Politechnika Wrocławska

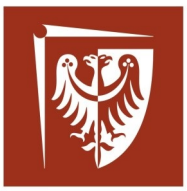
Wydział Elektroniki

Testowanie mutacyjne – optymalizacja procesu i praktyczne zastosowania

Rozprawa doktorska

Autor:
mgr Michał Mnich

Promotor:
dr hab. Adam Roman



8 kwietnia 2021

Podziękowania

Szczególne podziękowania za współpracę i pomoc w wykonaniu tej pracy przekazuję dr. Adamowi Romanowi.

Chciałbym także podziękować rodzicom Jadwidze i Włodzimierzowi, całej rodzinie oraz Przyjaciołom, bez których nie byłbym tym kim jestem.

Spis treści

Spis rysunków

Spis tablic

Spis pseudokodów

Rozdział 1

Wstęp

1.1 Cel pracy

Testowanie mutacyjne jest uznawane za jedną z najefektywniejszych metod testowania (w sposób pośredni) kodu oprogramowania. Niestety, wadą tego podejścia jest duża jego pracochłonność: przeprowadzenie testowania mutacyjnego wymaga zazwyczaj dużej ilości zasobów i trwa bardzo długo. Dlatego też kluczowe jest, aby proces testowania mutacyjnego można było uczynić jak najefektywniejszym, wprowadzając do niego różnorakie metody zwiększania jego efektywności.

Niniejsza praca dotyczy zagadnień optymalizacji procesu mutacyjnego oraz jego zastosowania w praktycznych procesach inżynierii oprogramowania. W pracy zaproponowano szereg mechanizmów optymalizacyjnych, mających na celu zmniejszenie czasu trwania procesu mutacji kodu.

Pierwszy mechanizm dotyczy redukcji liczby mutantów na podstawie analizy zmian w kodzie pomiędzy różnymi wersjami oprogramowania. Drugi oparty jest na podejściu bayesowskim do ustalania prawdopodobieństwa generacji mutantów z określonej grupy operatorów mutacyjnych. W pracy opisany został również model generowania wielu mutantów w jednej kompilacji wraz z eksperymentami weryfikującymi czy przy użyciu modelu następuje poprawa wydajności procesu testowania mutacyjnego. Ta część pracy zawiera także teoretyczne uzasadnienie tego, które mutanty mogą, a które nie mogą istnieć w jednej kompilacji dla programów w paradygmacie obiektowym.

Na potrzeby empirycznej weryfikacji skuteczności zaproponowanych mechanizmów optymalizacyjnych zaprojektowano, stworzono i opisano w pracy samoadaptacyjny, automatyczny i skalowalny system służący do przeprowadzania testowania mutacyjnego. System jest rozproszonym klastrem obliczeniowym z zaimplementowaną metodą optymalizacji procesu mutacji i testowania. Według najlepszej wiedzy autora, metody optymalizacji zastosowane w systemie nie były dotychczas wykorzystywane na potrzeby testów mutacyjnych.

Praca opisuje również teoretyczny model, wraz z doświadczalną jego weryfikacją w praktyce, zastosowania podejścia mutacyjnego w procesach zwinnego wytwarzania oprogramowania. Opisana metodyka dotyczy włączenia procesu mutacyjnego w model TDD (Test-Driven Development).

1.2 Tezy badawcze

W świetle opisanych powyżej celów tezy pracy można sformułować następująco:

Teza 1. Efektywność testowania mutacyjnego nie zmniejsza się znacząco, jeśli w nowej wersji oprogramowania mutanty generowane są wyłącznie dla kodu nowego bądź zmodyfikowanego.

Teza 2. Zmiana rozkładu prawdopodobieństwa losowania operatorów mutacyjnych, dokonana przy użyciu podejścia bayesowskiego, może być wykorzystana do istotnego zmniejszenia liczby mutantów przy dopuszczalnej niewielkiej utracie efektywności procesu.

Teza 3. Wzbogacenie podejścia Test-Driven Development o testowanie mutacyjne znacząco podnosi charakterystyki jakościowe kodu i zwiększa niezależność testowania poprzez obniżenie zjawiska stronniczości (ang. bias) autora testowanego kodu.

Teza 4. Podejście polegające na wprowadzeniu wielu mutacji do jednej kompilacji ma swoje ograniczenia i nie każdy operator mutacyjny może współistnieć z wszystkimi innymi.

Teza 5. Podejście polegające na wprowadzeniu wielu mutacji do jednej kompilacji dla języków, w których modyfikacja kodu pośredniego nie jest możliwa, może znacząco przyspieszyć proces testowanie mutacyjnego.

1.3 Struktura pracy

Rozdziały 2, 3, 4 są rozdziałami wstępnymi. Zawierają rys historyczny, przegląd najnowszych badań, modele teoretyczne związane z inżynierią oprogramowania i testowaniem mutacyjnym oraz inne znane z literatury metody optymalizacji testowania. Kolejne rozdziały (5, 6, 7, 8) prezentują oryginalne wyniki autora pracy. Rozdział 9 opisuje architekturę platformy do testów mutacyjnych wykonanej na potrzeby badań zawartych w tej rozprawie. Ostatni rozdział (10) jest rozdziałem zawierającym podsumowanie pracy. Szczegółowa zawartość rozdziałów wygląda następująco:

- Rozdział 2 zawiera rys historyczny i przegląd literatury dotyczącej testowania oprogramowania. Wskazuje na konieczność stosowania coraz to nowszych technik testowania programowania.
- Rozdział 3 zawiera rys historyczny i przegląd literatury dotyczącej specyficznego obszaru testowania oprogramowania, jakim jest testowanie mutacyjne.
- Rozdział 4 opisuje znane w literaturze podejścia do optymalizacji procesu testowania mutacyjnego, a także krótki przegląd propozycji modeli optymalizacyjnych, które zostały zdefiniowane w rozdziałach późniejszych.
- Rozdział 5 zawiera oryginalne rozważania opisujące warunki, w jakich niektóre operatory mutacyjne mogą lub nie mogą współistnieć w jednym kodzie. Na tej podstawie został opracowany model teoretyczny wraz z jego implementacją, realizujący proces testowania mutacyjnego oparty o generację wielu mutantów w jednej kompilacji.

- W rozdziale 6 wprowadzono model optymalizacyjny (tzw. 'Mutation Churn Model'), którego celem jest redukcja liczby mutantów w kodzie niepowodująca znaczącego spadku efektywności procesu testowania mutacyjnego. Omówione zostały również wyniki badań empirycznych weryfikujących skuteczność zdefiniowanego modelu.

- Rozdział 7 wprowadza drugi model optymalizacyjny, wykorzystujący techniki probabilistyczne, mianowicie model redukcji liczby mutantów oparty na podejściu bayesowskim. Tak jak w rozdziale poprzednim, również i tu omówiono wyniki badań empirycznych nad skutecznością tego modelu w zastosowaniach praktycznych.

- Rozdział 8 zawiera nowatorską propozycję metodyki wytwarzania oprogramowania opartą na mutacyjnym testowaniu oprogramowania. Metodyka ta jest modyfikacją podejścia TDD poprzez wzbogacenie jej o krok analizy mutacyjnej. W rozdziale tym zawarte są wyniki eksperymentów potwierdzających skuteczność zaproponowanego podejścia.

- Rozdział 9 zawiera opis systemu S.A.M. (Symultanic Automatic Mutation System) — autorskiej aplikacji stworzonej przez autora niniejszej dysertacji. System został zaimplementowany w oparciu o zadany model teoretyczny [2]. System S.A.M. jest narzędziem służącym do automatyzowania i zarządzania procesem testowania mutacyjnego.

- Rozdział 10 zawiera wnioski oraz podsumowanie.

Rozdział 2

Testowanie oprogramowania

Testowanie oprogramowania jest procesem należącym do dyscypliny inżynierii oprogramowania, którego głównym celem jest weryfikacja poprawności rozwiązań dostarczanych poprzez oprogramowanie, a także poprawa jakości samego kodu. Za działający poprawnie program można uznać taki, dla którego zostały spełnione wszystkie wcześniej ustalone wymagania [3]. Definicja ta może się jednak wydać kontrowersyjna, ponieważ często wiele wymagań nie jest definiowanych jawnie (np. większość wymagań niefunkcjonalnych), a także nie uwzględnia kwestii implementacji nadmiarowych funkcjonalności. W niniejszej pracy ograniczamy się jednak do sytuacji, w których poprawność dotyczy wyłącznie spełnienia wymagań, które zawsze będą dobrze określone.

Proces testowania jest czasochłonnym fragmentem cyklu życia oprogramowania, a co za tym idzie także kosztownym. Dla złożonych i nietrywialnych systemów dogłębne przetestowanie kodu jest zwyczajnie niemożliwe [4]. W związku z powyższym testowanie powinno być przeprowadzane rozsądnie i z umiarem, to znaczy tak, ażeby nie przekroczyć dostępnego budżetu czy czasu przeznaczonego na cały proces kontroli jakości, jednocześnie znajdując możliwie jak najwięcej defektów. Trzeba pamiętać, że testowanie jest również czymś innym niż debugowanie, które zajmuje się lokalizacją, analizą oraz eliminacją defektów powodujących awarie oprogramowania [5] wykryte w procesie testowania. Natomiast w testowaniu mamy do czynienia z kontrolą bądź analizą poprawności oprogramowania w sensie zgodności z założeniami modelu (np. specyfikacji wymagań). Niezgodności z modelem wynikające z błędów w implementacji mogą, ale nie zawsze bezwzględnie muszą doprowadzić do awarii oprogramowania. Wzorując się na danych zwartych w [4] jako główne cele testowania oprogramowania można wyróżnić:

- weryfikację zgodności abstrakcji, jaką jest program, z założeniami, które mają być przezeń realizowane;
- walidację, czy praca oprogramowania prowadzi do założonych wyników;
- kontrolę jakości, mającą na celu ustalenie, czy praca oprogramowania i sam proces wytwórczy są realizowane w satysfakcjonujący sposób.

Ponadto, testowanie oprogramowania jest procesem, dzięki któremu można nabrać zaufania co do

funkcjonowania stworzonego oprogramowania. Przykładem poziomu testowania, który dobrze służy temu celowi, może być testowanie akceptacyjne wykonywane przez klienta.

W kolejnych podrozdziałach opiszemy najważniejsze elementy procesu testowania oprogramowania. W przypadku testowania istnieje bardzo bogata literatura przedmiotu, dlatego poniższy opis jest ograniczony do niezbędnego minimum pozwalającego zrozumieć umiejscowienie w procesie testowym metod, technik i podejść omówionych w niniejszej rozprawie.

2.1 Definicja testowania

Na potrzeby niniejszej pracy przyjmijmy następującą definicję testowania.

Definicja 2.1 (Testowanie)**.** Testowanie oprogramowania to proces służący do oceny aplikacji z zamiarem ustalenia, czy opracowane oprogramowanie spełnia określone wymagania.

Proces ten pozwala także na zidentyfikowanie i wyeliminowanie wad programu. Jednolita, dokładna definicja testowania oprogramowania w literaturze nie występuje. Wielu autorów definiuje ten proces na swój własny sposób. Na przykład, zgodnie z ideą opisaną przez Myersa [6] testowanie to „proces wykonywania programu lub systemu z intencją znajdowania w nim błędów". Obecnie na proces ten patrzy się znacznie szerzej, bowiem testowaniu może podlegać każdy fragment procesu wytwórczego oprogramowania – od diagramów użycia aż po przejrzystość interfejsów. Ponadto, definicja Myersa wyklucza olbrzymi dział testowania, jakim jest testowanie statyczne, w którym produkty pracy testowane są bez uruchamiania kodu (np. przeglądy wymagań, inspekcje kodu, analiza statyczna).

2.2 Cel testowania

Oprogramowanie, wraz z rozwojem informatyki, stało się coraz bardziej złożone. Jednocześnie, w ostatnich latach ze względów biznesowych następuje olbrzymia presja na wytwórców oprogramowania, aby wciąż skracać tzw. czas do wydania (time-to-market). Powyższe zjawiska w naturalny sposób zaczęły przekładać się na liczbę pomyłek wytwórców aplikacji. Ludzkie błędy powodują wprowadzenie do kodu defektów, a te – jeśli wykonana zostanie linia programu zawierająca defekt – mogą doprowadzić do awarii oprogramowania.

Zdefiniujemy teraz precyzyjnie trzy ważne pojęcia użyte w poprzednim paragrafie. Pojęcia te są istotne, gdyż będziemy się do nich nieustannie odwoływać.

Definicja 2.2 (Błąd [7])**.** Błąd to ludzka pomyłka.

Definicja 2.3 (Defekt [7])**.** Defekt to niedoskonałość bądź usterka w produkcie pracy. Synonim: usterka.

Definicja 2.4 (Awaria [7])**.** Awaria to widoczne na zewnątrz odchylenie modułu lub fragmentu systemu od oczekiwanego, prawidłowego stanu. Awaria jest rezultatem wystąpienia defektu w kodzie.

Liczba defektów w kodzie jest silnie skorelowana z liczbą linii kodu (LOC). Im większa ilość kodu, tym więcej potencjalnych miejsc do popełnienia błędu przez programistę. Teza ta jest dosyć dobrze znana i jest poparta w literaturze między innymi w monografii [8].

W związku z opisanym wyżej rozwojem informatyki oraz dążeniem do maksymalnego skracania czasu cyklu wytwórczego niezwykle trudno (jeśli w ogóle) jest utrzymać stały poziom jakości wytwarzanego oprogramowania. Dlatego w ciągu ostatnich kilkunastu lat coraz większą uwagę przykłada się do zapewniania i kontroli jakości, a w szczególności – do procesu testowania oprogramowania. Testowanie nie jest jednak wytworem ostatnich lat. Pierwsze wzmianki na temat procesu testowania złożonych systemów pochodzą z... początku XIX wieku (sic!). Jedną z takich wzmianek można znaleźć w opisach prac nad maszyną różnicową Charlesa Babbage'a. Jednak na tamtym etapie był to proces bardziej przypominający obecne debugowanie niż testowanie. Tematyka testowania zaczęła się rozwijać wraz z zaproponowaniem przez Glenforda J. Myersa oddzielenia czynności testowania oprogramowania od jego debugowania [9].

Na przestrzeni ostatniego półwiecza na znaczenie i rolę testowanie oprogramowania patrzono w różny sposób. Zmiana perspektywy powodowana była zwiększającą się dojrzałością dyscypliny, jaką jest inżynieria oprogramowania, a w szczególności – inżynieria jakości. Lui [10], opierając się na pracy Gelperina i Hentzla [11] definiuje następujących pięć faz rozwoju testowania:

- Faza pierwsza (przed rokiem 1957) – era debugowania. W tym okresie testowanie nie było odróżniane od debugowania, były to pojęcia tożsame.
- Faza druga (1957-1978) – era dowodzenia poprawności. W tym okresie testowanie i debugowanie są odrębnymi pojęciami, a testowanie polega na dowodzeniu poprawności działania oprogramowania.
- Faza trzecia (1979-1982) – era dowodzenia niepoprawności. W 1979 Myers publikuje słynną monografię [9], w której definiuje testowanie jako „proces wykonywania programu z intencją znajdowania błędów". Testowanie nabiera charakteru „negatywnego", próby „niszczenia" oprogramowania.
- Faza czwarta (1983-1989) – era ewaluacji. W 1983 Institute for Computer Sciences and Technology of the National Bureau of Standards publikuje dokument „Guideline for Lifecycle

Validation, Verification and Testing of Computer Software". W rozumieniu autorów tego dokumentu testowanie jest integralną częścią cyklu życia. Rozróżniono też weryfikację od walidacji.

- Faza piąta (od roku 1990) – era zapobiegania. W 1990 Beizer publikuje drugie wydanie słynnej książki „Software Testing Techniques" [12] w której stawia tezę, iż „tworzenie testów jest jednym z najefektywniejszych sposobów zapobiegania usterkom". Rozszerza więc pojęcie testowania na aspekt zapewniania jakości (zapobiegania wprowadzania defektów do kodu). Praca Beizera uznawana jest za klasyczną pozycję promującą jedną ze standardowych dziś strategii analitycznych, jaką jest testowanie opartego na ryzyku.

Silny rozwój technik związanych z testowaniem oprogramowania napędzany jest poprzez ciągle rosnącą liczbę linii kodu w nowoczesnym oprogramowaniu. Na podstawie danych z [13] widać np., że kod Linuxa 1.0 z 1971 roku zawiera około tysiąca linii kodu, podczas gdy w roku 2015 oprogramowanie Google Internet Services posiada około... dwa miliardy LOC! Wobec tak szybkiego wzrostu złożoności oprogramowania koniecznym staje się stosowanie coraz to bardziej zaawansowanych technik testowania oprogramowania. Obecnie sama liczba linii kodu obejmującego tylko testy staje się już tak duża, że zaczyna pojawiać się coraz większa potrzeba weryfikacji poprawności samych testów w związku z pojawiającą się w nich coraz większą liczbą defektów. Paradoksem obecnej sytuacji jest fakt, że należałoby zacząć pisać testy do testów aby zweryfikować poprawność tych ostatnich. Takie podejście oczywiście nie prowadzi do niczego poza zjawiskiem regresu w nieskończoność. Rozwiązaniem problemu są techniki automatycznej weryfikacji poprawności testów, na przykład metody analizujące poprawność zachowania interfejsów podczas ich testowania. Jednak chyba najbardziej interesującą metodą jest możliwość weryfikacji poprawności testów jednostkowych, czyli tak zwane testowanie mutacyjne. Proces testowania mutacyjnego jest opisany w rozdziale 3.

2.3 Poziomy testów i techniki testowania

Na testowanie można patrzeć z punktu widzenia miejsca procesu wytwórczego, w jakim znajduje się aktualnie wytwarzane oprogramowanie. Tradycyjnie wyróżnia się cztery zasadnicze poziomy testów [4]:

- testy jednostkowe – polegające na testowaniu logicznie wydzielonych jednostek pracy programisty (np. klas, funkcji, modułów) w izolacji od pozostałych jednostek;
- testy integracyjne – polegające na testowaniu interfejsów i interakcji pomiędzy jednostkami programu (np. testy interfejsów, protokołów i innych połączeń między modułami lub systemami);
- testy systemowe – polegające na testowaniu w pełni zintegrowanego systemu, umożliwiające weryfikację procesów biznesowych realizowanych przez oprogramowanie;

- testy akceptacyjne – polegające na walidacji oprogramowania, czyli testowaniu z punktu widzenia użytkownika końcowego, mające na celu upewnienie się, że oprogramowanie spełnia potrzeby klienta.

Tradycyjnie, metody testowania można podzielić na trzy zasadnicze grupy [4]:

- testy czarnoskrzynkowe – testy projektowane są na podstawie zewnętrznego źródła wiedzy o programie (specyfikacji), zaś struktura wewnętrzna programu pozostaje przed testerem ukryta;
- testy białoskrzynkowe – testy projektowane są na podstawie znajomości wewnętrznej struktury testowanego obiektu (np. kodu źródłowego), a wyniki oczekiwane wyprowadzane są ze specyfikacji;
- testy oparte na doświadczeniu – mniej formalne podejście do testowania, wykorzystujące wiedzę, intuicję i doświadczenie testera.

