klasy. Naruszenie może być więc raportowana kilkukrotnie dla tego samego argumentu. Ponadto druga wymieniona tutaj reguła niejako duplikuję tę pierwszą.

Bardzo trudno ocenić w oparciu o zaprezentowane dane, w jakim stopniu dobre praktyki w pisaniu kodu były stosowane w badanym systemie. Wymagałoby to przebadania innych wzorcowych systemów i porównania z badanym. Narzędzia PMD należy używać przede wszystkim w trakcie pisania kodu i na bieżąco implementować jego zalecenia. Używanie PMD do analizy gotowego projektu jest uciążliwe.



Rysunek 31. Liczba naruszeń reguł GP. W całym systemie GS2 naruszanych jest 99 reguł z wybranych zestawów PMD. 60% z tych reguł łamanych jest mniej niż 20 razy.

Tabela 21. Reguły GP o największej liczbie naruszeń. Dwie pierwsze pozycje stanowiące prawię połowę naruszeń GP w całym GS2 dotyczą pól w klasie, które mogłyby mieć modyfikator final, ponieważ nie są zmieniane.

Reguła	Liczba naruszeń
MethodArgumentCouldBeFinal	2485
LocalVariableCouldBeFinal	966
FieldDeclarationsShouldBeAtStartOfClass	485
DataflowAnomalyAnalysis	429
DefaultPackage	400

Metryka PB. Metryka ta powstała w oparciu o program Findbugs w wersji 2.0.1 (sekcja 5.2.8) . Sama idea pomiaru została zaprezentowana w rozdziale 4.4. Używano programu z wiersza poleceń gdyż tak najłatwiej było uzyskać kompletny raport dla GS2. Sprawdzanych było 285 reguł za wyłączeniem reguł złych praktyk i reguł wydajnościowych. Znaleziono 184 naruszenia reguł w tym 60 o wysokim priorytecie, 45 o średnim i 79 o niskim. Nie będzie tutaj prezentowany histogram naruszeń ponieważ raport jest tak skonstruowany, że nie da się sensownie posegregować naruszeń. Wizualnie wychwycono jednak 3 najczęściej pojawiające się naruszenia:

- 51 naruszeń reguły o priorytecie wysokim: DM_DEFAULT_ENCODING – wywołanie metody konwertującej byte na String, albo String na byte z założeniem, że użyta jest odpowiednia platforma kodująca.
- 19 naruszeń o priorytecie niskim: REC_CATCH_EXCEPTION – wyłapywanie wyjątku, który nie jest wyrzucany – może wyłapywać też wyjątki RuntimeException i tym samym ukrywać potencjalne błędy.

- 9 naruszeń reguły o priorytecie średnim: EI_EXPOSE_REP: metody zwracające referencję do zmiennych pól jakiegoś obiektu eksponują jego wewnętrzną reprezentację – może prowadzić do niekontrolowanych zmian, czasem lepiej jest zwrócić kopię obiektu (w GS2 naruszenie dotyczy metod związanych z datami).

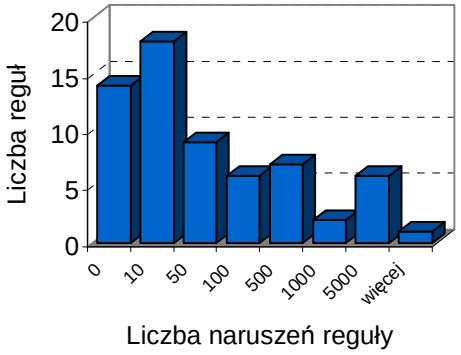
Poza tym wiele reguł jest naruszonych po kilka razy. Oznacza to, że z 285 reguł łamana jest ich mniejsza część. Dodatkowo należy pamiętać, że błędy zgłaszane przez Findbugs są jedynie ostrzeżeniami i nie muszą oznaczać błędu w rzeczywistości. Jak piszą autorzy, Findbugs może raportować do 50% błędnych ostrzeżeń [52]. Na stronie Findbugs możemy również zapoznać się z historycznym raportem platformy Eclipse-3.3M7 mającej 1745 kLOC dla Findbugs w wersji 1.2.1. Znaleziono 3730 dla błędów o wysokim i średnim priorytecie, czyli średnio 2,13 błędu/kLOC. W przypadku GS2 mamy 5,05 błędu/Kloc, przy czym zestawienie to należy traktować poglądowo, gdyż od wersji 1.2.1 do 2.0.1 dodano wiele nowych reguł (w najnowszym wydaniu dodano np. 24 testy).