Klasyczne techniki czarnoskrzynkowe wymieniane w literaturze to: technika podziału na klasy równoważności, analiza wartości brzegowych, tablice decyzyjne, testowanie maszyny stanowej, testowanie oparte na przypadkach użycia. Klasyczne techniki białoskrzynkowe to: testowanie instrukcji, decyzji, MC/DC, cała rodzina technik dotyczących testowania ścieżek w grafie przepływu sterowania. Do technik białoskrzynkowych opartych na składni można zaliczyć również testowanie mutacyjne opisane w następnym rozdziale. Klasyczne techniki oparte na doświadczeniu to: zgadywanie błędów, ataki usterkowe, testowanie oparte na liście kontrolnej czy – prawdopodobnie najpopularniejsza technika z tej grupy – testowanie eksploracyjne.

W niniejszej pracy nie omawiamy szczegółowo powyższych technik, gdyż wykracza to poza zakres dysertacji. Techniki te są szczegółowo omówione w literaturze. Dobrym źródłem mogą być np. [4, 9, 12, 14, 15]. W badaniach opisanych w tej pracy wykorzystywać będziemy głównie białoskrzynkowe testy jednostkowe.

2.4 Pokrycie

Większość technik testowania polega na zbudowaniu modelu formalnego oprogramowania i wyróżnienia w nim odpowiednich elementów pokrycia, czyli elementów modelu, które mogą być sprawdzone przy pomocy testów. Przykładem elementu pokrycia może być klasa równoważności w metodzie podziału na klasy równoważności, wartość brzegowa dziedziny w metodzie analizy wartości brzegowych, kolumna tablicy decyzyjnej, przejście w maszynie stanów, instrukcja wykonywalna w grafie przepływu sterowania, wynik decyzji w procesie biznesowym itp. [12].

W celu oceny dokładności testowania przy użyciu konkretnej techniki testowania wykorzystuje się pojęcie pokrycia. Pokrycie jest miarą ilorazową i wyraża stosunek liczby przetestowanych elementów pokrycia do liczby wszystkich możliwych do przetestowania elementów pokrycia. W

niniejszej pracy stosować będziemy dwie podstawowe miary pokrycia dla testów białoskrzynkowych: pokrycie instrukcyjne oraz pokrycie mutacyjne.

Definicja 2.5 (Pokrycie instrukcyjne)**.** Pokrycie instrukcyjne to stosunek liczby instrukcji wykonywalnych pokrytych przez testy do liczby wszystkich instrukcji wykonywalnych w danym kodzie.

Rozważmy prosty przykład ilustrujący pojęcie pokrycia instrukcyjnego. Dany niech będzie następujący kod programu.

Listing 2.1: Przykładowy kod źródłowy

```
int Funkcja(int i) {
1 int x = 0;
2 while (i > x) {
3   if (i > 10)
4       x = x + 7;
    else
5       x = x + 2;
  }
6 x = x + i;
7 return x;
}
```

Dla danych wejściowych $i = -3$ wykonanie `Funkcja(-3)` przejdzie po ścieżce 1, 2, 6, 7, czyli osiągnie pokrycie $4/7 \approx 57\%$. Z kolei dla testu `Funkcja(4)` przepływ sterowania przejdzie po ścieżce 1, 2, 3, 5, 3, 5, 3, 6, 7. Ten test pokryje więc instrukcje 1, 2, 3, 5, 6, 7, czyli osiągnie pokrycie $6/7 \approx 86\%$. Suita testowa złożona z tych dwóch testów łącznie pokrywa wszystkie instrukcje poza instrukcją nr 4, więc osiąga pokrycie ok. 86%. Zauważmy, że aby osiągnąć stuprocentowe pokrycie instrukcyjne, potrzeba co najmniej dwóch testów, ponieważ instrukcje 4 i 5 nie mogą być wykonane w ramach jednego wykonania kodu. Przykładem dwóch testów osiągających 100% pokrycia instrukcyjnego mogą być np. `Funkcja(11)` oraz `Funkcja(5)`.

Pokrycie instrukcyjne jest jedną z najpopularniejszych technik oceny siły testów jednostkowych stosowanych przez programistów. Formalną definicję pokrycia mutacyjnego podamy w dalszej części pracy, po opisaniu procesu testowania mutacyjnego (def. 3.1).

Rozdział 3

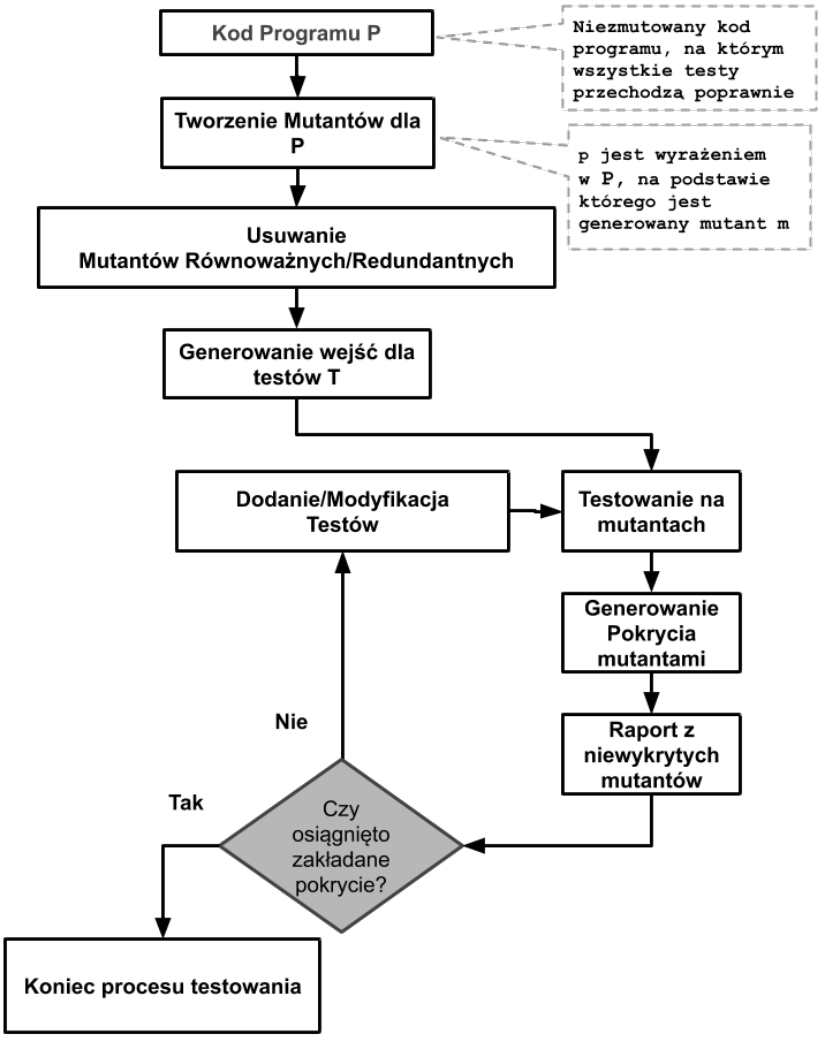
Testowanie mutacyjne

Testowanie mutacyjne jest procesem, którego bezpośrednim celem jest pomiar jakości istniejących testów. Dla zadanego programu generowany jest pewien zbiór modyfikacji kodu. Te modyfikacje nazywa się mutantami. Każdy mutant reprezentuje pewną wadę – zwaną mutacją – sztucznie wstrzykniętą do kodu przez tak zwany operator mutacyjny. Mutacja symuluje rzeczywistą pomyłkę, którą mógł popełnić programista pisząc kod. Wszystkie mutanty wraz z oryginalnym programem są testowane przez ten sam zestaw testów. Mutant jest wykrywany (inaczej: zabity), jeśli przynajmniej jeden test daje inny wynik dla mutanta niż dla oryginalnego programu. W zależności od stosunku wykrytych mutantów (który definiuje się jako tzw. miarę pokrycia mutacyjnego) wnioskuje się o jakości, tzn. o mocy danych testów. Jeśli bowiem żaden test nie był w stanie wykryć danego mutanta, oznacza to, że testy są słabe i powinno się dodać co najmniej jeszcze jeden test, który będzie w stanie wykryć tego niewykrytego mutanta.

Testy mutacyjne mają dość długą historię. Praca Saywarda, DeMillo i Liptona z 1978 roku [16] jest uważana za tę, która wprowadziła zagadnienie mutacji do inżynierii oprogramowania. Podstawy teorii testowania mutacyjnego przedstawiono dokładnie w podręczniku Ammanna i Offutta [14]. Z teoretycznego punktu widzenia testowanie mutacyjne lokuje się w obszarze testów opartych na usterkach (ang. fault-based testing). Możliwość wstrzykiwania usterek implikuje konieczność dostępu do testowanego kodu bądź innej struktury, na której przeprowadzamy proces mutacyjny. Testowanie mutacyjne można zatem zaklasyfikować do grupy technik białoskrzynkowych. Może być również zaklasyfikowane jako testowanie oparte na składni (ang. syntax-based testing), ze względu na stosowanie operatorów mutacyjnych modyfikujących treść kodu źródłowego. Testowanie mutacyjne może być stosowane na wszystkich poziomach testów, przy czym najczęściej spotyka się je w testach jednostkowych. Na wyższych poziomach testowania — integracyjnym, systemowym czy nawet akceptacyjnym — modelem, który podlega mutacjom, może być np. graf wywołań, projekt architektury czy specyfikacja wymagań zapisana w formalnym języku, umożliwiającym automatyczne stosowanie operatorów mutacyjnych.

3.1 Proces testowanie mutacyjnego

Proces testowania mutacyjnego został pierwotnie opracowany przez DeMillo w 1978 roku [16]. Proces ten został zaprezentowany na rysunku 3.1.



Rysunek 3.1: Proces testowania mutacyjnego

Danymi wejściowymi do procesu testowania mutacyjnego są:

- kod źródłowy oryginalnego programu P, nazwany *Testowanym Systemem* (TS),
- powiązany z kodem źródłowym zestaw testów T.

Tester tworzy wiele kopii TS, a następnie w każdym egzemplarzu wprowadza małe zmiany składniowe – *mutacje*. Zmodyfikowane kopie oryginalnego kodu nazywa się *mutantami*. Każda zmiana powinna reprezentować symulowaną pomyłkę programisty. Wszystkie zmiany wprowadzane są za pomocą predefiniowanych *operatorów mutacyjnych*, które przekształcają kod zgodnie ze zdefiniowanymi przez nie same regułami składniowymi. Konieczne jest, aby po takiej zmianie mutant mógł zostać skompilowany i uruchomiony. Więcej informacji o operatorach mutacyjnych znajduje się w dziale 3.4. Następnym krokiem procesu jest przetestowanie zarówno oryginalnego

programu, jak i wszystkich mutantów przy użyciu danego zestawu testów. Przez M oznaczać będziemy kod źródłowy poddany mutacji, a przez $t \in T$ — konkretny test. Proces uruchomienia testów na danym programie P oznacza się jako $P(t)$, a na mutancie M jako $M(t)$, gdzie $t \in T$. Jeśli dla danego M istnieje t takie, że $P(t) \neq M(t)$, to mówimy, że mutant M został wykryty lub *zabity* przez test t. Oznacza to, że dany zestaw testów jest w stanie wykryć wstrzyknięty defekt reprezentowany przez M. Jeśli jednak żaden test nie był w stanie wykryć M, czyli $\forall t \in T\ M(t) = P(t)$, oznacza to, że dany zestaw testów nie był w stanie wykryć tej usterki. W takim przypadku tester powinien dodać nowy test do T (ewentualnie zmodyfikować jakiś istniejący już test), który byłby w stanie zabić M. Odsetek zabitych mutantów nazywa się *wynikiem mutacji* lub pokryciem mutacyjnym (ang. *mutation score*).

Definicja 3.1 (Pokrycie mutacyjne)**.** Pokrycie mutacyjne to stosunek liczby mutantów zabitych przez dany zestaw testów do liczby wszystkich wygenerowanych przez operatory mutacyjne mutantów.

Testowanie mutacyjne może być zatem uznane tak naprawdę za proces *testowania testów*, a nie samego programu. W istocie rzeczy tak właśnie jest: testowanie mutacyjne weryfikuje jakość danego zestawu testów T. Testowanie mutacyjne ma oczywiście wpływ na jakość TS, ale jest on pośredni. Wpływ ten polega na tym, że każda sesja testów mutacyjnych może wskazać konieczność dodania nowych testów, zdolnych wykryć pozostałe przy życiu mutanty. Dodanie przez dewelopera odpowiednich testów zwiększa zdolność zestawu testów do wykrywania nowych defektów.

Jeżeli jakiś mutant jest wykrywany przez wszystkie albo prawie wszystkie testy, to istnieje możliwość, że jest w pewnym sensie „łatwy" do wykrycia, zatem możemy uznać, że jego wykorzystanie nie pozwala w istotny sposób zwiększyć jakości naszej suity testowej (patrz definicja 3.2). Metoda zmniejszenia liczby takich właśnie mutantów trywialnych, oparta na podejściu bayesowskim, została opisana w rozdziale 7.

Definicja 3.2 (Mutant trywialny)**.** Mutant trywialny to mutant zawsze wykrywany przez większość testów na przestrzeni wszystkich iteracji projektu. O trywialności danego mutanta możemy mówić tylko względem jednego projektu.

Jest oczywiste, że skuteczność procesu mutacji w znacznym stopniu zależy od rodzaju użytych operatorów mutacji. Tak, jak w przypadku mutantów, tak i w przypadku operatorów mutacyjnych możemy wyróżnić operatory trywialne[1], które skutkują tworzeniem mutantów trywialnych, a co

[1]Mówiąc o operatorach że są trywialne ma się na myśli sytuację, w której w procesie mutacji kodu użyto niewielu operatorów oraz były one upraszczane tylko do jakichś swoich podzbiorów. Przykładowo, jeśli wprowadzono operator, który ma na celu mutowanie warunków logicznych (OR, AND, NOT, XOR itd.), a użyto tylko mutacji zmieniającej OR na AND, to jest to sytuacja, gdzie zbiór operatorów mutacyjnych jest trywialny.

za tym idzie —- prawie wszystkie testy będą w stanie je wykryć. W kontekście wykrywalności mutantów generowanych przez konkretny operator mutacyjny możemy mówić o *efektywności danego operatora.*

Definicja 3.3 (Efektywność operatora mutacyjnego)**.** Efektywność operatora mutacyjnego wyrażona jest jako odsetek wygenerowanych mutantów, które nie zostały wykryte przez testy.

Jeżeli każdy mutant wygenerowany przez dany operator jest wykrywany, to znaczy, że efektywność danego operatora jest równa zeru. Im większy współczynnik wykryć, tym mniejsza efektywność operatora. Efektywność operatora mierzona jest względem danego programu, dla którego generowane są mutanty. Utrzymywanie mało efektywnych operatorów mutacyjnych może przyczyniać się do powstawania mutantów trywialnych.

Operatory mutacyjne mogą także definiować sposób, w jaki testy wykrywają mutanty. Stosując operację mutacji kodu zgodną z wybranym typem operatora mutacyjnego, wprowadzamy zmiany w kodzie symulujące defekty, które mogą spowodować „niewielką" awarię polegającą tylko na zmianie wewnętrznego stanu programu. Defekt może także generować „silną" awarię, której efektem jest nieprawidłowe wyjście[2] lub wyjątek. W związku z tym, na podstawie testu, który nie przeszedł, jest możliwe zdefiniowanie sposobu, w jaki mutant został wykryty przez testy. Definicję tę przedstawił w swojej książce Offutt [14] (str. 178-179). Autor podaje dwa sposoby, na jakie mutant może zostać wykryty przez testy: silny i słaby. O wykryciu mutanta przez test w sposób silny mówi się, gdy mając mutanta M dla programu P i test t wynik $P(t)$ jest różny od wyniku $M(t)$. Natomiast mając test t oraz mutanta M, który modyfikuje miejsce l w programie P powiemy, że t zabija M w sposób słaby, jeśli stan wewnętrzny programu P wykonanego na t jest odmienny od stanu M tuż po wykonaniu miejsca l.

Wspomniane zostało, że mutacje generowane poprzez operatory mutacyjne powinny być symulacją rzeczywistych błędów popełnianych przez programistów. W literaturze zaproponowano wiele typów operatorów mutacyjnych. Ich przegląd został opisany w podrozdziale 3.4. Informację na ten temat można również znaleźć w pracy [17]. Operatory mutacyjne mogą generować nie tylko proste mutanty syntaktyczne (np. zmieniając jeden operator relacyjny na inny), ale także mutanty odzwierciedlające typy błędów charakterystyczne dla programowania obiektowego [18].

3.2 Wady i zalety testowania mutacyjnego

Testy mutacyjne są uważane za jedną z najskuteczniejszych metod testowania. Ta teza została potwierdzona w sposób badawczy między innymi w pracy Kintis et al. [19]. Testowanie mutacyjne jest procesem wnoszącym do wytwarzania oprogramowania wiele korzyści, takich jak te opisane poniżej.

[2]Wyjście w tym kontekście jest zbiorem danych zwracanych przez program lub metodę.

- Zwiększenie jakości wytwarzanego kodu. Wynika ono ze zwiększenia jakości pisanych testów, co rzutuje bezpośrednio na zwiększenie zaufania do kodu programu, którego testy poprzez proces testowania mutacyjnego zostały przebadane pod kątem wykrywania dużej liczby defektów oraz potencjalnych problemów.

- Sam kod programu w następstwie testów mutacyjnych może także zostać wzbogacony o dodatkowe zabezpieczenia chroniące przed przejściem oprogramowania w niewłaściwy stan oraz wykrywające dotąd niewykryte przypadki brzegowe mogące dopuścić do bloku kodu bądź metody niewłaściwy zestaw danych.

- Bardzo istotną cechą testowania mutacyjnego jest fakt, że jest to proces całkowicie automatyczny. Wykonanie testów sprowadza się jedynie do skonfigurowania środowiska testowego[3] oraz uruchomienia automatycznych testów na poszczególnych mutantach. Każda kolejna iteracja testów nie wymaga już ponownej konfiguracji. Po uruchomieniu platformy testującej żadna ręczna ingerencja nie jest już wymagana (poza ewentualnym dodawaniem testów mających na celu zabicie niezabitego mutanta).

- Najbardziej kluczową cechą testów mutacyjnych, a zarazem jedną z przyczyn wymyślenia tej techniki jest testowanie testów, które wcześniej było uważane za jeden z trudnych do rozwiązania problemów. W metodykach wytwarzania oprogramowania niebazujących na testach mutacyjnych weryfikacja poprawności testów była trudna do wykonania, ponieważ wymagała ona wytwarzania testów, które testowały testy. Problem związany z tego typu weryfikacją poprawności i siły testów jest następujący: skoro możliwe jest, że w testach jednostkowych występują defekty wynikające z pomyłek programisty, to tak samo w testach testów mogą również wystąpić takie same pomyłki. To doprowadza do konkluzji sugerującej, że potrzebne byłyby także *testy testów testów*, a następnie kolejne poziomy testujące kolejne testy[4]. Problemy z weryfikacją poprawności oraz skuteczności testów sprawiają, że testowanie jest procesem trudnym, mogącym znacznie spowolnić wytwarzanie oprogramowania. Problem ten rozważany jest w wielu miejscach w literaturze przedmiotu (m.in. w pracach [20, 21]).