Metryka R. W celu policzenia metryki czytelności użyto programu Checkstyle (sekcja 5.2.9). Raport generowany był przy użyciu Maven'a. Nie brano pod uwagę wszystkich reguł dostępnych w Checkstyle, a jedynie 63, które są domyślnie ustawione w Checkstyle jako standard firmy Sun (można dodawać reguły, bądź modyfikować przy użyciu pliku konfiguracyjnego sun_checks.xml). W sumie Checkstyle odnotował 49984 naruszenia. Rysunek 32 prezentuje histogram liczby naruszeń reguł. Około połowa reguł naruszona jest mniej niż 10 razy. Natomiast Tabela 22 pokazuje reguły z największą liczbą naruszeń. I tak dwie pierwsze reguły stanowią 70% wszystkich naruszeń, są to:

- FileTabCharacter – raportowane jest naruszenie kiedy w tekście znaleziony zostaje znak tabulatora '\t'. Kod przestaje być czytelny jeśli inny programista ma ustawioną inną ilość znaków dla tabulatora.
- LineLenght – naruszenie raportowane jeśli liczba znaków w linii przekracza 80.

Podobnie jak w przypadku metryk GP i PB, tak i w przypadku metryki czytelności bardzo ciężko jest oceniać ogólną czytelność systemu stosując Checkstyle. Narzędzia tego należy używać równolegle z pisaniem kodu. Dodatkowo Checkstyle stawia jeszcze wyzwanie odpowiedniego skonfigurowania reguł. Odnosząc się do przypadku reguły LineLenght ktoś może przecież chcieć zdefiniować maksymalną dopuszczalną ilość znaków na 120. Można też wyłączyć niektóre standardowe reguły, albo dodać znacznie więcej reguł ze zbioru ok. 250 reguł oferowanych przez Checkstyle. Pytanie jedynie, czy warto poświęcać tyle czasu, żeby

osiągnąć „ładny”, bardziej czytelny kod? Wydaje się, że korzyści nie są współmierne do ilości czasu, który trzeba na to poświęcić.



Rysunek 32. Metryka R - histogram liczby naruszeń reguł. ¼ reguł nie jest łamana, kolejna ¼ łamana jest mniej niż 10 razy. 9 reguł z 63 łamanych jest ponad 1000 razy.

Tabela 22. 5 reguł o największej liczbie naruszeń metryki R. Dwie reguły stanowią 70% wszystkich naruszeń metryki R.

Reguła	Liczba naruszeń
FileTabCharacter	31142
LineLength	4032
JavadocMethod	2845
RegexpSingleline	2671
FinalParameters	2610

6.9 Podsumowanie

Rozdział szósty poświęcono na omówienie wartości metryk zebranych przy pomocy narzędzi: Stan, RefactorIT, Sonar, Simian, PMD, Findbugs i Checkstyle dla systemu GS2. Metryki zliczeń i powiązań pokazały, że GS2 napisany jest tak aby rozmiar artefaktów był jak najniższy. 80% wartości takich metryk jak LOC dla klas, NOF, TLC, CC utrzymywanych jest znacznie poniżej poziomu zalecanego do refaktoryzacji przez narzędzie Stan.

Podobny trend dało się zauważyć w przypadku pierwszych dwóch metryk z zestawu CK – WMC oraz RFC. Obydwie związane są z ilością metod w klasie. W zestawieniu z pomiarami tych metryk dla obiektowych projektów z NASA [27] okazało się, że GS2 ma wartości średnio dwukrotnie mniejsze. Dla metryki CBO uzyskano podobne rezultaty jak dla projektów NASA. Jeśli chodzi o przekraczanie wartości zalecanych do refaktoryzacji to jedynie po 4 klasy przekraczają próg zdefiniowany przez NASA dla metryk WMC oraz CBO, a dla metryki RFC żadna. Metryki dotyczące dziedziczenia DIT i NOC pokazały, że dziedziczenie wykorzystywane jest głównie poprzez użycie zewnętrznych bibliotek natomiast wewnątrz systemu klasy są rzadko rozszerzane z wykorzystaniem mechanizmu dziedziczenia