- Pośredni wpływ testowania mutacyjnego na liczbę testów w programie. Ze względu na to, jak proces testowania mutacyjnego się odbywa (uruchamianie testów na zmutowanym kodzie), można wykryć redundancję w testach, a następnie ją usunąć. Program prowadzony od początku w metodyce używającej testów mutacyjnych powinien się cechować rozsądną liczbą testów.

Niestety, istnieje również kilka problemów dotyczących techniki testowania mutacyjnego:

- Testowanie mutacyjne jest procesem czasochłonnym — zwykle w procesie mutowania generowana jest bardzo duża liczba mutantów, które muszą być testowane zazwyczaj dużą

[3] Jest to proces jednorazowy, wykonywany raz dla danego projektu. W kolejnych iteracjach modyfikuje się jedynie nieznacznie ustawienia konfiguracji, jeśli zajdzie tak potrzeba.

[4] Tego typu *regressus ad infinitum* nie dość, że jest nieskończony, to jeszcze — nawet gdyby „uciąć" go na pewnym poziomie — wymagałby ogromnego nakładu pracy, który w konsekwencji mógłby doprowadzić do całkowitego zablokowania rozwoju aplikacji.

liczbą testów. Wszystkie mutanty muszą zostać skompilowane. Następnie każdy z mutantów musi zostać przetestowany, dopóki jakiś test go nie zabije (lub dopóki wszystkie testy nie zostaną wykonane, a żaden z nich nie będzie w stanie wykryć tego mutanta). Całkowity potrzebny czas do przeprowadzenia sesji mutacyjnej na rozbudowanym pod względem kodu projekcie może być bardzo duży.

- Jeśli mutant wprowadza program w stan niekończącej się pętli[5], nie jest wiadome czy i kiedy należy zatrzymać działanie aplikacji. Możliwym jest bowiem, że mutant nie wprowadził oprogramowania w stan nieskończonej pętli, lecz tylko znacznie wydłużył działanie wykonywania kodu. Dylemat, kiedy przerwać wykonywanie mutanta jest problemem analogicznym do nierozstrzygalnego problemu stopu.

- Niektóre mutanty mogą być równoważne oryginalnemu kodowi (np. mutant zamieniający dwie sąsiednie, niezależne od siebie linie kodu). W ogólności sprawdzenie, czy mutant jest odpowiednikiem oryginalnego programu, jest nierozstrzygalne. Problem stwierdzenia równoważności dwóch programów jest nierozstrzygalny, co łatwo uzasadnić sprowadzając go do problemu stopu. Problem mutantów równoważnych jest szerzej opisany w dziale 3.9.

Oczywiście istnieją metody, które starają się przezwyciężyć te trudności przynajmniej do pewnego stopnia, ale wspomniane powyżej problemy są w ogólności nierozwiązywalne, a jednocześnie zbyt ważne, aby je zignorować.

3.3 Operatory mutacyjne

Większość używanych języków programowania jest opartych na gramatykach bezkontekstowych. Każde słowo należące do języka generowanego tą gramatyką (czyli każdy program) może zostać poddane zmianie w inne słowo z gramatyki także należące do tego języka (w inny program). Reguły opisujące, jak taka zmiana ma wyglądać i przebiegać, są nazywane operatorami mutacyjnymi, których mechanika została już opisana w dziale 3.1. Zgodnie z tym każdy mutant jest definiowany przez jakiś operator mutacyjny. A zatem operator mutacyjny może zostać zdefiniowany w sposób formalny poprzez operacje na *słowach bazowych* języka (patrz def. 3.4).

Definicja 3.4 (słowo bazowe, ang. ground string)**.** Słowo bazowe to słowo lub zestaw słów języka bezkontekstowego, którym jest zbiór wszystkich programów danego języka programowania. Każde słowo bazowe może zostać zmutowane, jeżeli istnieje co najmniej jeden operator mutacyjny mu odpowiadający.

Zgodnie z powyższą definicją operator mutacyjny w sposób formalny może zostać zdefiniowany w sposób następujący (def 3.5).

[5]Obserwując wykonanie programu z zewnątrz, nie jesteśmy w stanie zweryfikować czy mamy do czynienia z pętlą nieskończoną, czy też po prostu z długim wykonaniem mutanta.

Definicja 3.5 (operator mutacyjny)**.** Operator mutacyjny to zestaw syntaktycznych reguł opisujących sposób modyfikacji kodu tak, ażeby powstał mutant. Każdy operator mutacyjny dotyczy jednego typu słów bazowych.

Operatory mutacyjne mogą być definiowane w sposób dowolny, w zależności od potrzeb oraz danego języka. Ciekawy zbiór operatorów mutacyjnych wraz z ich klasyfikacją można znaleźć w monografii Ammanna i Offutta [14].

3.4 Typy operatorów mutacyjnych

Bazując na pracy [14] możemy wyróżnić cztery podstawowe typy operatorów mutacyjnych: operatory mutacyjne generujące mutanty projektowe, operatory mutacyjne generujące mutanty integracyjne, operatory mutacyjne generujące mutanty obiektowe oraz operatory mutacyjne generujące mutanty wykonywane na gramatyce.

3.4.1 Mutanty projektowe

Jak zostało wspomniane na początku tego działu, operatory mutacyjne są zwykle projektowane tak, aby naśladować typowe błędy programistów lub aby zachęcić testerów do przestrzegania typowych heurystyk testowych. Zbiór mutantów projektowych ma na celu symulowanie prostych defektów kodu na poziomie wyrażeń arytmetycznych, inkrementacji, odwołań do zmiennych lub warunków logicznych. Tego typu mutanty są zależne od języka, w jakim mają być wygenerowane. Lista mutantów przedstawionych poniżej może zostać użyta w większości kompilowalnych[6] języków wysokiego poziomu takich jak: C++, Java, C#, a także w dużej części języków skryptowych[7] takich jak: Javascript, Python, czy php.

- **ABS – Absolute Value Insertion**. Każde wyrażenie arytmetyczne jest modyfikowane przez funkcje `abs()`, `negAbs()` i `failOnZero()`. Funkcja `abs()` zwraca bezwzględną wartość wyrażenia, natomiast funkcja `negAbs()` zwraca ujemną wartość bezwzględną. `FailOnZero()` sprawdza, czy wartość wyrażenia to zero. Jeśli tak, mutant zostaje wykryty, w przeciwnym razie wykonanie będzie kontynuowane, a wartość wyrażenia — zwracana. Operator `failOnZero()` został zaprojektowany w szczególności, aby zmusić testera do sprawdzenia zachowania się oprogramowania, gdy jako wyniki operacji pojawią się wartości dodatnie, ujemne albo zera.

  Przykład. Wyrażenie `x = 3 * a;` może zostać zmutowane na następujące sposoby:

  Listing 3.1: Przykład mutantów generowanych przez ABS

[6] Języki kompilowalne to takie, które do wytworzenia działającego programu na podstawie kodu wymagają zewnętrznego narzędzia w postaci kompilatora.

[7] Języki skryptowe nie wymagają do tworzenia wykonywalnego programu kompilatora. Tutaj jego miejsce zajmuje interpreter, który czyta instrukcje kodu linijka po linijce, za każdym razem je wykonując.

```
x = 3 * abs(a);
x = 3 * negAbs(a);
x = 3 * failOnZero(a);
```

- **AOR – Arithmetic Operator Replacement**. Każde wystąpienie jednego z podanych operatorów arytmetycznych `+`, `-`, `/`, `*`, `**` i `%` będzie zastąpione przez inne z tej samej grupy. Może również być usunięte wraz z jednym z operandów. Działanie tego operatora może być wyjaśnione następującym przykładem. Wyrażenie `x = a + b;` może zostać zmutowane na następujące sposoby:

Listing 3.2: Przykład mutantów generowanych przez AOR

```
x = a - b;
x = a * b;
x = a / b;
x = a ** b;
x = a;
x = b;
x = a % b;
```

- **ROR – Relational Operator Replacement**. Każde wystąpienie jednego z operatorów relacyjnych (`<`, `<=`, `>`, `>=`, `=`, `!=`) może być zastąpione przez inne z tej samej grupy, a ponadto całe wyrażenie relacyjne może być zastąpione przez stałe logiczne `False` albo `True`. Działanie tego operatora wyjaśnia następujący przykład. Wyrażenie `if m > n` może zostać zmutowane na następujące sposoby:

Listing 3.3: Przykład mutantów generowanych przez ROR

```
if (m >= n)
if (m < n)
if (m <= n)
if (m == n)
if (m != n)
if (False)
if (True)
```

- **COR – Conditional Operator Replacement**. Każde wystąpienie operatora logicznego (koniunkcji &&, alternatywy ||, zaprzeczenia !) może być zastąpione innym operatorem z tej samej grupy. Wartości całych wyrażeń mogą być zastąpione także przez stałe logiczne `True`, `False`, lub jakąś składową wyrażenia. W zależności od języka operatory logiczne mogą się różnić. Na przykład, wyrażenie `if (a && b)` może zostać zmutowane na następujące sposoby:

Listing 3.4: Przykład mutantów generowanych przez COR

```
if (a || b)
```

```
if (a & b)
if (a | b)
if (a ^ b)
if (false)
if (true)
if (a)
if (b)
```

- **SOR – Shift Operator Replacement**. Każde wystąpienie jednego z operatorów przesunięcia <<, >> i >>> może być zastąpione przez każdy z pozostałych operatorów lub usunięte wraz z jednym z operandów. Przykład. Wyrażenie `x = m « a;` może zostać zmutowane na następujące sposoby:

Listing 3.5: Przykład mutantów generowanych przez SOR

```
x = m >> a$;
x = m >>> a$;
x = m$;
```

- **LOR – Logical Operator Replacement**. Przekształca logiczne operatory bitowe takie jak bitowa koniunkcja & czy bitowa alternatywa | w taki sam sposób jak operator COR. Na przykład, wyrażenie `x = m & n;` może zostać zmutowane na następujące sposoby:

Listing 3.6: Przykład mutantów generowanych przez LOR

```
x = m | n$;
x = m$;
x = n$;
```

- **ASR – Assignment Operator Replacement**. Każde wystąpienie jednego z operatorów przypisania (+ =, − =, ∗ =, / =, % =, & =, | =, =, <<=, >>=, >>>=) może być zastąpione przez każdy z pozostałych operatorów. Na przykład, wyrażenie `x += 3;` może zostać zmutowane na następujące sposoby:

Listing 3.7: Przykład mutantów generowanych przez ASR

```
x -= 3;
x *= 3;
x /= 3;
x %= 3;
x &= 3;
x |= 3;
x ^= 3;
x <<= 3;
x >>= 3;
x >>>= 3;
```

- **UOI – Unary Operator Insertion**. Każdy operator unarny (arytmetyczny +, arytmetyczny -, logiczny !) jest zastępowany przez każdy z pozostałych operatorów. Przykład. Wyrażenie `x = 3 * a;` może zostać zmutowane na następujące sposoby:

Listing 3.8: Przykład mutantów generowanych przez UOI

```
x = 3 * +a;
x = 3 * -a;
x = +3 * a;
x = -3 * a;
```

- **UOD – Unary Operator Deletion**.

  Każdy operator unarny (arytmetyczny +, arytmetyczny -, logiczny !) jest usuwany. Na przykład, wyrażenie `if !`$(a > -b)$ może zostać zmutowane na następujące sposoby:

Listing 3.9: Przykład mutantów generowanych przez UOD

```
if (a > -b)
if !(a > b)
```

- **SVR – Scalar Variable Replacement**. Każda referencja zmiennej może być zastąpiona przez inną zmienną odpowiedniego typu zgodnego z deklaracją zmiennej. Na przykład, wyrażenie `x = a * b;` może zostać zmutowane na następujące sposoby:

Listing 3.10: Przykład mutantów generowanych przez SVR

```
x = a * a;
a = a * b;
x = x * b;
x = a * x;
x = b * b;
b = a * b;
```

- **BSR — Bomb Statement Replacement**. Każda instrukcja może być zastąpiona specjalną funkcją `Bomb()`. `Bomb()` sygnalizuje awarię natychmiast po jej wykonaniu. Ten operator wymusza na testerze pokrycie testami każdej linii kodu, która może podlegać tej mutacji oraz wprowadzenie zabezpieczeń przed krytycznymi defektami. Na przykład, wyrażenie `x = a * b;` może zostać zmutowane w następujący sposób:

Listing 3.11: Przykład mutantów generowanych przez BSR

```
Bomb();
```

3.4.2 Mutanty integracyjne

Integracyjne operatory mutacyjne (nazywane również „mutacją interfejsu") działają mutując połączenia między komponentami oprogramowania. W tym wypadku większość mutacji dotyczy

metod w programie. Mutanty integracyjne symulują defekty specyfikacji projektu programu oraz standardowe defekty programistyczne. Dobrym przykładem takiego defektu jest wypadek marsjańskiego lądownika Mars Climate Orbiter z września 1999 roku, gdzie przyczyną katastrofy było wysłanie przez jeden komponent wartości zmiennej siły w jednostkach anglosaskich (funty), a przy odbiorze założono, że wartość była podana w jednostkach układu SI (w newtonach).

- **IPVR – Integration Parameter Variable Replacement**. Każdy parametr w wywołaniu metody jest zastępowany przez inną zmienną ze zgodnym typem.
- **IUOI – Integration Unary Operator Insertion**. Każde wyrażenie w wywołaniu metody jest modyfikowane poprzez wstawienie wszystkich możliwych unarnych operatorów z przodu do tyłu. Na przykład, wyrażenie `i++` zostanie zamienione na `++i`.
- **IPEX – Integration Parameter Exchange**. Każdy parametr w wywołaniu metody jest wymieniany z innym parametrem kompatybilnym pod względem typu. Przykładowo, jeśli metoda `f(int a, int b, bool c)` do tej pory była wywoływana w następujący sposób: `f(a=1, b=4, c=false)`, to po mutacji wywołanie może wyglądać w następujący sposób `f(b=4, a=1, c=false)`.
- **IMCD – Integration Method Call Deletion**. Każde wywołanie metody po mutacji jest usuwane. Jeśli metoda zwraca wartość i jest używana w wyrażeniu, wywołanie metody jest zastępowane odpowiednią stałą.
- **IREM – Integration Return Expression Modification**. Każde wyrażenie w każdej instrukcji `return` w metodzie jest modyfikowane przez zastosowanie operatorów mutacyjnych UOI oraz AOR opisanych w punkcie 3.4.1.

3.4.3 Mutanty obiektowe

Są to mutanty operujące na poziomie klas i obiektów. Stosowane są tylko i wyłącznie w językach, w których wspierana jest obiektowość.

- **AMC – Access Modifier Change**. Poziom dostępu dla każdej zmiennej lub metody zostaje zmieniony na inny. Przykładowo, poziom dostępu `public` zostanie zmieniony na `private`. Tego typu mutacje sprawdzają, czy testy uwzględniają odpowiednie poziomy dostępu do elementów składowych klas.
- **HVD – Hiding Variable Deletion**. Każde ukrycie zmiennej dziedziczonej z klasy nadrzędnej zostaje usunięte. W ten sposób odnosząc się do zmiennej zawsze odnosimy się do wartości z klasy bazowej. Ta mutacja jest symulacją jednego z częstych błędów popełnianych przez programistów.
- **HVI – Hiding Variable Insertion**. Mutacja opisana przez ten operator jest symetryczna do HVD. W tym wypadku dodajemy celowo zmienną, która nadpisuje wartość zmiennej z klasy bazowej. Jeżeli test nie wykryje różnic, oznacza to, że mutant nie został wykryty.

- **OMD – Overriding Method Deletion**. Każde nadpisanie zmiennej z klasy bazowej jest usuwane. Skutkuje to odnoszeniem się poprzez referencję zawsze do klasy bazowej zamiast do klasy dziedziczącej.

- **OMM – Overridden Method Moving**. Każde wywołanie nadpisanej metody z klasy bazowej w nadpisanej metodzie jest wstawiane na koniec i na początek tej metody. Jest to symulacja uruchomienia metody z klasy bazowej w niewłaściwym miejscu i kolejności.

- **OMR – Overridden Method Rename**. Zmiana nazwy nadpisywanej metody z klasy bazowej. W wielu przypadkach ten mutant może wywołać błąd kompilacji, ponieważ dla większości języków nadpisanie metody odbywa się przez słowo kluczowe `override` wskazujące, że metoda o danej sygnaturze istnieje w klasie bazowej. Brak tej metody może skutkować błędem kompilacji. Jest to typ mutacji weryfikujący, czy testy uwzględniają poprawność polimorfizmu klas.

- **SKD – Super Keyword Deletion**. Mutacje tego typu usuwają wszystkie wystąpienia słowa kluczowego `super`/`base` (w zależności od języka). Po usunięciu tego słowa kluczowego w programie zawsze używana jest referencja do lokalnej wartości, a nie do dziedziczonej z klasy bazowej.

- **PCD – Parent Constructor Deletion**. Tego typu mutanty usuwają wywołania konstruktorów klasy bazowej w klasach dziedziczących. Jeżeli domyślny konstruktor klasy bazowej może wprowadzić program w nieprawidłowy stan, to test powinien to wykryć.

- **ATC – Actual Type Change**. Za pomocą słowa kluczowego `new()` typ bazowy obiektu zostaje zmieniony. Nowy typ musi być kompatybilny, czyli należeć do tej samej dziedziny wartości, co typ przed mutacją.

- **DTC – Declared Type Change**. Typ deklarowanego obiektu zostaje zmieniony na inny, kompatybilny z nim. Najczęściej jest to typ klasy, po której dany obiekt dziedziczy. Na przykład, w języku C# można wymienić typ `list<>` na typ `IEnumerable<>`. Taka zmiana powinna być natychmiastowo wykryta przez testy.

- **PTC – Parameter Type Change**. Dla wszystkich parametrów danego obiektu jest wykonany operator mutacyjny DTC.

- **RTC – Reference Type Change**. W części języków programowania istnieje możliwość zastosowania mutanta, który podmienia prawą stronę wyrażenia przy przypisaniu referencji na inny obiekt z kompatybilnym typem.

- **OMC – Overloading Method Change**. Zamieniane są ciała każdej pary metod, które mają tę samą nazwę. To zapewnia, że metody przeciążone są odpowiednio wywoływane.

- **OMD – Overloading Method Deletion**. Każda deklaracja metody przeciążenia jest usuwana. Operator OMD zapewnia pokrycie przeciążonych metod, ponieważ przeciążone metody muszą być wywoływane co najmniej raz.

- **AOC – Argument Order Change**. Kolejność argumentów w wywołaniach metod zmienia się tak, aby były takie same, jak przeciążanej metody, jeśli taka istnieje. Powoduje to wywołanie innej metody, a więc sprawdzenie usterki przeciążenia niewłaściwej metody.

- **ANC – Argument Number Change**. Ten sam przypadek co AOC, tylko w tym wypadku zmieniamy liczbę argumentów.

- **TKD – `this` Keyword Deletion**. Każde wystąpienie słowa kluczowego `this` jest usuwane. Powoduje to, że zawsze będziemy używać lokalnej wersji zmiennej dostarczonej na przykład w parametrze metody zamiast tej, która należała do obiektu. Słowo `this` powoduje, że jeżeli w metodzie występuje zmienna lokalna o tej samej nazwie co parametr obiektu, należy użyć parametru obiektu. Bez `this` zawsze zostanie użyta zmienna lokalna.

- **SMC – Static Modifier Change**. Każda instancja modyfikatora statycznego jest usuwana. Modyfikator statyczny jest dodany do zmiennych instancji. Ten operator sprawdza użycie zmiennych instancji i klas.