(jedynie 6% klas ma swych bezpośrednich potomków). Metryka LCOM1 pokazała niepokojące wartości. Jednakże problemy ze spójnością nie zostały potwierdzone przez metrykę LCOM4. Jedynie ok. 7% klas można by podzielić na mniejsze. Nie zaprezentowano metryk z zestawu MOOD z racji braku dostępnego narzędzia *open source*, które by ten zestaw implementowało.

Metryki dotyczące pakietów z zestawu Martina pokazały, że GS2 ma generalnie dobrze zorganizowane pakiety. Metryka Ce w 80% nie przekracza progu 20 powiązań do zewnątrz. Oznacza to, że użycie klas z zewnętrznych pakietów jest utrzymana na względnie niskim poziomie. Natomiast metryka Ca ma wartości znacznie niższe od zalecanych do przeglądu w programie RefactorIT, co oznacza, że pakiety głównie „świadczące usługi" innym pakietom nie są nadmiernie używane w innych pakietach. Wykryto natomiast nie stosowanie się do zasady SAP. Pakiety, które są stabilne nie są jednocześnie abstrakcyjne, co może powodować problemy z ich rozszerzalnością.

W 42% klas wykryto łamanie prawa Demeter. W wielu nawet kilkadziesiąt razy, co zwiększa liczbę powiązań między klasami. Wykryto również, że 40% pakietów zaangażowana jest w cykle, czyli łamana jest zasada ADP. Cykle należy rozbijać stosując zasadę DIP lub poprzez tworzenie nowych pakietów. Okazało się również, że RefactorIT podaje błędną rekomendację dla metryki EP, gdyż kłóci się ona z zasadą SAP.

W całym GS2 wykryto 4,7% zduplikowanych linii kodu co jest rezultatem porównywalnym z innymi dojrzałymi systemami.

Pokrycie testami jednostkowymi jest na niskim poziomie i wynosi średnio 20% dla klas i 15% dla pakietów.

Jeśli chodzi o metryki dobrych praktyk GP, potencjalnych błędów PB oraz czytelności R to odnotowano trudności związane z używaniem ich do oceny jakości całego systemu. Narzędzia które implementują reguły – testy dla tych metryk są tak skonstruowane aby wdrażać ich zalecenia w trakcie pisania kodu. Sporo trudności nastręcza wygenerowanie histogramu liczby naruszeń reguł, a w przypadku raportu generowanego przez Findbugs jest to wręcz niemożliwe. Jeśli chodzi o metrykę GP to połowa z ponad 7000 naruszeń zgłoszonych przez PMD dotyczyła możliwości dodania modyfikatora `final` do pól klas, które nigdy nie są zmieniane. Naruszenia dla metryki R dotyczyły w 70% używania znaku tabulatora oraz zbyt długich linii kodu. Natomiast najwięcej naruszeń dla metryki PB (około 20%) dotyczyło użycia metody konwertującej String na byte lub byte na String.

7 Wnioski

Zaprezentowane w niniejszej pracy metryki analizy statycznej oprogramowania w dużym stopniu mogą się przyczynić do utrzymania wysokiej jakości pisanych programów. Są one szczególnie przydatne w przypadku kiedy należy rozwijać zastane oprogramowanie, a nie ma możliwości stworzenia systemu od początku.

7.1 Obszary mierzone przy udziale metryk

Omówione w rozdziale 3 obszary mierzone przy udziale metryk badających kod źródłowy pokazują różnorodność zagadnień, które można poddać kontroli i ocenie jakości. Szczególnie cenne wydaje się być badanie paradygmatów programowania zorientowanego obiektowo. Bardzo przydatna jest też możliwość weryfikacji praw i zasad, które powstały wraz z rozwojem obiektowych języków. Przy użyciu metryk bardzo dużo możemy powiedzieć o jakości klas i pakietów oraz powiązań między klasami lub pakietami. Natomiast metryki nie dadzą nam oceny architektury tworzonego systemu w sensie specyficznych odpowiedzialności poszczególnych modułów. Brakuje też opracowań, które poruszałyby zagadnienie używania metryk w kontekście stosowania wzorców projektowych.

7.2 Metryki w analizie statycznej oprogramowania

Definiowanie metryk dotyczących oprogramowania zorientowanego obiektowo jest zadaniem nietrywialnym. Wymagane jest głębokie zrozumienie paradygmatów obiektowych czy też idei stojących za prawami, czy zasadami podlegającymi pomiarom. Jak pokazują prace Mayer'a i Hall'a nawet najbardziej popularne zestawy metryk obiektowych MOOD czy CK nie są pozbawione błędów wynikających z niepoprawnego zrozumienia podstaw teoretycznych [23, 34]. Problematyczne są też różnice między językami oprogramowania. Niektóre metryki powinny zostać zredefiniowane w zależności od używanego języka (np. mechanizm wielodziedziczenia powinien być uwzględniany w metrykach DIT i NOC z zestawu CK). Pomimo niedociągnięć w definicjach metryki z zestawu MOOD i CK są wystarczająco dobre, aby używać ich do wykrywania klas szczególnie narażonych na występowanie w nich błędów (korelacja między wysokimi wartościami metryki, a liczbą błędów wykrywanych w badanych typach) [32]. Cenne są też badania potwierdzające korelację wysokich wartości metryk z zestawu CK z niską produktywnością programistów, czy kosztów czasowych napisania kodu [33]. Zestawy MOOD i CK doczekały się najwięcej publikacji i opracowań wyników, przez co najłatwiej korzystać z tych dwóch zestawów. Możliwe jest porównywanie otrzymanych historycznie rezultatów z wartościami metryk ze swoich projektów, a także lepiej zrozumieć zagrożenia wynikające z niestosowania się do zaleceń wynikających z interpretacji tych metryk.

Niestety bardzo często z definicji, czy badań teoretycznych nie wynikają jasne wskazówki co do wartości preferowanych metryk. Bardziej jest mowa o tym, czy pożądane są wartości niskie, czy wysokie. W celu uzyskania jakiegoś poglądu na temat wartości zalecanych konieczne jest zbadanie systemów uznawanych za dobre jakościowo, a następnie porównywanie wartości w nowych systemach, z tymi już przebadanymi. Właśnie tak postąpiono w przypadku opracowań zalecanych wartości dla zestawu MOOD i CK [20, 26, 27]. Należy jednak brać poprawkę na to, że badane systemy mogą mieć różną wielkość, specjalizację, czy stopień skomplikowania, przez co do takich rekomendacji należy podchodzić z ostrożnością.

Metryki MOOD i CK skupiają się głównie na jakości klas i relacji między klasami. Jeśli chodzi o wyższy poziom abstrakcji dotyczący organizacji pakietów, prym wiedzie Robert Martin, który zaproponował wiele zasad, a także kilka metryk, które implementują te zasady [6, 35]. Pomimo solidnego rozeznania teoretycznego brakuje jednak w literaturze ich

praktycznej weryfikacji co pozwoliłoby na korzystanie z zaleceń wynikających z interpretacji tych metryk z większą pewnością. W rozdziale 4 omawiającym metryki opisano również kilka metryk rozmiaru i proste metryki złożoności znalezione w różnych narzędziach.

Często pewne rekomendacje co do wartości progowych można znaleźć w narzędziach je implementujących (szczególnie dla metryk zliczeń). Jednakże autorzy narzędzi nie popierają takich rekomendacji badaniami ani teoretycznymi, ani praktycznymi, przez co należy je traktować z rezerwą.

7.3 Narzędzia open source implementujące metryki analizy statycznej dla języka Java

Zestaw narzędzi implementujących metryki zaprezentowany w rozdziale 5.2 pokazuje, że jeśli chodzi o język Java to nie ma za wiele narzędzi *open source* wspierających obliczanie metryk w tym języku. Można więc wysnuć wniosek, że metryki analizy statycznej nie cieszą się specjalną popularnością wśród programistów. Szczególnie brakuje tutaj implementacji zestawu MOOD, który jest solidnie opisany w literaturze i jest w pewnym sensie komplementarny do zestawu CK (uwzględnia polimorfizm i ukrycie informacji). Omówione narzędzia oferują dość szeroką gamę metryk zliczeń, które nie zawsze wnoszą wartość dodaną. Narzędzia te nie są też wolne od błędów implementacyjnych (np. metryka DIT w Stanie nie uwzględnia wszystkich poziomów dziedziczenia aż do klasy Object). Pewne niespójności wynikają również z niedokładnych definicji metryk (np. Stan do wyliczania metryk z zestawu CK uwzględnia interfejsy, a RefactorIT nie).