- **VID – Variable Initialization Deletion**. Usuwane są inicjalizacje wszystkich zmiennych składowych. W konsekwencji zostaną ustawione tylko wartości domyślne oraz wartości typu `null`.

- **DCD – Default Constructor Deletion**. Usunięcie deklaracji konstruktora domyślnego.

3.4.4 Mutanty wykonywane na gramatyce

Często zdarza się, że program dostaje zniekształcone dane wejściowe, a następnie je odrzuca. Aspekt nieprawidłowego wejścia powinien zdecydowanie zostać przetestowany. Takie podejście często nazywane jest w literaturze testowaniem negatywnym (ang. dirty testing) i jest szczególnie wykorzystywane w testowaniu interfejsów (API). Jest to rzecz, która często wymyka się uwadze twórcom oprogramowania, którzy koncentrują się na tworzeniu programu z zawsze poprawnym stanem wewnętrznym. Nieprawidłowe dane wejściowe mogą wprowadzić program w nieprzewidziany i niewłaściwy stan wewnętrzny. Dlatego przetestowanie poprawności danych wejściowych jest jednym z kluczowych zagadnień kontroli jakości.

Zanim przejdziemy do omówienia samych operatorów, przypomnijmy podstawowe pojęcia teorii języków formalnych.

Definicja 3.6 (gramatyka)**.** Gramatyką nazywamy czwórkę $G = (V_N, V_T, v_0, P)$, gdzie V_N to zbiór tzw. symboli nieterminalnych, V_T – terminalnych (przy czym zakłada się, że $V_N \cap V_T = \emptyset$), $v_0 \in V_N$ to symbol początkowy gramatyki, a $P \subset (V_N \cup V_T)^+ \times (V_N \cup V_T)^*$ jest zbiorem produkcji.

Definicja 3.7 (wywód)**.** Wywodem w gramatyce $G = (V_N, V_T, v_0, P)$ nazywamy zwrotne i przechodnie domknięcie relacji P i oznaczamy je symbolem $\rightarrow^*$.

Definicja 3.8 (język)**.** Językiem generowanym przez gramtykę $G = (V_N, V_T, v_0, P)$ nazywamy zbiór wszystkich i tylko takich słów nad alfabetem terminalnym, które można wywieść z symbolu początkowego:

$$L(G) = \{w \in V_T^* : \ v_0 \rightarrow^* w\}.$$

Języki programowania oparte są zazwyczaj na gramatykach bezkontekstowych. Gramatykę nazywamy bezkontekstową, jeśli każda jej produkcja ma postać $u \rightarrow \alpha$, gdzie $u \in V_N$, $\alpha \in (V_N \cup V_T)^*$.

Jeżeli dane wejściowe są zgodne z jakąś gramatyką, to w celu przetestowania tych danych mutacyjnie możemy się posłużyć następującymi operatorami mutacyjnymi.

- **Nonterminal Replacement**. Każdy nieterminalny symbol w danych wejściowych zostaje zastąpiony innym nieterminalnym znakiem. Jest to bardzo szeroki operator mutacji, który może skutkować wieloma ciągami znaków, które są niepoprawne, a czasem nawet bezużyteczne dla testowania. Jeśli gramatyka zawiera określone reguły lub ograniczenia składniowe, można uniknąć niektórych nieterminalnych zamienników. Jest to analogiczne do unikania błędów kompilatora w mutacji kodu programu. Na przykład, jeśli $V_N = \{A, B, C\}$, a do zbioru P należy produkcja $A \rightarrow BC$, to przykładami mutantów tej produkcji są $A \rightarrow AC$, $A \rightarrow BA$, $A \rightarrow BB$, $B \rightarrow BC$ itd.

- **Terminal Replacement**. Operator jest analogiczny do „Nonterminal Replacement" z tą różnicą, że zamieniane symbole są terminalami. Jeśli rozważymy przykład z poprzedniego punktu i dodatkowo założymy, że $V_T = \{x, y\}$, to np. produkcja $A \rightarrow xBy$ może być zmutowana w produkcję $A \rightarrow yBy$ lub $A \rightarrow xBx$.

- **Terminal and Nonterminal Deletion**. Każdy terminal lub nieterminal w ciągu danych wejściowych może zostać usunięty.[8] Dla przykładu z powyższych punktów, produkcja $A \rightarrow xBy$ może zostać zmutowana np. w produkcje: $A \rightarrow By$, $A \rightarrow xy$, $A \rightarrow By$, $A \rightarrow x$ itd.

- **Terminal and Nonterminal Duplication**. Każdy terminal lub nieterminal może zostać zduplikowany. Na przykład, produkcja $A \rightarrow x$ może zostać zamieniona na $AA \rightarrow x$ lub $A \rightarrow xx$.

[8] Proces doboru słów do usunięcia powinien być losowy, jeśli nic nie wiemy na temat gramatyki generującej poprawne dane.

Wszystkie powyżej opisane operatory mutacyjne są tylko wycinkiem większej gamy operatorów, ponieważ możliwości definiowania operatorów są w zasadzie nieograniczone. Przykładem innych tego typu operatorów może być np. zamiana kolejności symboli w produkcji, dodanie symbolu do produkcji, dodanie produkcji wymazującej (np. $A \rightarrow \lambda$) itp. Często specjalistyczne oprogramowanie do testów mutacyjnych posiada własną gamę zdefiniowanych operatorów, których cześć pokrywa się z wymienionymi powyżej, ale część może też być nowa. Platformy do testów mutacyjnych czasem dają nawet możliwość definiowania własnych operatorów mutacyjnych. Lista opisanych powyżej operatorów może być traktowana jednak jako zestaw bazowy, do którego można wprowadzać dodatkowe, własne pomysły.

3.5 Mutanty pierwszego rzędu

Definicja 3.9 (mutant pierwszego rzędu)**.** Mutant pierwszego rzędu (FOM - ang. First Order Mutant) to mutant powstający poprzez zastosowanie jednego operatora mutacyjnego oraz pojedynczej mutacji.

Tradycyjne pojęcie mutacji kodu dotyczy właśnie mutantów pierwszego rzędu. Tabela 3.1 prezentuje przykłady takich mutantów. Symbol P' oznacza mutację programu P.

Tabela 3.1: Przykłady mutantów pierwszego rzędu

Program P	Program P'
...	...
`while(a > 0) {`	`while(a >= 0) {`
`a = b - c;`	`a = b - c;`
`c++;`	`c++;`
}	}
...	...

3.6 Mutanty wysokiego rzędu

Definicja 3.10 (mutant wysokiego rzędu)**.** Mutant wysokiego rzędu (HOM - ang. High Order Mutation) to mutant, do stworzenia którego zastosowano więcej niż jeden operator mutacyjny.

Mutacje wysokiego rzędu uzyskiwane są poprzez wstrzyknięcie do programu P więcej niż jednego defektu składniowego. Innymi słowy, HOM są utworzone przez połączenie co najmniej dwóch FOM. Przykładem takiej mutacji jest $M_2(M_1(P))$ gdzie M_1, M_2 są różnymi operatorami mutacyjnymi

lub tym samym operatorem działającym na dwóch różnych miejscach programu P. HOM otrzymują nazwę zgodnie z liczbą połączonych FOM. Zatem mutant wytworzony z dwóch FOM będzie nazywany mutantem drugiego rzędu (SOM - Second Order Mutation), z trzech – mutantem trzeciego rzędu TOM (Third Order Mutation) i tak dalej. Przykład mutantów wysokiego rzędu przedstawiony jest w tabeli 3.2, gdzie P oznacza program przed mutacją, M_i oraz M_j – mutanty pierwszego rzędu wytworzone z P, a M_{ij} – mutanta drugiego rzędu będącego połączeniem dwóch mutantów pierwszego rzędu M_i oraz M_j.

Tabela 3.2: Przykłady mutantów wysokiego rzędu

Program P	M_i (FOM)	M_j (FOM)	$M_{i,j}$ (SOM)
...	...	...	...
`while(a > 0)`	`while(a >= 0)`	`while(a > 0)`	`while (a >= 0)`
{	{	{	{
`a = b - c;`	`a = b - c;`	`a = b + c;`	`a = b + c;`
`c++;`	`c++;`	`c++;`	`c++;`
}	}	}	}
...	...	...	...

3.7 Mutacje wysokopoziomowe na procesach biznesowych

Wspomnieliśmy wcześniej, że testowanie mutacyjne nie musi ograniczać się wyłącznie do testów niskopoziomowych, takich jak testowanie jednostkowe, gdzie modelem podlegającym mutacjom jest kod źródłowy programu. Testowanie mutacyjne może być także wykorzystywane w testach wysokopoziomowych, np. w celu testowania procesów biznesowych opisywanych w języku UML[9] czy BPMN[10].

3.7.1 BPMN

Interesujące badania na temat zastosowania testów mutacyjnych w BPMN zawarte są w pracy [22]. Opisuje ona testowanie mutacyjne na podstawie BPMN, które odbywa się dzięki transformacji modelu z zapisu w BPMN do WS-BPEL (Web Services Business Process Execution Language). WS-BPEL jest opartym na plikach XML językiem definiującym procesy biznesowe usług sieciowych. Po skonwertowaniu modelu do formatu WS-BPEL mutacje są wykonywane bezpośrednio na XML-owym kodzie WS-BPEL. Po wygenerowaniu zbiorów mutantów zgodnie z definicją testowania mutacyjnego uruchamiane są testy weryfikujące poprawność modelu. Po zakończeniu procesu mutacji raport z wykrywalności mutantów definiuje siłę mechanizmów testujących dany model.

W pracy zostały wprowadzone także operatory mutacyjne operujące na kodzie XML. Autor wymienia ich 25, jednak w samej pracy wyróżniono tylko cztery operatory główne, które posiadają swoje podtypy.

[9]UML czyli Unified Modeling Language (zunifikowany język modelowania), język wykorzystywany do modelowania różnego rodzaju systemów i procesów biznesowych. Język został stworzony przez Grady'ego Boocha, Jamesa Rumbaugha oraz Ivara Jacobsona.

[10]BPMN czyli Business Process Modeling and Notation (notacja i modelowanie procesu biznesowego) jest graficzną, intuicyjną notacją, która służy do opisywania procesów biznesowych.

- **Identifier Mutation Operator (I)**. Ten operator posiada dwa podtypy. Pierwszy zamienia każdy znacznik kodu XML na inny o tym samym typie. Drugi natomiast zamienia znacznik na inny dowolny znacznik[11].

- **Expression Mutation Operator (E)**. Jest to zbiór wszystkich operatorów mutacyjnych, które działają na operatorach takich jak + ,- ,/ ,* , <, >, <=, >=, !=, ==, = oraz wyrażeniach typu czasowego <timeDate>. Ten zbiór operatorów mutacyjnych częściowo odpowiada zbiorowi projektowych operatorów mutacyjnych opisanych w punkcie 3.4.1. Oba zbiory nie pokrywają się w całości, co wynika ze wspomnianego wcześniej uzależnienia operatorów mutacyjnych od języka, w jakim są one stosowane.

- **Activity Mutation Operator (A)**. Operatory mutacyjne tego typu modyfikują kolejność oraz warunki uruchamiania konkretnych fragmentów modelu BPMN przez testy lub inne fragmenty modelu. Operatory mutacyjne w tym wypadku działają na odniesieniach do konkretnych fragmentów kodu XML, na warunkach zawartych w parametrach, na wyrażeniach arytmetycznych parametrów oraz na wszystkich innych elementach określających przepływ aktywności fragmentów modelu.

- **Exception and Event Mutation (X)**. Nie jest to operator mutacyjny złożony z podtypów. Jego mechanika działania polega na modyfikacji zachowania modelu podczas wywołania wyjątku lub określonego zdarzenia.

3.7.2 UML

Jedne z ciekawszych badań nad testowaniem mutacyjnym procesów biznesowych opisanych w języku UML prowadzą naukowcy z AIT Austrian Institute of Technology. Na obecną chwilę zaprojektowali oni narzędzie służące do wykonywania testów mutacyjnych na UML-owych diagramach stanów „MoMUt" [23].

Narzędzie MoMut dostarcza zautomatyzowane mechanizmy generacji pokrycia defektów kodu, analizy zestawów testowych oraz lokalizacji defektów modelu. Opis i zastosowanie technologii znajduje się w [24]. W artykule autorzy wprowadzili pojęcie testowania mutacyjnego opartego na modelu (ang. MBMT — model-based mutation testing). Od strony procesowej MBMT nie różni się niczym od standardowego testowania mutacyjnego kodu. Jedyną różnicą jest środowisko, w jakim ta technologia działa, czyli język UML. W MBMT został także zaproponowany osobny zestaw operatorów mutacyjnych. Technologia MoMUt jest już na etapie, na którym osiągnięta została stabilność platformy mutacyjnej i może ona zostać wprowadzona do użytku przemysłowego. Zastosowanie praktyczne narzędzia MoMut jest opisane w [25].

3.8 Mutanty bazodanowe

Wszystkie języki programowania, które mogą być testowane w jakiś sposób za pomocą testów automatycznych, mogą być testowane mutacyjnie. Języki stosowane do tworzenia i zarządzania

[11] Trzeba pamiętać, że XML jest językiem bardzo elastycznym, pozwala na swobodną reprezentację danych oraz typów. Wobec tego, aby ten operator mógł zostać zastosowany, deklaracje typów danych zawartych w XML muszą być dobrze zdefiniowane oraz podlegać regułom podobnym do tych, jakie są stosowane w językach programowania.

bazami danych (czyli języki z rodziny SQL) spełniają tę definicję. Obecnie prowadzone są badania w celu zastosowania testów mutacyjnych w językach bazodanowych. W kontekście tych badań warto odnieść się do dwóch interesujących prac:

- Pierwszą z nich, dotyczącą testowania mutacyjnego w zastosowaniu do języków bazodanowych oraz baz danych jest praca magisterska złożona na Virginia Polytechnic Institute and State University o tytule „Towards A Sufficient Set of Mutation Operators for Structured Query Language (SQL)" [26]. Praca ta zajmuje się praktycznym zastosowaniem bazodanowych testów mutacyjnych.
- W pracy „Structural Coverage Criteria for Testing SQL Queries" [27] zaproponowano i zdefiniowano zestaw bazodanowych operatorów mutacyjnych. Autorzy częściowo wzorowali się na operatorach zaproponowanych w pracy Ammanna i Offutta [14], a także tych opisanych w podrozdziale 3.3.

3.9 Mutanty równoważne

Stwierdzenie, czy dwa zmutowane fragmenty tego samego programu są równoważne jest tym samym, co stwierdzenie równoważności dwóch programów. Wspomnieliśmy wcześniej, że jest to problem nierozstrzygalny, co wynika z twierdzenia o nierozstrzygalności problemu równoważności maszyn Turinga. Definicję równoważności mutantów można sformułować w następujący sposób (niech $D, \mathcal{M}$ oznaczają odpowiednio: zbiór wszystkich danych wejściowych oraz zbiór wszystkich możliwych mutantów programu P):

Definicja 3.11 (Mutant równoważny)**.** Mutant $M \in \mathcal{M}$ jest mutantem równoważnym programowi P, jeżeli dla każdego $d \in D$ zachodzi $M(d) = P(d)$.

Listing 3.12: Przykład mutanta równoważnego

```
...
int i = 0;
int j = 0;
while (true) {
  j = j + i;
  i = i + 1;
  if (i == 5) break; // kod oryginalny
}
...

...
int i = 0;
int j = 0;
while (true) {
```

```
  j = j + i;
  i = i + 1;
  if (i >= 5) break; // mutant rownowazny
}
...
```

Przykład mutanta równoważnego jest pokazany na listingu. W mutancie operator porównania został zamieniony na operator „większe lub równe", ale zmiana ta nie spowoduje innego wykonania programu – oba kody są semantycznie równoważne.

W literaturze problem stwierdzenia, czy mutant jest równoważny programowi opisany jest jako problem równoważności mutantów (ang. Equivalent Mutant Problem). Możliwość istnienia mutantów równoważnych pociąga za sobą to, że osiągnięcie stuprocentowej wykrywalności mutantów może być celem niemożliwym do osiągnięcia. Co więcej, obecność tego rodzaju mutantów może zaburzyć wynik miary pokrycia mutacyjnego, sztucznie go zaniżając.

Mutanty równoważne stały się jednym z bardziej istotnych problemów, które dotykają techniki testów mutacyjnych. W ciągu ostatnich lat pojawiło się wiele publikacji, w których podejmowane są – oparte na heurystycznych algorytmach – próby choćby częściowego rozwiązania tego problemu. Na przykład Schuler i Zeller [72] zaobserwowali, że jeśli mutant zmienia pokrycie, jest 75% szans na to, że nie jest równoważny. Wiele prac na temat mutantów równoważnych opiera się na badaniach empirycznych. W pracy Papadakisa i Malevrisa [28], gdzie optymalizacja była stosowana dla języka programowania C, doprowadzono do redukcji około 80-90 procent wygenerowanych równoważnych mutantów. W pracy Kintis et al. [29] uzyskano wyniki wahające się od 65.5% do 86.8%. W tej pracy badania były przeprowadzane w języku Java. W obu pracach stosowano podejście generowania tylko mutantów wysokiego rzędu. W pracy [30] został zaprezentowany przegląd najciekawszych metod stosowanych w celu rozwiązywania problemu mutantów równoważnych. Szczególnie ciekawe ze względu na automatyzację procesu selekcji mutantów tak ażeby w jak największym stopniu wyeliminować wystąpienie mutantów równoważnych są prace stosujące do tego celu algorytmy takie jak na przykład *Last2First* [30].

Algorytm *Last2First* polega na tworzeniu mutantów wysokiego rzędu na poprzez „sklejanie" mutantów pierwszego rzędu. W pierwszym kroku algorytm sortuje listę mutantów pierwszego rzędu w kolejności ich wygenerowania. Następnie tworzony zostaje mutant drugiego rzędu z ostatniego i pierwszego mutanta na wcześniej wygenerowanej liście. Po wygenerowaniu mutanta drugiego rzędu oba użyte mutanty pierwszego rzędu usuwane są z listy. Cały proces jest powtarzany aż do ostatniego elementu na liście. Podobnym algorytmem do *Last2First* jest *RandomMix* [30], tylko w tym wypadku proces łączenia jest losowy.

Innym ciekawym algorytmem jest *JudyDiffOp* [30][12], którego pseudokod autorzy podają w swojej pracy. Mechanika tego algorytmu polega na przechodzeniu po kodzie, dopóki nie odnajdzie się miejsca, w którym może zostać stworzony mutant. Następnie dla tego miejsca w kodzie tworzone jest po jednym mutancie dla każdego typu operatora. Tworzone w ten sposób mutanty

[12]Oryginalne wersje algorytmów *JudyDiffOp*, *RandomMix* oraz *Last2First* znajdują się w pracy [31]. Podczas procesu mutacji zastosowanie znalazły ich zmodyfikowane wersje opisane w [30].

są mutantami pierwszego rzędu. Następnie wszystkie mutanty pierwszego rzędy dla tego miejsca w kodzie są łączone w mutanta wyższego rzędu. Autorzy w swojej pracy wskazują, że ta metoda daje co najmniej 50-procentową redukcję liczby mutantów. Według autorów stosowanie technik opartych na algorytmie *JudyDiffOp* jest jednym z najnieskuteczniejszy rozwiązań w celu redukcji mutantów równoważnych.

3.10 Testowanie mutacyjne jako model predykcji defektów

Analiza mutacyjna może być również wykorzystana jako prosty model przewidywania defektów oprogramowania. Odbywa się to poprzez zastosowanie modyfikacji metody wielokrotnych łowień. Załóżmy, że zestaw testów T był w stanie znaleźć N spośród D rzeczywistych defektów w T. Wartość D jest nieznana i chcemy dokonać jej estymacji. Załóżmy ponadto, że w wyniku procesu testowania mutacyjnego, używając tego samego zestawu testów T, miara pokrycia mutacyjnego (stosunek zabitych mutantów do wszystkich wygenerowanych) wyniosła S. Jeśli tylko operatory mutacyjne dobrze naśladowały rzeczywiste pomyłki, jakie popełniają programiści w tym projekcie, to można dokonać następującego szacowania: skoro na sztucznie wygenerowanych mutantach testy osiągnęły pokrycie mutacyjne S, to podobne pokrycie -— w terminach odsetka wykrytych rzeczywistych defektów —- powinny osiągnąć testy T uruchamiane na oryginalnym kodzie. Zatem $S \approx N/D$. Stąd możemy uzyskać estymację szukanej całkowitej liczby defektów w kodzie: $D \approx N/S$. Oczywiście wszystkie uwagi z poprzedniego podrozdziału dotyczące niepewności pomiaru w związku z mutantami równoważnymi pozostają w mocy. Predykcja defektów oprogramowania jest szczegółowo opisana w pracy [32].

3.11 Obecnie prowadzone badania

Obecnie prowadzone badania dotyczące testowania mutacyjnego dotyczą głównie dwóch zagadnień:

- Optymalizacji mutantów, której poświęcony jest rozdział 4.
- Zastosowania testowania mutacyjnego. W tym wypadku główne badania polegają na wprowadzeniu technik mutacyjnych do jak największej liczby procesów wytwórczych. Testowanie mutacyjne wprowadza się w takich obszarach jak: bazy danych i ich języki, projektowanie oprogramowania (UML, BPMN, itp.), mutowanie danych wsadowych itd.

Rozdział 4

Optymalizacja mutacji

W dzisiejszych czasach bardzo powszechne jest stosowanie zwinnych technik wytwarzania oraz nowych praktyk inżynierii oprogramowania, takich jak ciągła integracja czy ciągłe dostarczanie oprogramowania. Tego typu podejścia w dużym stopniu wykorzystują testy regresji. Testy regresji służą do sprawdzania, czy wprowadzona do oprogramowania zmiana (np. nowa funkcjonalność) nie spowodowała problemów w innych częściach oprogramowania. Zestawy testów regresji mają tendencję do szybkiego rozrastania się, a co za tym idzie, problematyczne staje się ich efektywne wykonanie. Tym bardziej stosowanie testowania mutacyjnego na tak dużym zestawie testów jest bardzo czasochłonne oraz jest związane z wieloma problemami opisanymi powyżej w rozdziale 3. W literaturze zaproponowano kilka metod rozwiązania tego problemu. Jednym z najprostszych podejść jest zmniejszenie liczby wykonywanych testów, tzn. zredukowanie rozmiaru suity testów regresji. Kolejnym jest ustalanie priorytetów dla testów. W tym wypadku próbuje się zdefiniować kolejność wykonania testów dla każdego mutanta tak, aby test, który zabija danego mutanta, został wykonany tak wcześnie, jak to możliwe. Zagadnienie to jest bliżej opisane w pracach [33] oraz [34]. Minimalizacja rozmiaru suity testów regresji oraz ustalanie ich priorytetów szerzej zostało opisane w podrozdziałach 4.5 oraz 4.6.

W trakcie badań opisanych w niniejszej pracy przeanalizowanych zostało wiele technik optymalizacyjnych stosowanych w testowaniu mutacyjnym. W tym rozdziale znajduje się — oparty na literaturze — opis najczęściej stosowanych technik optymalizacyjnych oraz wstępny opis technik opracowanych na potrzeby badań wykonanych przez autora niniejszej rozprawy. Wszystkie dyskutowane poniżej podejścia są godne uwagi ze względu na wiedzę, na podstawie której można poprawić wydajność przeprowadzania testów mutacyjnych.

4.1 Próbkowanie mutantów

Jedną z pierwszych technik optymalizacji procesu testowania mutacyjnego jest ograniczanie liczby mutantów poprzez próbkowanie (ang. sampling). To zagadnienie było poruszane przez badaczy wielokrotnie. Jako najprostsze podejście można przytoczyć to opisane w pracy Mathura i Wonga [35], w której autorzy proponują technikę próbkowania mutantów polegającą na losowym usuwaniu kolejno 10%, 15%, 20%, 25% aż do 40% z wszystkich mutantów, a następnie badaniu wpływu

tej operacji na pokrycie mutacyjne z uwzględnieniem liczby niewykrytych mutantów. W pracy jest też wspomniana druga metoda ograniczania liczby mutantów polegająca na wykorzystaniu tylko konkretnych typów mutantów ABS (Absolute Value Insertion) wstawiających wartości bezwzględne oraz ROR (Relational Operator Replacement) zmieniających operatory relacyjne. Mutanty innych typów są ignorowane. Na badanym przez autorów oprogramowaniu metoda pierwsza wykazała skuteczność w zmniejszeniu liczby mutantów do ok. 60%, natomiast metoda druga — do ok. 80%. Zastosowanie obu metod przyniosło tylko nieznaczny spadek pokrycia mutacyjnego do ok. 80%.

W pracy Offutta, Rothermela oraz Zapfa [36] zastosowano podejście polegające na podzieleniu mutantów na trzy grupy. Podział został wykonany na podstawie operatorów mutacyjnych zdefiniowanych przez oprogramowanie Mothra [37]. Każdy podzbiór operatorów mutacyjnych wykazywał wykrywalność mutantów przez testy na poziomie od 97% do 99%. Jest to dobry wynik, ale ze względu na małą liczbę programów oraz ich wielkość (która wynosiła od 10 do 48 linii kodu) wyniki eksperymentu niekoniecznie mogą przenosić się na duże programy.

Przytoczone powyżej prace są jednymi z pierwszych, które dotyczą ograniczania liczby mutantów za pomocą próbkowania. Do tej pory powstało ich wiele. Jedną z ciekawszych jest praca Ilony Blumke oraz Karola Kuleszy z 2013 roku [38], w której autorzy postanowili zastosować wyżej wspomniane metody próbkowania (oraz inne znane z literatury i przytoczone w ich pracy) w celu sprawdzenia skuteczności tych metod w programach obiektowych. Poprzednie eksperymenty przeprowadzono z użyciem języków strukturalnych, głównie Fortrana. Eksperymenty opisane przez autorów wykazały, że stosowanie losowych próbek 60% mutantów w programach Java może znacznie obniżyć koszty testów przy akceptowalnym wyniku pokrycia mutacyjnego. W opisanych w pracy eksperymentach można znaleźć informację, że nawet 20% mutantów było w stanie wykryć ponad 30% defektów w testach. Autorzy przeprowadzili swoje eksperymenty na specjalnie w tym celu napisanych czterech programach ze średnio dwiema-trzema klasami nieprzekraczającymi łącznie 250 linii kodu. Programy były implementowane w paradygmacie obiektowym i zostały napisane w języku Java zgodnie ze standardem Java 8. Interesującym rozwinięciem tej pracy byłoby zastosowanie metody w niej opracowanej do projektów większych, komercyjnych lub opartych na otwartym kodzie (ang. open-source).

W naszych badaniach także zajmujemy się ograniczaniem liczby mutantów, proponując do tego model matematyczny losujący mutanty na podstawie prawdopodobieństwa warunkowego, oparty na podejściu bayesowskim. Więcej informacji odnośnie tej optymalizacji znajduje się w rozdziałach 6 oraz 7.

4.1.1 Skrypty mutacyjne

Innym podejściem do ograniczenia liczby mutantów może być stosowanie technologii związanej z tzw. skryptami mutacyjnymi. Testowanie mutacyjne postrzegane jako technologia typu „czarnoskrzynkowgo” (ang. black-box) opiera się na założeniu, że przestrzeń, w jakiej porusza się testowane oprogramowanie, posiada nieznaną charakterystykę i strukturę. Innymi słowy, nie jest wiadome, jakie mutacje i w jaki sposób wpłyną na testy, wobec czego używa się wszelkich

możliwych do zastosowania w oprogramowaniu testującym operatorów mutacyjnych.

Wykonując proces testowania mutacyjnego po raz kolejny na jakimś programie lub na jego kolejnej wersji nabywa się jednak pewną wiedzę odnośnie skuteczności konkretnych operatorów mutacyjnych w danym oprogramowaniu. W konsekwencji, w celach optymalizacji można usunąć z puli operatorów mutacyjnych operator, którego efektywność (patrz def. 3.3) jest bliska zeru. Proces usuwania mutantów zgodnych z konkretnym typem operatora mutacyjnego niekoniecznie musi być zastosowany do wszystkich mutantów danego typu. Usunięciu może ulec na przykład tylko konkretny podzbiór mutantów generowanych za pomocą danego operatora. Technika umożliwiająca tego typu optymalizację polegającą na zmniejszeniu liczby generowanych mutantów opisana jest w pracy [39] oraz w dokumentacji znajdującej się na stronie projektu MAJOR [40] (punkt 3.1.2).

4.2 Równoważność operatorów mutacyjnych

Analiza mutacyjna oprogramowania jest bardzo potężnym narzędziem, dzięki któremu jakość testów i testowanego oprogramowania znacznie wzrasta. Niestety, jak to zostało wspomniane we wcześniejszych rozdziałach, proces testowania mutacyjnego może być bardzo czasochłonny, zwłaszcza gdy jest wykonywany na dużym i złożonym oprogramowaniu. Jest to jeden z powodów, dla których technika ta jeszcze nie jest tak popularna w rzeczywistych projektach profesjonalnego wytwarzania oprogramowania. Kolejną optymalizacją, która może przyśpieszyć ten proces, jest technika polegająca na redukcji użycia równoważnych (redundantnych) operatorów mutacyjnych. Mutant nazywany jest redundantnym, jeśli wynik jego działania można przewidzieć na podstawie wyników działania innych mutantów.

Praca „Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis" autorstwa Justa, Kapfhammera i Schweiggerta [41] porusza problem redukcji liczby mutantów generowanych przez równoważne operatory mutacyjne. Autorzy publikacji badają zastosowanie zredukowanego, ale wystarczającego do przeprowadzenia skutecznej analizy mutacyjnej zestawu mutantów wygenerowanych za pomocą nieredundantnych wersji operatorów warunkowych COR i relacyjnych ROR. Opisane w pracy modyfikacje operatorów mutacyjnych COR oraz ROR zostały zaproponowane w pracy Kamińskiego, Ammanna i Offutta [42]. Badania wykonane zostały na 10 programach liczących łącznie około 400 000 linii kodu. Zastosowany mechanizm optymalizacji wykazał znaczną poprawę wydajności. Czas wykonania programu zmniejszył się w najlepszym wypadku o 65%, natomiast średnio spadł o 30%. Liczba generowanych mutantów spadła maksymalnie o 37% oraz średnio o 25%. Jest to bardzo dobry wynik, wskazujący na skuteczność metody. W pracy wspominano, że liczba niewykrytych mutantów także spadła, jednak ze względu na to, że usunięcie części mutantów wynikało z równoważności operatorów, nie powinno spowodować to utraty części istotnych danych, choć nie można wykluczyć takiego przypadku. Eksperymenty związane z badaniem były przeprowadzanie przy pomocy wspomnianego już wcześniej oprogramowania MAJOR.

4.3 Optymalizacja operatorów logicznych

Dalsze badania prowadzone przez Justa i Schweiggerta nad zmniejszaniem liczby mutantów poprzez eliminację mutantów generowanych przez równoważne operatory mutacyjne doprowadziły w 2014 r. do stworzenia pracy [43]. Została w niej zaprezentowana ciekawa technika optymalizacji logicznych operatorów mutacyjnych. Optymalizacja polega na używaniu do mutacji tylko określonego podzbioru mutantów (tzn. odpowiednio zdefiniowanych operatorów logicznych), który jest minimalnym zbiorem gwarantującym generację niepoprawnego (zmutowanego) wyniku dla każdej możliwej kombinacji wartości logicznych zmiennych występujących w rozważanym wyrażeniu. Zbiór ten ma taką własność, że każdy z tych mutantów zwraca wartość niepoprawną dla jednej kombinacji zmiennych wejściowych oraz poprawne dla wszystkich pozostałych kombinacji. Każdy mutant zwraca wartość niepoprawną dla innej kombinacji wartości zmiennych wejściowych.

Dla przykładu rozważmy następujące typy operatorów mutacyjnych należące do COR (spis wszystkich operatorów mutacyjnych znajduje się w podrozdz. 3.4):

- operatory warunkowe
  - iloczyn logiczny: `a && b`
  - alternatywa logiczna: `a || b`
  - operator równoważności: `a == b`
  - operator nierównoważności: `a != b`
- operatory specjalne
  - przyjmowanie wartości z lewej strony: `lhs`
  - przyjmowanie wartości z prawej strony: `rhs`
  - operator zamiany na prawdę: `true`
  - operator zamiany na fałsz: `false`
- jednoargumentowe operatory boolowskie
  - negacja lewego argumentu: `!a <op> b`
  - negacja prawego argumentu: `a <op> !b`
  - negacja wyrażenia: `!(a <op> b)`

Rozpatrzmy przykładowo pierwszorzędową[1] mutację wyrażenia `a && b`. Minimalny zbiór mutantów pokrywający każde możliwe wyjście z programu został podany w tabeli 4.1.

[1] Trzeba pamiętać, że optymalizacja mutacji poprzez usuwania redundantnych operatorów logicznych odbywa się na poziomie mutacji pierwszego rzędu.

Tabela 4.1: Optymalizacja operatorów logicznych dla `a && b`

		Wyrażenie Oryginalne	Minimalny zbiór mutantów pokrywający wszystkie wyjścia			subsumowane mutacje COR				subsumowane mutacje UOI		
a	**b**	**a && b**	**FALSE**	**LHS**	**RHS**	**a==b**	**a||b**	**a!=b**	**TRUE**	**!(a && b)**	**!a && b**	**a && !b**
0	**0**	0	0	0	0	**1**	0	0	**1**	**1**	0	0
0	**1**	0	0	0	**1**	0	**1**	**1**	**1**	**1**	**1**	0
1	**0**	0	0	**1**	0	0	**1**	**1**	**1**	**1**	0	**1**
1	**1**	1	**0**	1	1	1	1	**0**	1	**0**	**0**	**0**

W tabeli pogrubione wartości oznaczają pozycje, na których wynik mutanta różni się od oryginału.

Pogrubione wartości w kolumnach 4-7 oznaczają zmutowane wartości logiczne całego wyrażenia. Jak widać zbór mutantów: `False`, `LHS`, `RHS`, `a==b` pokrył wszystkie możliwe mutacje wyjścia dla formuły `a && b`. Podobne zbiory można wygenerować dla wszystkich operatorów mutacyjnych operujących na zdaniach logicznych, czyli takich jak: COR, ROR oraz UOI. Generacja wszystkich takich zbiorów została opisana we wcześniej cytowanej pracy [43].

Definicja 4.1 (subsumpcja mutantów)**.** Mutant m_1 subsumuje mutanta m_2 jeżeli wykrycie (zabicie) m_1 oznacza także wykrycie m_2.

W pracy wykazano, że dla ROR 3 z 7 operatorów stanowi zbiór wystarczający do pełnego pokrycia mutacyjnego wyjścia podczas generacji pierwszorzędowych mutantów logicznych. Dla COR jest to natomiast 4 z 10. Wykazano także, że COR subsumuje UOI. W pracy przedstawiono także wyniki empirycznych badań dla powyższej optymalizacji. Badania były wykonane na 10 programach liczących łącznie około 410 000 linii kodu. Badania te wykazały, że eliminacja redundantnych logicznych operatorów mutacyjnych spowodowała zmniejszenie całkowitego czasu analizy mutacyjnej o ponad 20%. Pokazały one zarazem, jak częste są redundantne mutanty tego typu.

4.4 Mutowanie na bajtkodzie

Jeszcze jedną możliwą optymalizacją mutacji jest mutowanie bezpośrednio na bajtkodzie. Technika wprowadzania zmian w kodzie bajtowym oprogramowania polega na manipulowaniu danymi w konkretnych przestrzeniach pamięci, które odpowiadają konkretnym fragmentom oprogramowania. Technika ta w pewnym sensie jest bardziej rozbudowanym wariantem mechanizmów refleksji, które spotykane są w językach opartych na maszynie wirtualnej[2]. Generacja mutantów bezpośrednio na bajtkodzie znacznie przyspiesza czas wykonywania testów mutacyjnych, ponieważ w tym przypadku kompilacja odbywa się tylko raz. Wszystkie mutanty generowane są na już skompilowanym kodzie bajtowym. Języki programowania, które umożliwiają zastosowanie tej techniki do testów mutacyjnych to w szczególności Java oraz szeroko rozumiane środowisko .NET[3].

[2]Maszyna wirtualna kontroluje wszystkie odwołania uruchamianego programu bezpośrednio do sprzętu lub systemu operacyjnego i zapewnia ich obsługę. Jest pośrednikiem między programem zapisanym w kodzie bajtowym a instrukcjami procesora. Taka architektura pozwala, by program zapisany w kodzie bajtowym wykonywał instrukcje w sposób tożsamy z bezpośrednim wysyłaniem żądań do systemu.

[3]Kod napisany w C#, VisualBasic oraz F# jest kompilowany do tożsamego kodu bajtowego.

Technika manipulacji bajtkodem w celu generacji mutantów stosowana jest między innymi w oprogramowaniu PIT [44], gdzie każdy mutant jest generowany za pomocą frameworka ASM [45] służącego do manipulacji kodem w wirtualnej maszynie Javy (JVM). Innym znanym oprogramowaniem stosującym technikę generacji mutantów bezpośrednio na kodzie bajtowym jest oprogramowanie MuJava [46]. Zgodnie z informacjami zawartymi w pracy Ma, Offutta i Kwona [47] do generowania mutantów na platformie MuJava stosowana jest biblioteka BCEL (Byte Code Engineering Library) [48].

4.5 Optymalizacja poprzez zmniejszenie liczby testów

Głównym czynnikiem wpływającym na ilość czasu potrzebnego do wykonania testów mutacyjnych na danym programie jest liczba tych testów. Czas ten można zredukować poprzez zmniejszenie liczby testów oraz poprzez zdefiniowanie „efektywnej" kolejności ich wykonania. Optymalizacja kolejności wykonywania testów jest opisana w rozdziale 4.6, natomiast w tym rozdziale zostaną opisane metody, jakie mogą zostać użyte w celu zmniejszenia liczby testów w programie przy zachowaniu rozsądnego zasięgu mutacji.