Warto przy okazji rozpoczynania pracy z nowym narzędziem sprawdzić na wybranych przykładach, czy metryka wyliczana jest w sposób prawidłowy. Dodatkowo krytycznie należy też pochodzić do nowych propozycji metryk napotykanych w narzędziach, które nie mają odniesień do opracowań teoretycznych. Tak jest np. z metryką EP zdefiniowaną w RefactorIT, która miała wyrażać zasadę enkapsulacji pakietów. W trakcie analizy w rozdziale 6.5 okazało się, że sugerowane wartości metryki łamią zasadę stabilne abstrakcyjne SAP zdefiniowaną przez Martina (omówiona w sekcji 3.3.3).

W codziennej pracy najwygodniejszym narzędziem do analizy statycznej oprogramowania okazał się być program Stan. Jego dużą zaletą jest też możliwość generowania raportu o ogólnej jakości całego sytemu (generowanie histogramów oraz listy klas i pakietów z największą ilością naruszeń wybranych metryk). Stan oferuje też bardzo pomocne

wizualizacje zależności wewnątrz typów i pomiędzy nimi na wszystkich poziomach abstrakcji. Stanowi to cenne wsparcie dla wartości metryk, ułatwia podjęcie decyzji o refaktoryzcaji. Wadą jest brak możliwości wygenerowania raportu z wartościami dla wszystkich typów, tak aby można było prowadzić dalszą, samodzielną analizę. Taką możliwość daje RefactorIT w dużym stopniu używany w tej pracy do oceny systemu GS2. RefactorIT wspiera też wiele metryk, których brak jest w Stanie. Z kolei najprężniej rozwijającym się projektem *open source* jest Sonar. W tej pracy wykorzystany jedynie do policzenia metryki LCOM4. Natomiast wydaje się to być najbardziej przyszłościowy projekt, w dodatku umożliwia też analizę innych obiektowych języków programowania.

Narzędzia, które opierają się o zestaw reguł – testów czyli Findbugs, Checkstyle i PMD niezbyt nadają się do oceny całości systemów pod kątem potencjalnych błędów, stosowania się do dobrych praktyk, czy czytelności kodu. Należy je stosować przede wszystkim w trakcie pisania kodu i na bieżąco weryfikować zgłaszane naruszenia. Odkładanie sprawdzania kodu na ostatni moment może spowodować zasypaniem tysiącami naruszeń. W przypadku Checkstyle dodatkowo warto redefiniować reguły tak, aby skupić się jedynie na naruszeniach istotnych w danym projekcie, czy organizacji - szczególnie dla metryk dotyczących czytelności kodu.

7.4 Zastosowanie metryk w praktyce

Jeśli chodzi o analizę wyników metryk to najlepszą metodą prezentacji wartości metryk są histogramy. Dają one wgląd w rozkład wartości metryki w całym systemie, czyli pozwalają ocenić mierzony obszar kompleksowo. Jeszcze ważniejsza jest identyfikacja klas i pakietów, które przyjmują wartości metryk spoza zalecanego zakresu. Są to klasy i pakiety, które należy przejrzeć pod kątem zgłaszanego naruszenia i ewentualnie przeprojektować.

Wspomniano już, że wartości progowe metryk są ustalane na bazie wybranych systemów, które klasyfikowane są jako dobre jakościowo i rzadko rekomendacje wartości wynikają z definicji. Należy więc te wartości traktować jedynie za punkt odniesienia ponieważ w zależności od specyfiki systemu progi mogą się zmieniać. Z tego też powodu większość narzędzi pozwala na modyfikowanie ustalonych progów. Nawet jeśli zalecany próg wartości jakiejś metryki jest przekroczony, nie oznacza to, że trzeba koniecznie refaktoryzować kod. Naruszenia wartości metryk stanowią raczej ostrzeżenie o zwiększonym ryzyku wystąpienia błędu, czy o potencjalnych problemach z późniejszym utrzymaniem.

7.5 Wnioski ogólne

Najważniejsza rekomendacja jeśli chodzi o użycie metryk jest taka, aby używać ich od początkowej fazy tworzenia oprogramowania. Dzięki metrykom można wychwycić złe tendencje w rozwoju oprogramowania i wyeliminować je na wczesnym etapie, przez co koszt refaktoryzacji nie będzie wysoki. Ponadto metryki pomagają w lepszym zrozumieniu złożoności pisanego systemu. Złożoność rozumiana jest tutaj jako liczba i charakter zależności pomiędzy typami. Każdy programista powinien dążyć do jak najlepszego zrozumienia zależności występujących w systemie, a metryki mogą być w tym bardzo pomocne.

Większość użytej w niniejszej pracy literatury dotyczącej metryk obiektowych powstała w latach dziewięćdziesiątych. Pokazuje to spadek zainteresowania metrykami obiektowymi. Wydaje się jednak, że temat ten nie jest wyczerpany i potrzebne są dalsze badania, a może propozycje nowych metryk lub przynajmniej udoskonalenie tych już istniejących. Ciągle mało przeprowadzono badań potwierdzających użyteczność metryk, a więc ich powiązań z takimi zewnętrznymi cechami jakości oprogramowania jak: pielęgnowalność, funkcjonalność, niezawodność, użyteczność, efektywność, czy przenośność.

8 Podsumowanie

W pracy magisterskiej przeprowadzono studium metryk statycznej analizy oprogramowania. Omówiono obszary, które podlegają pomiarom przez metryki. Zaprezentowano przegląd metryk analizy statycznej, omówiono publikacje naukowe weryfikujące zaprezentowane wcześniej propozycje metryk. Przedstawiono zestaw narzędzi dla języka Java, które wspierają metryki analizy statycznej, podano rekomendacje do stosowania najlepszych narzędzi oraz wskazano ich mocne i słabe strony. Zastosowano omówione metryki i narzędzia w praktyce i przeprowadzono audyt kodu dla części projektu GridSpace2 padając rekomendacje dla przeglądu kodu w miejscach naruszeń zalecanych wartości metryk. Na koniec zaprezentowano najważniejsze wnioski, które jednocześnie stanowią ocenę przydatności metryk w inżynierii jakości oprogramowania. Tym samym zostały zrealizowane wszystkie cele stawiane tej pracy magisterskiej.

Praca może stanowić pewnego rodzaju podręcznik do nauki metryk analizy statycznej oprogramowania w ujęciu praktycznym. Zawiera ona przegląd wszystkich najważniejszych metryk, przykłady użycia, a także odniesienia do praktycznych rezultatów omówionych w publikacjach naukowych. Stosowanie metryk w praktyce nie jest sprawą prostą i przede wszystkim wymaga doświadczenia, wyrobienia sobie własnych praktyk. Literatura rzadko wypowiada się jednoznacznie na temat wartości zalecanych, często bazują one na porównaniach z projektami ocenionymi jako dobre jakościowo. Idąc dalej, nawet duże naruszenie wartości jakiejś metryki w wybranym module nie od razu oznacza konieczność refaktoryzacji. Stanowi raczej ostrzeżenie dla projektanta o możliwości wystąpienia potencjalnych problemów.

Metryki statycznej analizy oprogramowania mogą być cennym wsparciem w procesie tworzenia aplikacji. Stanowią rdzeń statycznej analizy oprogramowania. Dają możliwość oceny kodu na różnych poziomach abstrakcji: od metody poprzez klasy i pakiety, aż po cały tworzony system. Pozwalają badać paradygmaty oprogramowania zorientowanego obiektowo, które przekładają się na cechy zewnętrzne oprogramowania takie jak: funkcjonalność, niezawodność, przenośność, efektywność, czy pielęgnowalność.

Słowniczek pojęć

Statyczna analiza oprogramowania - jest to analiza oprogramowania przeprowadzana bez uruchomienia programu. Analizie podlega kod źródłowy lub pliki binarne generowane przez kompilator.