Minimalizacja zestawu testów jest problemem trudnym, którego rozwiązanie jest ważne zarówno w celu zmniejszenia kosztów testowania mutacji, jak i precyzyjnej oceny jakości istniejących zestawów testów. Dla zbioru testów można określić, czy jest on nadmiarowy względem testowanego oprogramowania czy też nie. Zestaw testów będzie nadmiarowy, jeśli zawiera test, który operuje na dokładnie tych samych fragmentach kodu oraz wykrywa te same zestawy mutantów co inne testy. W pracy [49] postanowiono spróbować rozwiązać ten problem poprzez sprowadzenie — za pomocą techniki programowania liniowego — zbioru testów do postaci wektora, w którym każdy element jest reprezentacją zasięgu i skuteczności testu w kontekście pokrycia mutacyjnego kodu. Wartość ta jest wyrażana przez koszt czasowy oraz obliczeniowy generowany dla danego testu. Następnie autorzy stosują na takim zestawie danych minimalizację funkcji liniowej ze zmienną decyzyjną w postaci binarnej. Podczas całego procesu cały czas sprawdzane jest, czy w ten sposób minimalizowany zestaw testów dalej pokrywa kod w tym samym stopniu. Autorzy pracy prowadzili badania mutując język WS-BPEL, opisany w punkcie 3.7.1. Jednak według twórców artykułu ich rozwiązanie może być stosowane do dowolnego języka.

Badania, które zostały przeprowadzone w związku z powyższą pracą, dotyczyły pięciu różnych opisów procesów biznesowych. Przeprowadzone eksperymenty wykazały średnio redukcję liczby testów aż o 52%, w najgorszym wypadku o 32%, a w najlepszym — o 78%.

Powyższa technika jest interesująca ze względu na swoją uniwersalność, zastosowanie w procesach projektowania oprogramowania oraz ze względu na prowadzenie badań pod kątem optymalizacji testowania mutacyjnego. Samo zagadnienie minimalizacji liczby testów nie jest stosowane tylko w celach optymalizacji testów mutacyjnych. Sam proces testowania także wymaga takiej optymalizacji ze względu na czas wykonywania poszczególnych testów, których nadmiarowość może znacznie wydłużać cały proces[4]. Wobec tego metody redukcji liczby testów badane są już

[4] Przy bardzo dużych projektach przewidywany czas wykonania testów mierzony jest w godzinach, a nawet dniach. Często suity testów regresji są tak duże, że samo ich wykonanie — nie mówiąc o przeprowadzeniu

od lat. Ciekawa praca podsumowująca te badania została napisana przez Singha [50]. W pracy tej zostało pokrótce przytoczonych dziewiętnaście innych prac zajmujących się zagadnieniem minimalizacji zestawów testowych.

4.6 Optymalizacja poprzez priorytetyzację wykonywanych testów

Priorytetyzacja wykonania testów na zmutowanym kodzie jest jedną z często stosowanych metod optymalizacji procesu testowania mutacyjnego. Tego typu optymalizacją zajmuje się wielu badaczy, a stosowana jest w wielu rodzajach oprogramowania. Przykładem oprogramowania do testów mutacyjnych stosującego tę optymalizację (polegającą na uruchamianiu testów w jak najwydajniejszej kolejności) jest oprogramowanie PIT [44]. W przeciwieństwie do podejścia stosowanego w aplikacjach takich jak MAJOR [40], gdzie optymalizacja polega na zmniejszeniu liczby mutantów oraz usuwaniu mutantów generowanych przez równoważne operatory mutacyjne, autorzy oprogramowania PIT skupili się na uruchamianiu testów w taki sposób, aby te z nich, które wykonują się jak najszybciej i wykrywają jak najwięcej mutantów, były wykonywane jako pierwsze.

Platforma PIT przerywa testowanie danego mutanta z chwilą pierwszego negatywnego rezultatu któregoś z testów, co oznacza, że mutant został wykryty przez jakiś test, a dalsze testowanie tego mutanta nie ma już sensu. Wobec tego, ustalenie kolejności wykonania testów jest kluczowe. Framework PIT przyjmuje heurystykę uruchamiania testów mniej kosztownych, w sensie czasu ich wykonania na oryginalnym programie. Algorytm, który jest stosowany do tego celu, można opisać następującym wzorem: test t_0 będzie miał wyższy priorytet niż test t_1, jeżeli waga $w(t_0)$ testu t_0 będzie większa niż waga $w(t_1)$. Formalnie można zapisać to jako:

$$t_0 \succ t_1 \iff T(t_0) - T(t_1) - W_{dif}(t_0, t_1) > 0.$$

W podanym wzorze T oznacza czas wykonania testu, a $W_{dif}(t_0, t_1) = W(t_0) - W(t_1)$, gdzie W oznacza wagę danego testu wyrażaną wzorem

$$W(t) = \begin{cases} 1000 - c(t)/10 \; gdy \; DH(t) \\ -c(t)/10 \; w \; przeciwnym \; przypadku. \end{cases}$$

We wzorze tym c oznacza liczbę pokrytych bloków kodu przez test, natomiast wyrażenie $DH(t)$ jest prawdą wtedy i tylko wtedy, gdy dla danej klasy testowej, do której należy test t, istnieje klasa w kodzie źródłowym z taką samą nazwą (jeżeli testowanie jest wykonywane w frameworku JUnit v. 4.0) lub bez prefiksu "`Test`" (jeżeli program jest testowany w JUnit v. 3.0).[5]. Innymi słowy, wartość $DH(t)$ określa, czy dany test należy do klasy testów zaprojektowanych do bezpośredniego testowania konkretnej klasy.

jakiejkolwiek analizy mutacyjnej — trwa tak długo, że wykonywane jest w nocy, poza godzinami pracy zespołu.

[5] W JUnit 4.0 korzysta się z tzw. adnotacji `@Test` przed każdą metodą będącą testem jednostkowym. Natomiast w JUnit 3.0 metody, które są testami, muszą zaczynać się od słowa kluczowego `Test`.

Powyższy algorytm tworzy ranking skuteczności testów pod względem czasu wykonania na niezmutowanym kodzie, a następnie stosuje ją do testowania mutantów. Ciekawym usprawnieniem tego algorytmu jest zaproponowany przez Marcina Datę algorytm opisany w pracy [51]. Autor zaproponował własną metodę optymalizacji kolejności oraz czasu wykonania testów na zmutowanym kodzie, a następnie porównał ją z czterema innymi: tą stosowana w oprogramowaniu PIT, losową, optymalną oraz najgorszą[6]. Metoda została oparta na klasyfikacji czasu wykonania testu oraz na wynikach testów regresji.

W pierwszym kroku tego rozwiązania wykonuje się statystyczną analizę kodu źródłowego oraz testów. W tym celu wykorzystywane jest oprogramowanie SonarQube 4.5.6, z dodatkowym pluginem SourceMeter w wersji 8.0. Następnie, na podstawie analizy zdefiniowanych w pracy [51] (str. 23-24) standardowych metryk kodu: LCOM5[7], McCC[8], WMC[9], CBO[10], NOI[11], RFC[12], M[13], DIT[14] oraz LOC[15] definiowany jest zestaw cech dla modelu, który staje się pewnego rodzaju „identyfikatorem" mutanta. W kolejnym kroku, przy pomocy oprogramowania PIT przeprowadza się analizę mutacyjną na pewnej części programu, zapisując jaki mutant z jakim zestawem cech jak szybko został zabity przez dany test. Tworzona zostaje w ten sposób baza danych, która służy modelowi do predykcji w dalszej części testów mutacyjnych. W modelu kluczowy jest fakt, że dzięki analizie kodu i przypisaniu konkretnym mutantom cech można przewidywać, że skoro dany test był szybki i skuteczny wykrywając danego mutanta, to powinien posiadać podobną skuteczność, testując inne mutanty o cechach zbliżonych do wcześniej wykrytego. W modelu testy są klasyfikowane według skuteczności: najbardziej skuteczne będą przesuwane na szczyt kolejki uruchomień, a uznane przez model za niezdolne do zabicia mutanta nie będą uruchamiane w ogóle.

[6]Wykonania optymalne oraz najgorsze zostały opracowane w sposób ręczny post factum dla konkretnego programu.

[7]Lack of Cohesion in Methods 5 – miara powiązania atrybutów klasy i jej metod. Im większa wartość miary, tym niższa kohezja. Pożąda się sytuacji gdy kohezja jest duża. Formalnie, LCOM5 jest zdefiniowana następująco: $LCOM5(C) = \frac{1}{|M|-1}(|M| - \frac{1}{|A|}\sum_{a \in A} \mu(a))$, gdzie C – dana klasa, M – zbiór metod tej klasy, A – zbiór atrybutów, $\mu(a)$ – liczba metod wykorzystujących atrybut a.

[8]McCabe's Cyclomatic Complexity – złożoność cyklomatyczna McCabe'a. Jest obliczana dla metod lub klas. Wyrażana jest jako liczba tzw. ścieżek liniowo niezależnych lub, równoważnie, jako liczba punktów decyzyjnych powiększona o 1. Wysoka wartość tej metryki może oznaczać skomplikowaną strukturę kodu.

[9]Weighted Methods per Class – miara złożoności danej klasy oraz jej poszczególnych metod. Wyraża sumę złożoności wszystkich metod zawartych w danej klasie. Złożoność może być definiowana w różny sposób, np. przy pomocy metryki McCabe'a.

[10]Coupling Between Object classes – liczba klas sprzężonych z daną klasą, tzn. liczba obcych klas, które używają metod lub atrybutów danej klasy. CBO jest wskaźnikiem ilości pracy jaką trzeba włożyć w utrzymanie i testowanie danej klasy.

[11]Number of Outgoing Invocations – liczba unikalnych, bezpośrednich wywołań metod zewnętrznych.

[12]Response For a Class – zlicza wywołania metod obcych klas. Wyraża stopień interakcji metod klasy z innymi metodami.

[13]M – liczba wszystkich metod w danej klasie.

[14]Depth of Inheritance Tree – głębokość w drzewie dziedziczenia dla danej klasy.

[15]Lines of Code – liczba wykonywalnych linii kodu w klasie.

Rozdział 5

Wiele mutantów w jednej kompilacji

Interesującą metodą procesu testowania mutacyjnego może być podejście polegające na generowaniu wielu mutantów w jednej kompilacji. Dzięki temu oszczędza się czas na kompilacji kodu, a przy pomocy specjalnych, dodatkowych parametrów wywołania można sterować tym, który z zaimplementowanych mutantów ma być wykonany podczas uruchomienia kodu. Zagadnienie wielu mutantów w jednej kompilacji może być rozwiązywane na dwa sposoby: przy pomocy mutacji wyższego rzędu lub techniki mutacji warunkowej. Mutacje wyższego rzędu są opisane w rozdziale 3.6. Niestety korzystanie z tej techniki niesie ze sobą pewne problemy, a czasem nawet może doprowadzić do zmniejszenia wydajności procesu mutacji. Zastosowanie tego podejścia jest omówione w pracy Nguyena oraz Madeyskiego [52]. Natomiast w tym rozdziale zostanie zaproponowany model optymalizacji procesu testowania mutacyjnego oparty na generowaniu wielu mutantów w jednej instancji (kompilacji) programu, polegający na generowaniu dodatkowych bloków kodu zawierających wszystkie mutanty, które mogły być wygenerowane w danym fragmencie kodu. Rozdział opisuje założenia modelu optymalizacyjnego, teoretyczny model wraz z opisem implementacji prototypu oprogramowania oraz wyniki badań mających na celu zbadanie poprawności modelu i jego założeń. Podobny proces generacji wielu mutantów w jednej kompilacji zaproponowany został w pracy [53]. Niestety autorzy w tej pracy nie zastosowali swojego modelu w praktyce, a jedynie modyfikowali ręcznie bloki kodu, które mają być zmutowane. Badania zawarte w tym rozdziale maja natomiast mają na celu wytworzenie implementowanego modelu oraz przetestowanie jego skuteczności.

5.1 Założenie problemu

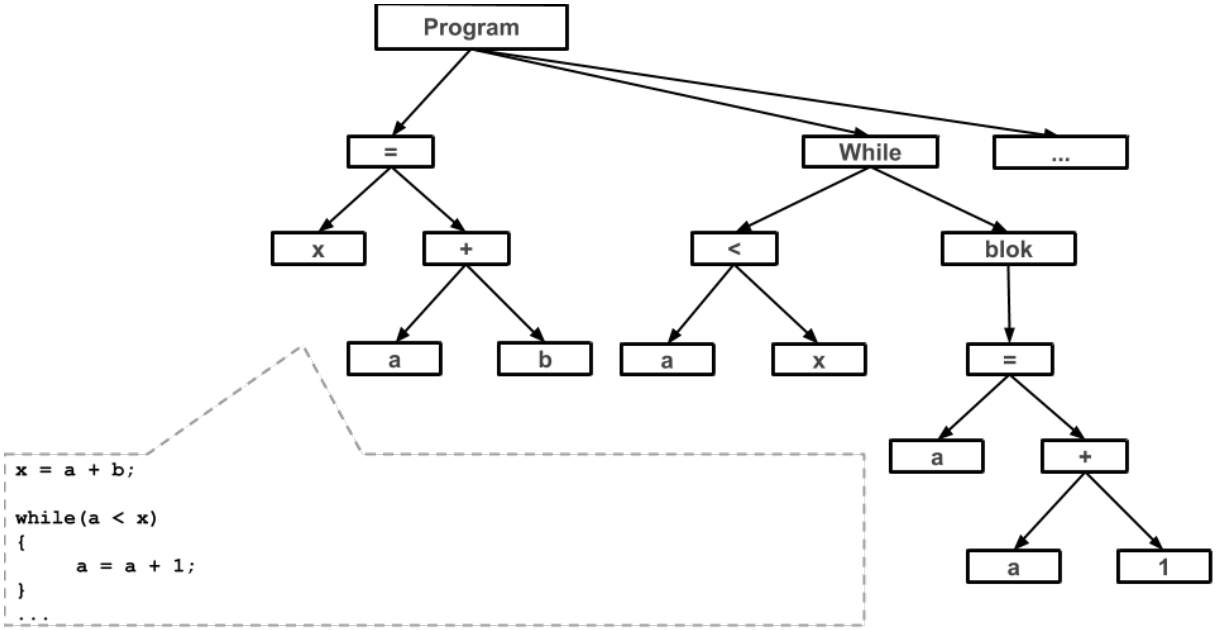
Generowanie wielu mutantów w jednej kompilacji programu dokonane będzie przy użyciu techniki mutacji warunkowej. Podejście to pozwala zredukować istotnie czas przeprowadzenia analizy mutacyjnej, ponieważ unika się wielokrotnego, czasochłonnego kompilowania kodu, a jedynie wykonuje się wielokrotnie jeden, raz skompilowany kod. Proces ten polega na wstrzykiwaniu w miejsca, w których następuje generacja mutanta, bloków kodu reprezentującego wszystkie możliwe warianty mutacji. Następnie, przy uruchomieniu programu za pomocą zewnętrznej zmiennej sterującej określa się, który mutant ma być aktywowany.

Koncepcja generowania jednej kompilacji dla całego procesu mutacyjnego jest szczególnie istotna w kontekście przeprowadzania tego procesu dla języków, które nie są oparte na żadnej formie kodu pośredniego uruchamianego na wirtualnej maszynie. W tych przypadkach nie można bowiem przeprowadzać mutacji bezpośrednio na bajtkodzie. Stąd zastosowanie tego podejścia w takich językach jak Java, C# lub VisualBasic jest znikome. Natomiast spory potencjał dla zastosowania omawianej metody istnieje w przypadku, gdy testowanie dotyczy programu napisanego w języku niższego poziomu. Takimi językami są C, C++, Pascal, Fortran oraz wszystkie języki, których kod podczas procesu kompilacji jest tłumaczony bezpośrednio na kod maszynowy. Dla języków skryptowych opartych na interpreterze generowanie wielu mutantów w jednej kompilacji niczym się nie rożni od generowania wielu kompilacji. Jest to podyktowane tym, że proces kompilacji w językach skryptowych jest ograniczony lub w ogóle nie istnieje. Zastąpiony jest on przez proces interpretacji kodu, która polega na czytaniu instrukcji zawartych w programie linia po linii, a następnie wykonywaniu ich. Wobec powyższego, dla języków skryptowych każde uruchomienie programu jest zarazem rozpoczęciem procesu interpretacji kodu, co sprawia, że proces generowania wielu mutantów w jednej kompilacji nie ma tu zastosowania.

5.2 Teoretyczny model budowy wielu mutantów w jednej kompilacji

Teoretyczny model generowania wielu mutantów w jednej kompilacji można zbudować na podstawie analizy działania poszczególnych operatorów mutacyjnych. Technicznie w podejściu tym generacja mutantów następuje poprzez transformację drzewa składni abstrakcyjnej (AST).

Definicja 5.1 (Drzewo składniowe)**.** Drzewo AST (ang. abstract syntax tree), czyli etykietowane drzewo składni abstrakcyjnej, to drzewiasta struktura przedstawiająca wynik przeprowadzenia analizy składniowej zdań zgodnie z zadaną gramatyką. Każdy węzeł wewnętrzny drzewa jest reprezentacją pewnej konstrukcji leksykalnej.



Rysunek 5.1: Przykładowe drzewo AST dla zadanego kodu

Analizując AST dla konkretnego programu, można wygenerować na podstawie kodu oprogramowania zmutowany kod, w którym wszystkie mutanty znajdują się obok siebie, „zamknięte" w instrukcjach warunkowych `if` lub `switch`. Przykład takiego kodu prezentuje listing 5.1.

Listing 5.1: Pseudokod opisujący mechanikę optymalizacji polegającej na generowaniu wielu mutantów w jednej kompilacji

```
public int eval(int x){
int a = 3, b = 1, y;
y = (M_NO==1)? a + x: // mutant 01
(M_NO==2)? a / x: // mutant 02
(M_NO==3)? a % x: // mutant 03
a * x; // bez mutacji
y += b;
return y;
}
```

W powyższym kodzie M_{NO} jest zmienną sterującą oznaczającą mutanta, który ma być uruchomiony. Proces testowania mutacyjnego w tym wypadku przebiega w następujący sposób:

Listing 5.2: Proces testowania mutacyjnego z wieloma mutacjami w kompilacji

```
1. Zdefiniuj M_NO := 1.
2. Uruchom testy dla mutanta M_NO.
3. Wygeneruj raport.
4. M_NO := M_NO + 1.
5. Jesli M_NO przekracza wartosc liczby mutantow, zakoncz.
6. W przeciwnym razie wroc do kroku 2.
```

Powyższa technika została opisana w pracy Kapfhammera, Justa i Schweiggerta [54].

Na podstawie analizy operatorów mutacyjnych zaproponowanych przez Ammanna i Offutta [14] oraz opisanych w podrozdz. 3.3 można określić, które mutanty mogą być tworzone w jednej kompilacji, a które nie i dlaczego. Poniższa lista jest teoretycznym fragmentem badań zawartych w tej pracy i nie odnosi się do innych badań. Mechanika działania wszystkich poniżej analizowanych operatorów mutacyjnych opisana jest w podrozdz. 3.4.

- AMC – Access Modifier. Operatory mutacyjne operujące na polimorfizmach, poziomach dostępów oraz wszystkich szkieletowych fragmentach kodu nie mogą istnieć razem w jednej kompilacji, ponieważ wymagałyby tworzenia klas o takich samych sygnaturach, ale o innych poziomach dostępu lub innego rodzaju polimorfizmów. Pojedyncze operatory mutacyjne tego typu mogą istnieć w jednej kompilacji z innymi mutantami, ale takimi, które nie modyfikują logiki dostępu do klas lub polimorfizmów.