Metryka - funkcja odwzorowującą jednostkę oprogramowania w wartość liczbową. Ta wyliczona wartość jest interpretowalna jako stopień spełnienia pewnej własności jakości jednostki oprogramowania.

Jakość oprogramowania - „zgodność z wymaganiami" albo „przydatność do użytku" Najpopularniejsze jednak ujęcie to powiązanie jakości z brakiem defektów.

GridSpace2 - platforma służąca do tworzenia naukowych aplikacji obliczeniowych tzw. eksperymentów *in silico* z użyciem zasobów udostępnianych przez polskie i europejskie centra obliczeniowe. Do głównych cech platformy można zaliczyć:

- możliwość budowania eksperymentów z wielu istniejących kodów obliczeniowych i aplikacji w ramach zadań lokalnych i gridowych,
- dostęp poprzez wygodny interfejs webowy z dowolnego komputera niezależnie od długości działania uruchamianych eksperymentów,
- dostępność rejestru skonfigurowanych aplikacji i interpreterów na różnych sajtach obliczeniowych,
- zarządzanie plikowymi rezultatami obliczeń,
- możliwość publikacji utworzonych eksperymentów wraz z towarzyszącymi danymi.

Literatura

[1] Przykład funkcji o niskiej wartości CC.
http://pl.wikipedia.org/wiki/Metryka_oprogramowania

[2] Strona projektu GridSpace2 https://gs2.plgrid.pl/

[3] Strona pracowników Distributed Computing Environments Team Cyfronetu
http://dice.cyfronet.pl

[4] G. Booch i inni, *Object-Oriented Analysis and Design with Applications*, Third Editon, Addison-Wesley, 2007

[5] K. J. Lieberherr, I. M. Holland, A. J. Riel *Object-Oriented Programming: An Objective Sense of Style*, 1988

[6] R. Martin, *Design Principles and Design Patterns,* 2000

[7] T. DeMarco, *Structured Analysis and System Specification. ,* Prentice Hall, 1988

[8] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 1988

[9] B. Liskov, *Data Abstraction and Hierarchy.* SIGPLAN Notices, 1988.

[10] Przykład zasady DIP http://www.oodesign.com/dependency-inversion-principle.html

[11] B. Walter, *Testowanie jednostkowe,* Instytut Informatyki, Politechnika Poznańska

[12] I. D. Baxter, I. D. Yahin, I. D. Moura, I. D. Sant'Anna i I. D. Bier, *Clone Detection using Abstract Syntax Trees*, WCRE, 1995

[13] B. Pietrzak, J. Nawrocki, B. Walter, *Automatyczne wykrywanie duplikatów w kodzie programów*

[14] K. Kontogiannis, R. Demorei, E. Merlo, M. Galler, M. Bernstein, *Pattern Matching for Clone and Concept Detection,* Automated Software Engineering, 3, 77–108, 1996

[15] http://www.kclee.de/clemens/java/javancss/

[16] T. McCabe, *A Software Complexity Measure*, IEEE Transactions on Software Enginnering vol. SE-2, no. 4, December 1976, s 308-320

[17] Artykuł objaśniający metryki FAT i TANGLED
http://www.headwaysoftware.com/documents/XS-MeasurementFramework.pdf

[18] Przykład funkcji o niskiej wartości CC.
http://pl.wikipedia.org/wiki/Metryka_oprogramowania

[19] F. B. e Abreu, R. Carapuca *Object-Oriented Software Engineering:Measuring and Controlling the Development Process,* "4th Int. Conf. on Software Quality", McLean, VA, USA, 3-5 October 1994