- HVD – Hiding Variable Deletion oraz HVI – Hiding Variable Insertion. Oba przypadki określają mutanty, które usunęły lub dodały nadpisane zmienne. Aby mutanty generowane przez ten operator zawrzeć w jednej mutacji oraz aby mogły współistnieć z mutantami innego typu, należy przy każdym użyciu zmiennej, która nadpisuje zmienną z klasy bazowej, dodać warunek określający, czy mutacja ma nastąpić w tym miejscu czy nie [1]. Dla HVI

[1] Tego typu operacje można zastosować głównie w językach takich jak C#, C++ oraz Java

nadpisywane zmienne muszą zostać uprzednio zadeklarowane w kodzie. Pseudokod opisujący jak wiele mutantów tego typu może istnieć w jednej kompilacji, wygląda następująco.

Listing 5.3: Pseudokod dla przypadku HVD

```
if (MUT) { Uzyj zmiennej z danej klasy}
else { Uzyj zmiennej z klasy bazowej }
```

- OMD – Overriding Method Deletion. Operator usuwa metody nadpisujące metody z klasy bazowej. Wiele mutantów tego typu może istnieć w jednej kompilacji. W ciele każdej metody na początku należy dodać sekcję odwołującą się do metody w klasie bazowej, jeżeli dany mutant ma być użyty. Kod takiego rozwiązania może wyglądać następująco.

Listing 5.4: Pseudokod dla przypadku OMD

```
Metoda( Parametry ) {
   if(MUT) {
     Uruchom metode z klasy bazowej o tej samej sygnaturze.
       return;
    }
    Cialo metody nadpisujacej.
}
```

- OMR – Overridden Method Rename. Operator mutacyjny zmienia nazwy w klasach bazowych metod, które są nadpisywane. Ma to na celu zbadanie, czy program odwołuje się poprawnie do metod z klas nadrzędnych. Mutanty tego typu symulują błąd odwołania się do klasy nadrzędnej, podczas gdy celem było odwołanie się do klasy bazowej i na odwrót. Tego typu mutanty weryfikują poprawne zastosowanie projektowanych polimorfizmów i podobnie jak AMC wymagają osobnej kompilacji dla każdego mutanta. Natomiast z innymi mutantami może występować tylko jeden mutant typu OMR.
- OMM – Overridden Method Moving. Dobry przykład, który można zawrzeć w jednej kompilacji. Celem tej mutacji jest sprawdzenie, czy metody, które nadpisują metody bazowe zarazem wywołujące metodę bazową, nie robią tego w nieprawidłowy sposób. Przykładowy kod dla tego przypadku prezentowany jest na listingu 5.5

Listing 5.5: Pseudokod dla przypadku OMM

```
Kod niezmutowany:
class A { ... Void Metoda01(){ cialo } }
class B : A {
    overide Void Metoda01()
    {
        base.Metoda01();
        cialo nadpisane
    }
}

Mutant:
class A { ... Void Metoda01(){ cialo } }
class B : A {
   overide Void Metoda01(MutParam)
    {
        if(MutParam==1) base.Metoda01();
        cialo nadpisane
        if(MutParam==2) base.Metoda01();
    }
}
```

Mutantów tego typu można wytworzyć wiele w jednej kompilacji. Wywołanie metody z klasy bazowej może nastąpić praktycznie w dowolnym miejscu. Innymi słowy, dla OMM w jednej kompilacji można wytworzyć n mutantów dla każdej metody, gdzie n to liczba linii kodu mutowanej metody. OMM może występować w jednej kompilacji z innymi operatorami mutacyjnymi.

- SKD – Super Keyword Deletion Operator. W zależności od języka programowania usuwa słowo kluczowe `super` lub `base`. Może zostać „wpleciony” w OMM. Realizacja tego przypadku polega na usunięciu słowa `super` albo `base` w zależności od języka. Utworzenie wielu mutantów tego typu w kodzie za pomocą instrukcji warunkowych odbywać się będzie podobnie jak dla OMM. Mechanika takiego kodu powinna bazować na pseudokodzie zaprezentowanym na listingu 5.6.

Listing 5.6: Pseudokod dla przypadku SKD

```
Kod nie zmutowany:
class A { ... Void Metoda01(){ cialo } }
class B : A {
override Void Metoda01(){
    base.Metoda01();
    cialo nadpisane
    }
}

Mutant:
class A { ... Void Metoda01(){ cialo } }
class B : A {
override Void Metoda01(MutParam) {
   if(Mut) Metoda01(); //Mutant aktywny, usuwamy base
   else base.Metoda01(); //Mutant nieaktywny
   cialo ...
    }
}
```

Aby SKD zastosować w wariancie wielu mutantów w jednej kompilacji, dla każdego wywołania metody z klasy bazowej powinien być zastosowany taki warunek jak w przykładzie powyżej. Mutant może istnieć z mutantami innego typu w jednej kompilacji.

- PCD – Parent Constructor Deletion. W językach takich jak C# i Java ten operator mutacyjny pod względem mechaniki jest identyczny z SKD. Kod takiego mutanta w tych językach będzie wyglądał w sposób następujący:

Listing 5.7: Pseudokod dla przypadku PCD

```
//Java, kod niezmutowany:
class A { public A(){} }
class B extends A {
    public B(){
        Super();
    }
}
//Mutant:
class A { public A(){} }
class B extends A {
   public B(){
         if(MUT) {
        }
         else{
         Super();
      }
    }
}
```

Ten sam kod będzie działał w C# z tą różnicą, że zamiast `super` trzeba użyć słowa kluczowego `base` oraz zamiast `extends` znaku dwukropka „:”. W przypadku C++ mutant niestety będzie musiał być generowany per kompilacja, co wynika z budowy języka. Dziedziczenie w C++ wygląda bowiem następująco: `Subclass(int foo, int bar) : Superclass(foo)`. W C++ dyrektywa dziedziczenia jest zawarta w deklaracji konstruktora.

- ATC – Actual Type Change. Ten operator mutacyjny może generować wiele mutantów w jednej kompilacji, które mogą występować z innymi mutantami tylko w językach, w których występuje słabe typowanie zmiennych[2]. Mechaniczne wstawianie wielu mutantów do jednej kompilacji będzie podobne jak w przypadku SKD, OMM i PCD. Poniższy pseudokod obrazuje ten mechanizm.

Listing 5.8: Pseudokod dla przypadku ATC

```
public void Metoda01(){
   var a = int(10);
      if(MUT) {
     var a = new long(10);
    }
}
```

Tego typu mutanty głównie będą symulować błędy związane z oczekiwanym typem zmiennej lub z niewłaściwymi wartościami wychodzącymi poza zakresy danych typów.

- DTC – Declared Type Change. Zmiana typu zmiennej zadeklarowanej zawsze musi generować osobą kompilację, bowiem nie mogą istnieć dwie zmienne o tej samej nazwie zadeklarowane dla dwóch różnych typów.

- PTC – Parameter Type Change. W tym wypadku mamy ten sam problem, co w przypadku DTC.

- RTC – Reference Type Change. Ten operator mutacyjny zmienia prawą stronę przypisania referencji, wobec czego operacja przypisania może zostać zamknięta w bloku warunkowym `if`. Mutacja będzie polegać na uruchamianiu odpowiedniego `if` w zależności od zmiennej sterującej, a następnie zgodnie z instrukcją zawartą w bloku `if` nastąpi przypisanie do zmiennej referencji na zmienną o tej samej wartości, ale innym typie[3]. Mutanty tego typu mogą występować razem w jednej kompilacji wraz z mutantami pochodzącymi od innych operatorów. Mechanikę takich operatorów obrazuje listing 5.9:

[2]System typów, w którym typ danej zmiennej nie jest jasno ustalony i może zmieniać się w trakcie działania programu. Występuje głównie w językach skryptowych, takich jak PHP czy JavaScript. Występuje także w C# jako deklaracja zmiennych lokalnych typu ”`var`”

[3]Przypisywany typ zmiennej musi być kompatybilny z typem zmiennej, do której dane są przypisywane. W przeciwnym wypadku nastąpi błąd kompilacji. Dobrym przykładem kompatybilnego typu jest typ obiektu nadrzędnego w hierarchii dziedziczenia.

Listing 5.9: Pseudokod dla przypadku RTC

```
public void Metoda01(){
    a = b
    if(MUT01) {
        a = b_prim
    }
    if(MUT02) {
        a = b_bis
    }
    ...
}
```

- OMC – Overloading Method Change oraz OMD – Overloading Method Deletion. W obu przypadkach ciała metod bazowych i przeciążanych muszą zostać zamknięte w bloku warunkowym `if`. Dla OMC w obu metodach będzie znajdować się kod metody przeciążanej i przeciążającej. Następnie, jeżeli mutant ma się pojawić, to warunek `if` wykona metody w taki sposób, że w metodzie przeciążanej uruchomione zostanie ciało metody przeciążającej i na odwrót. Dla OMD w przypadku wystąpienia mutanta warunek wykona metodę przeciążaną. Pseudokod tego mechanizmu może wygląda następująco:

Listing 5.10: Pseudokod dla przypadku OMC

```
//Bez mutacji:
public void Metoda01(int a, b){ //Pierwsza metoda glowna INT
   cialo ... }
public void Metoda01(double a, b){//DOUBLE
   cialo ... }
//Mutant:
public void Metoda01_Helper(int a, b){//INT
   cialo ... }
public void Metoda01_helper(double a, b){//DOUBLE
   cialo ... }
public void Metoda01(int a, b){
   if(MUT==OMC) Metoda01_helper(double a, b);
    if(MUT==OMD) Metoda01_Helper(int a, b); //Uzywamy tylko glownej metody
        pierwszej zadeklarowanej
    else Metoda01_Helper(int a, b);
}
public void Metoda01(double a, b){
   if(MUT==OMC) Metoda01_Helper(int a, b);
    if(MUT==OMD) Metoda01_Helper(int a, b); //Uzywamy tylko glownej metody
        pierwszej zadeklarowanej
    else Metoda01_helper(double a, b);
}
```

Ten mechanizm można rozbudowywać do dowolnej liczby metod. Wzrośnie wtedy tylko liczba kombinacji wywołań. Wiele mutantów tego typu operatora może występować w

jednej kompilacji. Mutanty innych operatorów także mogą współistnieć w jednej kompilacji z tymi mutantami.

- AOC – Argument Order Change oraz ANC – Argument Number Change. Do tych operatorów można zastosować mechanizm taki jak w powyższym przykładzie, z tą różnicą, że przeładowywać będziemy zmutowane wywołania metod. Ciało każdej metody powinno zostać przeniesione do metody z tagiem „`_Helper`", a następnie metoda ta powinna zostać wywołana w oryginalnej metodzie oraz w mutantach. Tak jak powyższe, mutanty tego typu nie muszą wytwarzać osobnej kompilacji i są kompatybilne z innymi operatorami mutacyjnymi.

- TKD – `this` Keyword Deletion. Tworzenie wielu mutantów tego operatora mutacyjnego w jednej kompilacji jest analogiczne do SKD, z tym że w tym przypadku zamiast `base` usuwamy `this`.

- SMC – Static Modifier Change. Ten operator działa na deklaracji statyczności klasy rozpatrywanej na poziomie kompilacji kodu. Każdy mutant tego typu zawsze będzie tworzył osobną kompilację.

- VID – Variable Initialization Deletion. Utworzenie wielu mutantów w jednej kompilacji dla tego operatora jest możliwe poprzez dodanie w konstruktorze obiektu bloku ustawiającego wszystkie zmienne składowe na `null` lub ich wartość domyślną. Mutant w większości przypadków spowoduje rzucenie wyjątku w trakcie pracy programu, ale może występować wraz z mutantami innego typu.

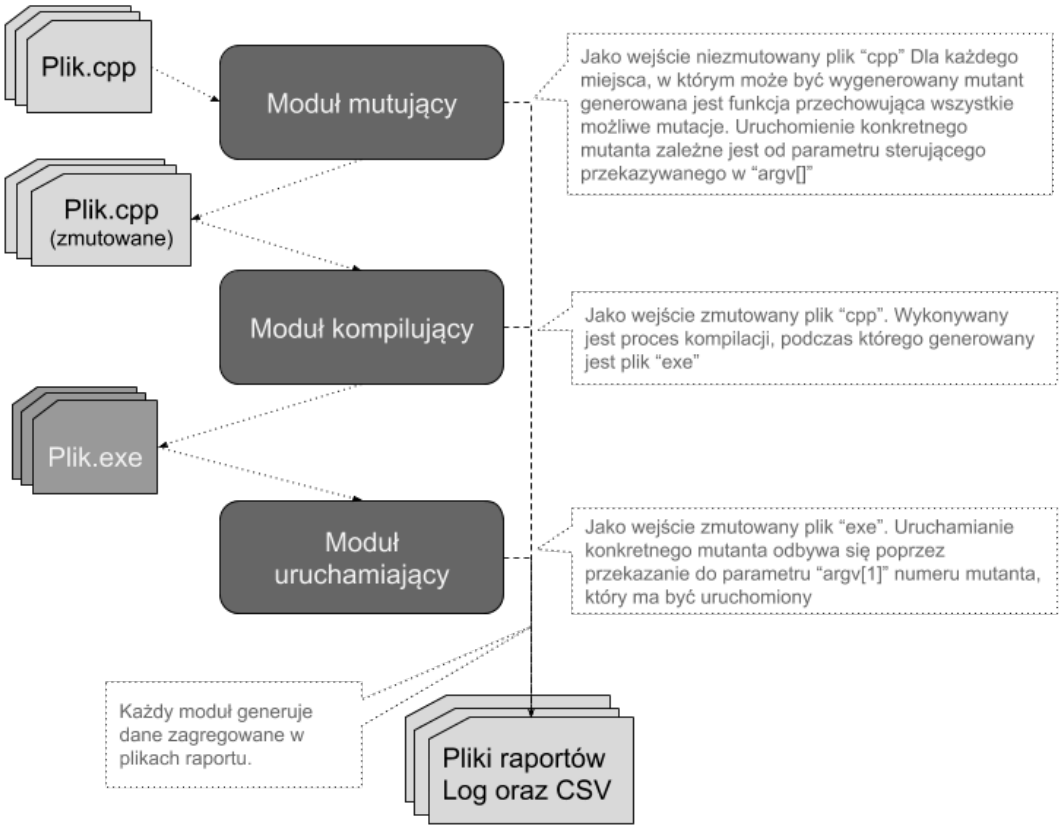
- DCD – Default Constructor Deletion. Istnieje teoretyczna możliwość tworzenie wielu mutantów tego typu w jednej kompilacji. Polegałaby ona na tym, że przy deklaracji każdego konstruktora domyślnego trzeba dodać jakikolwiek parametr. Następnie, jeśli mutant ma się nie pojawić, to należy uruchomić konstruktor z dodanym wcześniej parametrem. Jeśli zaś mutant ma się pojawić, to należy uruchomić konstruktor domyślny, który teraz nie jest zadeklarowany.

Zaproponowany powyżej model może być podstawą do budowy systemu testów mutacyjnych stosującego metodę tworzenia mutantów optymalizowaną poprzez generowanie jak największej liczby mutantów w jednej kompilacji.

5.3 Platforma

Na potrzeby zbadania efektywności procesu testowania mutacyjnego opartego o generację wielu mutantów w jednej kompilacji została stworzona platforma pozwalająca na transformację kodu programu w sposób pozwalający na przeprowadzanie procesu testowania mutacyjnego w jednej instancji oprogramowania bez konieczności ingerencji w bajtkod lub generowania każdorazowo nowej kompilacji. Platforma nosi nazwę MMIOC co jest akronimem od "Many Mutants In One Compilation" Platforma składa się z trzech modułów: generującego mutanty, kompilującego oraz

uruchamiającego. Oprogramowanie mutuje kod w C++. Diagram funkcjonowania modułów przedstawiono na rys. 5.2.



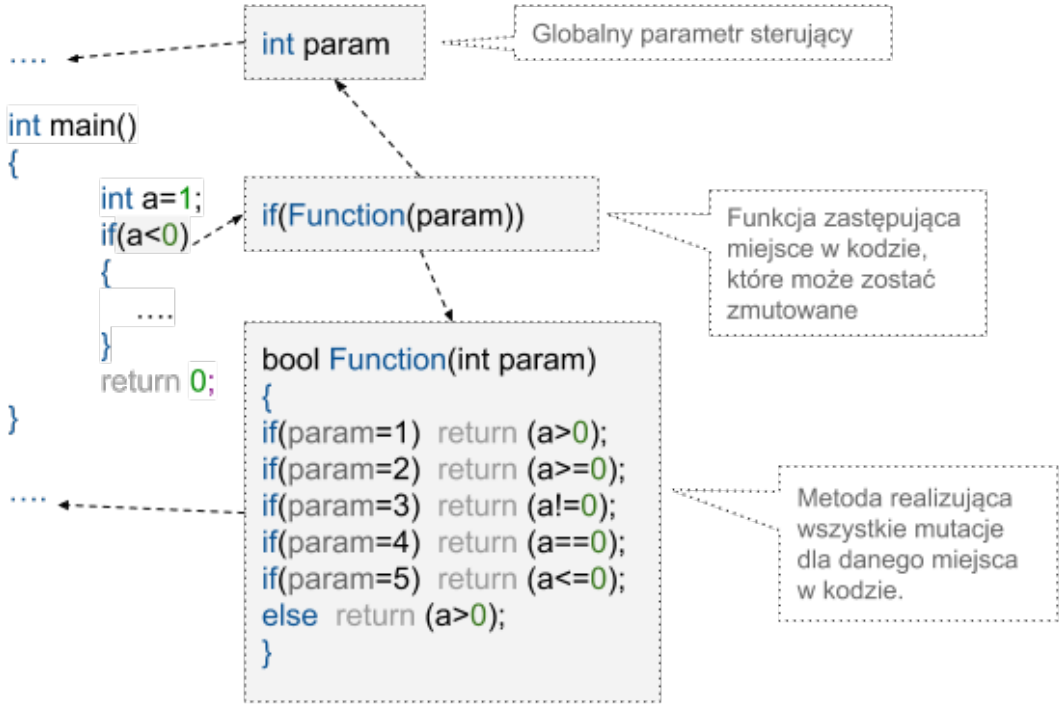
Rysunek 5.2: Funkcjonowanie platformy mutującej

5.3.1 Moduł mutujący

Ten moduł przyjmuje pliki z rozszerzeniem „cpp", a następnie wykonuje na nich proces mutacji według następującego algorytmu:

- Kod programu jest cięty na bloki. Blokiem jest metoda lub klasa.
- Dla każdego bloku wykrywane są miejsca w kodzie, które mogą zostać zmutowane.
- Dla każdego miejsca w kodzie mogącego wygenerować mutacje generowana jest funkcja, która przetrzymuje wszystkie możliwe mutacje tego fragmentu kodu. Każdy mutant w takiej funkcji jest sygnowany numerem służącym do wywoływania określonej mutacji. Numer jest wartością zmiennej globalnej dla całego programu. Wszystkie funkcje trzymające mutacje także są globalnie numerowane. Każda generacja takiej funkcji odbywa się na podstawie analizy sygnatury danego fragmentu kodu, dzięki czemu wszystkie zmienne używane w tym fragmencie kodu zostają przekazane do nowo generowanej metody.
- Wszystkie elementy kodu mogące wygenerować mutację są podmieniane na wywołanie wygenerowanej uprzednio funkcji. Następnie ciało każdej funkcji jest doklejane na końcu pliku.

Powyższy algorytm zobrazowany jest na rys. 5.3.