[20] F. B. e Abreu, M. Goulão, *Toward the Design Quality Evaluation of Object-Oriented Software Systems,* Conference on Software Quality, Austin, Texas, 23 to 26 October 1995,
[21] B.Walter, *Metryki obiektowe* , Instytut Informatyki, Politechnika Poznańska
[22] R. Harrison, S. J. Counsell, R. V. Nithi, *An Evaluation of the MOOD Set of Object-Oriented Software Metrics*, IEEE Transactions on Software Engineering, tom 24, nr 6, 1998.
[23] T. Mayer, T. Hall, *Measuring OO Systems: A Critical Analysis of the MOOD Metric*, TOOLS '99 Proceedings of the Technology of Object-Oriented Languages and Systems, 1999.
[24] S. R. Chidamber and C. F. Kemerer, *A Metrics Suite for Object Oriented Design,* IEEE Transactions on Software Engineering, vol. 20, no. 6, June 1994, s. 476-493
[25] L.Etzkorn, C.Davis, W.Li *A Statistical Comparison of Various Definitions of the LCOM Metric*, The University of Alabama in Huntsville, 1997
[26] S.H. Kan, *Metryki i modele w inżynierii jakości oprogramowania*, PWN 2006, s. 355-383
[27] L. Rosenberg, *Applying and Interpreting Object Oriented Metrics,* Software Assurance Technology Center (SATC) w NASA.
[28] http://www.aivosto.com/project/help/pm-oo-cohesion.html za Henderson-Sellers, B, L, Constantine and I, Graham , *Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design),* Object-Oriented Systems, 3(3), pp143-158, 1996.
[29] W. Li, S. Henry. *Maintenance Metrics for the Object Oriented Paradigm*. In Proc. 1st Int. Software Metrics Symp., Los Alamitos,CA, May 21-22 1993, IEEE Comp. Soc. Press, 1993, 52-60.
[30] M.Hitz, B.Montazeri *Measuring Coupling and Cohesion In Object-Oriented Systems,* Institut für Angewandte Informatik und Systemanalyse, University of Vienna, 1995
[31] Przykład spójnej i niespójnej klasy według LCOM4 za http://www.aivosto.com/project/help/pm-oo-cohesion.html
[32] V.Basili, L.Briand i W.Melo, *A Validation of Object-Oriented Design Metrics as Quality Indicators.* IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996.
[33] S. R. Chidamber, D. P. Darcy, C. F. Kemerer, *Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis*, IEEE Transactions on Software Engineering, tom 24, nr 8, 1998.
[34] T. Mayer, T. Hall, *A Critical Analysis of Current OO Design Metric*, Software Quality Control, tom 8, nr 2, 1999, Centre for Systems and Software Engineering (CSSE)

[35] R. Martin, *OO Design Quality Metrics An analysis of dependencies*, 1994,
[36] B. Pietrzak, J. Nawrocki, B. Wolter, *Automatyczne wykrywanie duplikatów w kodzie programów*, Instytut Informatyki, Politechnika Poznańska i TiP Spółka z o.o.
[37] Artykuł dotyczący pokrycia testami przy użyciu programu Cobertura http://www.ibm.com/developerworks/java/library/j-cobertura/
[38] Artykuł porównujący narzędzia PMD, Checkstyle i Findbugs: http://www.sonarsource.org/what-makes-checkstyle-pmd-findbugs-and-macker-complementary/
[39] Strona platformy Eclipse. http://www.eclipse.org/
[40] Strona wtyczek Eclipse'a *http://marketplace.eclips e.org*
[41] Strona projektu Apache Maven. http://maven.apache.org
[42] Strona projektu Apache Ant *http://ant.apache.org*
[43] Martin Fowler, *Continuous Integration*, http://www.martinfowler.com/articles/continuousIntegration.html
[44] Strona programu Apache Continuum *http://continuum.apache.org*
[45] Strona programu STAN. http://stan4j.com
[46] Strona programu RefactorIT http://sourceforge.net/projects/refactorit/
[47] Strona platformy Sonar http://www.sonarsource.org/
[48] Strona progamu JDepend. http://clarkware.com/software/JDepend.html
[49] Strona programu CKJM. http://www.spinellis.gr/sw/ckjm/
[50] Strona programu Simian. http://www.harukizaemon.com/simian/
[51] Strona programu Cobertura. http://cobertura.sourceforge.net/
[52] Strona programu Findbugs. http://findbugs.sourceforge.net/
[53] Strona programu Checkstyle. http://checkstyle.sourceforge.net/
[54] Strona programu PMD. http://pmd.sourceforge.net/
[55] Strona programu Esential metrics firmy Powersoft http://www.powersoftware.com/em
[56] Strona infrastrukury PL-Grid. http://www.plgrid.pl/
[57] Infrastruktura PL-Grid – wprowadzenie. http://www.plgrid.pl/wprowadzenie/