Rysunek 5.3: Algorytm mutacji

W celu lepszego zrozumienia działania algorytmu mutującego poniżej zostało zaprezentowane działanie procesu mutacji zgodnie z opisywanym w tym dziale algorytmem. Listing 5.11 prezentuje kod przed mutacją, natomiast listing 5.11 prezentuje zmutowany kod.

Listing 5.11: Kod przed mutacją

```
using namespace std;

int main(int argc, char* argv[])
{
    char* cc = argv[1];
   int a=1;

   if(a<0) //mutowane miejsce
   {
       cout << "IF a<0";
   }

   return 0;
}
```

Listing 5.12: Kod po mutacji

```
 using namespace std;

 bool f1(int param,int a);
 bool f2(int param,int a);

 char **argv;
 string g_argv;

int main(int argc, char* argv[]){
g_argv = argv[1]; //dla mutacji

   if(f1(atoi (g_argv.c_str()), a )){
       cout << "IF a<0";
   }

   if(f2(atoi (g_argv.c_str()), a )){
       cout << "IF a>0";
   }
   return 0;
}

// Blok kodu mutacji ===================================
 bool f1(int param,int a){
if(param==0){return (a<0);}
else if(param==1){return (a<=0);}
else if(param==2){return (a>=0);}
else if(param==3){return (a!=0);}
else if(param==4){return (a==0);}
else {return (a<0);}
}

 bool f2(int param,int a){
if(param==5){return (a<0);}
else if(param==6){return (a<=0);}
else if(param==7){return (a>=0);}
else if(param==8){return (a!=0);}
else if(param==9){return (a==0);}
else {return (a>0);}
}
// Blok kodu mutacji ===================================
```

5.3.2 Moduł kompilujący

Moduł kompilujący jako pliki wsadowe przyjmuje zmutowane pliki „cpp” a następnie za pomocą zewnętrznego kompilatora na przykład „gcc” uruchamia proces kompilacji kodu zwracając skompilowany, wykonywalny plik exe. Skompilowany plik w języku c++ jest skompresowanym zestawem

plików „obj” (object code) zawierających instrukcje procesora w formie kodu maszynowego. Przykładowa komenda kompilująca wygląda następująco:

`g++ -o .\Ścieżka do pliku\Plik .\Ścieżka do pliku\Plik.cpp`.

Następnie taki plik może zostać uruchomiony z określonym parametrem sterującym wywołaniem konkretnego mutanta.

5.3.3 Moduł uruchamiający

Zadaniem tego modułu jest uruchamianie skompilowanego pliku z konkretną wartością parametru sterującego wywołaniem mutantów. Każdy mutant ma przypisany do siebie numer od 1 do maksymalnej liczby mutantów. Uruchomienie programu za pomocą komendy `prog.exe 12` sprawi, że zostanie aktywowany mutant nr 12 i program zostanie wykonany zgodnie z zadaną mutacją. Zmutowane programy można uruchamiać w różnych konfiguracjach ręcznie po jednej mutacji lub grupowo (wszystkie mutacje za pomocą skryptów systemowych).

Opis komend do paragramu wraz z wywołaniami znajduje się w dokumentacji użytkowej projektu zawartej w dodatku B.

5.4 Eksperyment

Celem poniższego eksperymentu było sprawdzenie efektywności procesu testowania mutacyjnego z użyciem metody generacji wielu mutantów w jednej kompilacji. Efektywność tego procesu może być zbadana poprzez pomiar różnic czasu wykonania generacji mutantów, kompilacji zmutowanego kodu oraz uruchamiania kodu dla każdego wygenerowanego mutanta z użyciem generacji wielu mutantów w jednej kompilacji oraz z użyciem metody tradycyjnej. Eksperyment został przeprowadzony na programach napisanych w języku C++, bowiem jego między innymi właśnie dotyczy problem związany z brakiem możliwości modyfikowania programu na bajtkodzie. Według naszej najlepszej wiedzy wszystkie programy pozwalające na mutowanie oprogramowania napisanego w C++ opierają się na generowaniu wielu kompilacji tego samego kodu, z reguły przy pomocy specjalnie zmodyfikowanego pod ten proces kompilatora. Przykładowo, w publikacji [55] autorzy odnoszą się właśnie do tego procesu przy użyciu autorskiej wersji kompilatora opartego o Clang/LLVM framework. Na potrzeby tego eksperymentu platforma MMIOC została zmodyfikowana w taki sposób, że jednocześnie generuje plik zawierający wiele mutacji w jednej kompilacji oraz zestaw katalogów zawierających każdorazowo skompilowanych plik z pojedynczą mutacją. Oczywistym jest, że wykonanie jednokrotnej kompilacji będzie szybsze niż kompilacja i wykonanie wielu plików. Jedynym problemem może być czas generacji zagregowanego pliku zawierającego wszystkie możliwe mutacje oraz późniejsze jego uruchomienia. Poniższy eksperyment w szczególności pozwala ocenić istotność tego problemu.

5.4.1 Przebieg eksperymentu

Eksperyment został wykonany na dwóch zestawach danych. Pierwszy z nich zawierał 8 syntetycznych plików „cpp”, w których znajdowały się proste programy z instrukcjami decyzyjnymi, które

następnie były mutowane. W pierwszym programie znajdowała się tylko jedna instrukcja mogąca generować 6 mutantów. W drugim programie znajdowały się już dwie takie instrukcje mogące sumarycznie wygenerować 12 mutantów[4]. Natomiast w ósmym programie takich instrukcji był już 10, co generowało łączną liczbę 60 mutantów. Dokładna liczba instrukcji mogących generować mutanty znajduje się w tabeli 5.4. Tę wartość opisuje metryka *NMP*. W programach 7 oraz 8 znajdowały się zagnieżdżone instrukcje warunkowe. Pliki z kodami źródłowymi użyte w tym eksperymencie noszą nazwy synt01.cpp, synt02.cpp do synt08.cpp.

Drugi zestaw kodów źródłowych to biblioteka z metodami sortującymi dane. W eksperymencie biblioteka została podzielona na 5 plików. Każdy z nich zawierał pewną liczbę algorytmów sortujących.

- Pierwszy plik o nazwie „sort01.cpp" zawierał tylko i wyłącznie algorytm sortowania bąbelkowego.
- Drugi plik o nazwie „sort02.cpp" zawierał algorytm sortowania bąbelkowego oraz algorytm sortowania szybkiego.
- Trzeci plik o nazwie „sort03.cpp" zawierał algorytm sortowania bąbelkowego, sortowania szybkiego oraz sortowania przez wybieranie.
- Czwarty plik o nazwie „sort04.cpp" zawierał algorytm sortowania bąbelkowego , sortowania szybkiego, sortowania przez wybieranie oraz sortowania przez scalanie.
- Piąty plik o nazwie „sort05.cpp" zawierał algorytm sortowania bąbelkowego, sortowania szybkiego, sortowania przez wybieranie, sortowania przez scalanie oraz sortowania stogowego.

Następnie dla każdego z zestawów danych został wykonany pełen proces mutacji kodu agregujący wszystkie mutanty do jednej kompilacji oraz generujący wiele kompilacji, z których każda zawierała jedną mutację. Kody w obu procesach zostały także skompilowane i uruchomione symulując wykonanie testów. Podczas całego procesu platforma zbierała dane dotyczące czasu wykonania mutacji kompilacji oraz wykonania programów. W eksperymencie wykorzystano następujące metryki.

- AMNA (Average Mutation Time - Not Agreagated) – średni czas (w ms) wykonania mutacji nie zagregowanych w jeden plik.
- AMA (Average Mutation Time - Agreagated) – średni czas (w ms) wykonania mutacji zagregowanych w jeden plik.
- TMNA (Total Mutation Time - Not Agreagated) – całkowity czas (w ms) wykonania mutacji nie zagregowanych w jeden plik.

[4]Nie każda instrukcja decyzyjna generuje taką sama liczbę mutantów. Przykładowo, instrukcje zawierające operatory takie jak jak `<, <=, >, >=, ==, !=` generują co najmniej 6 mutacji, Natomiast instrukcje takie jak `!bool value` oraz `bool value` generują tylko 2 mutanty

- TMA (Total Mutation Time - Agreagated) – całkowity czas (w ms) wykonania mutacji zagregowanych w jeden plik.
- ACNA (Average Compilation Time - Not Agreagated) – średni czas (w ms) kompilacji plików zawierających pojedyncze mutacje.
- ACA (Average Compilation Time - agreagated) – średni czas (w ms) kompilacji pliku zawierającego zagregowane mutacje.
- TCNA (Total Compilation Time - Not Agreagated) – całkowity czas (w ms) kompilacji plików zawierających pojedyncze mutacje.
- TCA (Total Compilation Time - Agreagated) – całkowity czas (w ms) kompilacji pliku zawierającego zagregowane mutacje.
- ARNA (Average Run Time - Not Agreagated) – średni czas (w ms) wykonania plików zawierających pojedyncze mutacje.
- ARA (Average Run Time - Agreagated) – średni czas (w ms) wykonania pliku zawierającego zagregowane mutacje.
- TRNA (Total Run Time - Not Agreagated) – całkowity czas (w ms) wykonania plików zawierających pojedyncze mutacje.
- TRA (Total Run Time - Agreagated) – całkowity czas (w ms) wykonania pliku zawierającego zagregowane mutacje.
- LOC (Lines Of Code) – liczba linii kodu.
- CycAVG (Cyclomatic Avrage Value) – wartość średnia dla metryk złożoności cyklomatycznej (patrz def. 5.2).
- CycMAX (Cyclomatic Max Value) – wartość maksymalna dla metryk złożoności cyklomatycznej (patrz def. 5.2).
- MN (Mutants Number) – liczba wszystkich wygenerowanych mutantów.
- NMP (Number of Mutated Places) – liczba wszystkich miejsc w kodzie, które mogą zostać zmutowane.

Definicja 5.2 (Złożoność cyklomatyczna (za [56]))**.** Złożoność cyklomatyczną (CC) definiujemy jako liczbę niezależnych ścieżek w programie. Jeśli program reprezentowany jest jako graf przepływu sterowania, to złożoność cyklomatyczną można wyrazić wzorem $L - N + 2P$, gdzie:
L — liczba krawędzi w grafie,
N — liczba wierzchołków grafu,
P — liczba rozłącznych części grafu (np. wywoływany graf lub procedura).

Ze względu na to, że czasy wykonania, kompilacji i uruchomienia programu są częściowo zależne od systemu operacyjnego oraz obciążenia procesora[5], w celu zminimalizowania wpływu tych zależności Eksperyment pierwszy i drugi były powtórzone odpowiednio 12 oraz 21 razy dla tych samych danych. Następnie wyniki z obu eksperymentów zostały uśrednione. Wszystkie obserwacje odstające dla tych samych wywołań eksperymentów zostały usunięte z danych[6].

5.4.2 Wyniki

Wyniki eksperymentu zostały przedstawione w pięciu tabelach. Tabela 5.1 zawiera dane obrazujące najlepsze wartości metryk AMA, AMNA, TMA, TMNA, ACA, ACNA, TCA, TCNA, ARA, ARNA, TRA, TRNA dla pierwszego zestawu plików z syntetycznym kodem oraz drugiego zestawu plików zawierającego algorytmy sortowania. Tabele 5.2 oraz 5.3 przedstawiają wartości średnie oraz odchylenia standardowe dla tych samych metryk i projektów. Tabela 5.4 przedstawia wartości metryk LOC, CycAVG, CycMAX, MN oraz NMP dla obu zestawu danych. Tabela 5.5 zawiera średni całkowity czas wykonania obu eksperymentów dla mutantów w jednej kompilacji oraz tych nie zgregowanych.

Tabela 5.1: Najlepsze wartości dla metryk AMA, AMNA, TMA, TMNA, ACA, ACNA, TCA, TCNA, ARA, ARNA, TRA, TRNA.

Plik	AMNA	AMA	TMNA	TMA	ACNA	ACA	TCNA	TCA	ARNA	ARA	TRNA	TRA
synt01	9,90	11,87	51,47	11,87	1 023,16	1 081,10	6 138,97	1 081,10	173,24	91,95	1 039,42	551,68
synt02	7,13	12,39	75,78	12,39	1 023,59	1 039,99	12 283,11	1 039,99	179,09	86,62	2 149,03	1 039,40
synt03	7,58	3,23	128,27	3,23	985,62	1 060,88	17 741,13	1 060,88	165,69	81,54	2 982,34	1 467,78
synt04	7,59	3,36	175,08	3,36	1 018,25	1 037,71	24 438,02	1 037,71	174,51	81,02	4 188,25	1 944,50
synt05	5,98	3,27	172,34	3,27	1 021,35	1 023,57	30 640,58	1 023,57	179,16	80,01	5 374,70	2 400,22
synt06	7,14	4,20	251,27	4,20	1 021,55	1 061,81	36 775,93	1 061,81	177,52	79,79	6 390,89	2 872,28
synt07	7,15	13,35	293,96	13,35	1 021,26	1 038,95	42 893,11	1 038,95	168,70	80,87	7 085,25	3 396,51
synt08	8,10	3,42	341,87	3,42	1 085,17	1 100,62	46 662,26	1 100,62	176,19	80,04	7 576,25	3 441,67
sort01	3,37	3,25	41,93	3,25	974,95	988,00	11 699,35	988,00	83,71	77,54	1 004,52	930,49
sort02	3,31	4,46	61,54	4,46	982,95	977,66	17 693,10	977,66	81,99	75,45	1 475,74	1 358,01
sort03	3,37	4,58	99,01	4,58	1 000,02	991,29	30 000,51	991,29	1 065,88	953,01	31 976,51	28 590,30
sort04	3,58	5,08	147,38	5,08	1 003,96	1 016,12	42 166,14	1 016,12	933,80	837,48	39 219,40	35 174,14
sort05	4,07	15,80	241,65	15,80	1 017,56	1 028,20	61 053,33	1 028,20	975,70	851,41	58 541,79	51 084,42

[5]Nowoczesne systemy operacyjne starają się zarządzać procesami w możliwie najbardziej efektywny sposób utrzymując proces w pamięci operacyjne tak długo jak jest to opłacalne. Jeżeli dany program statystycznie w bardzo krótkich odstępach czasu będzie wielokrotnie uruchamiany, to system operacyjny nie będzie go wywłaszczał, co znacznie skróci czas jego uruchomienia i wykonania. Nie jest to regułą, ponieważ jeżeli system będzie z jakiegoś powodu potrzebował zasobów, wywłaszczy program od razu. Innymi słowy, ten sam program uruchamiany co milisekundę może za każdym razem być wykonywany w innym czasie. Dostępność czasu procesora też będzie miała na to wpływ.

[6]Usunięto dane, których wartości czasu wykonania przekraczały wartość średnią o co najmniej dwa rzędy wielkości. Takie rozbieżności wynikają ze wspomnianego wcześniej sposobu zarządzania procesami w systemie.

Tabela 5.2: Średnie wartości dla metryk AMA, AMNA, TMA, TMNA, ACA, ACNA, TCA, TCNA, ARA, ARNA, TRA, TRNA.

Plik	AMNA	AMA	TMNA	TMA	ACNA	ACA	TCNA	TCA	ARNA	ARA	TRNA	TRA
synt01	11,08	13,24	57,90	13,24	1 091,38	1 112,90	8 776,24	1 112,90	183,77	96,34	1 620,63	593,69
synt02	8,59	14,53	92,25	14,57	1 073,61	1 090,79	17 744,97	1 090,79	184,69	90,08	2 355,57	1 118,70
synt03	8,53	5,48	147,19	4,24	1 057,10	1 106,90	19 027,84	1 106,90	185,50	83,83	3 338,92	1 508,89
synt04	8,35	5,44	192,37	4,37	1 066,58	1 079,00	29 776,52	1 079,00	189,43	82,69	4 431,78	2 106,75
synt05	6,72	5,44	193,86	4,24	1 088,03	1 089,31	35 338,10	1 089,31	187,60	82,06	5 627,88	2 579,87
synt06	8,01	6,25	282,47	5,23	1 078,39	1 151,51	38 821,99	1 151,51	186,67	84,29	6 720,11	3 160,30
synt07	7,85	14,14	322,83	13,94	1 109,96	1 115,48	46 618,28	1 115,48	193,80	85,14	7 636,23	3 575,83
synt08	8,98	3,70	380,72	3,70	1 174,62	1 271,58	50 508,87	1 271,58	182,71	81,46	8 001,73	3 619,31
sort01	5,86	4,24	61,03	4,60	1 163,40	1 255,97	12 171,93	1 255,97	287,62	124,23	2 213,10	1 490,82
sort02	4,88	10,51	73,53	11,49	1 245,99	1 179,05	18 025,27	1 179,05	223,97	123,04	4 031,47	2 214,70
sort03	5,16	10,74	127,93	10,74	1 221,59	1 207,20	30 717,79	1 207,20	1 370,68	1 156,28	35 766,92	32 092,43
sort04	5,09	7,38	187,15	7,94	1 229,53	1 235,05	43 040,79	1 235,05	1 307,32	1 215,50	42 386,53	39 066,10
sort05	5,46	18,07	263,28	18,07	1 238,76	1 289,93	62 310,62	1 289,93	1 344,83	1 073,35	62 474,80	54 666,43

Tabela 5.3: Wartości odchylenia standardowego dla metryk AMA, AMNA, TMA, TMNA, ACA, ACNA, TCA, TCNA, ARA, ARNA, TRA, TRNA.

Plik	AMNA	AMA	TMNA	TMA	ACNA	ACA	TCNA	TCA	ARNA	ARA	TRNA	TRA
synt01	0,88	1,05	5,81	1,05	34,65	26,88	2 778,25	26,88	9,30	3,03	726,19	30,80
synt02	1,39	1,42	17,90	1,59	28,66	25,73	5 910,84	25,73	7,45	4,93	320,88	106,48
synt03	1,12	3,81	23,45	0,89	35,32	32,84	635,73	32,84	19,64	1,87	353,56	33,67
synt04	0,68	3,40	15,52	1,17	27,40	42,46	7 810,00	42,46	14,60	1,56	276,31	301,17
synt05	1,12	3,66	33,17	0,74	39,95	35,49	7 219,68	35,49	9,84	1,49	295,19	291,91
synt06	0,93	3,18	32,42	0,92	29,36	79,88	1 056,93	79,88	9,66	4,07	347,78	335,39
synt07	0,65	0,84	26,52	0,76	71,02	40,76	2 982,88	40,76	25,71	5,25	478,17	220,60
synt08	1,71	0,32	79,90	0,32	121,52	201,75	5 225,29	201,75	10,10	2,30	551,70	321,14
sort01	1,33	1,01	8,57	1,77	394,38	509,08	717,97	509,08	323,64	98,73	803,85	1 184,76
sort02	1,37	4,50	5,46	5,45	471,04	419,57	347,53	419,57	127,85	97,16	2 301,39	1 748,95
sort03	1,58	6,05	17,20	6,05	479,16	439,16	700,47	439,16	490,76	316,02	5 108,59	5 010,48
sort04	2,11	2,88	51,52	3,71	458,52	436,86	908,14	436,86	427,32	510,06	4 439,28	5 936,53
sort05	2,20	2,29	24,82	2,29	409,74	498,16	1 360,76	498,16	393,47	289,34	5 415,18	5 679,08

Tabela 5.4: Metryki LOC, CycAVG, CycMAX, MN oraz NMP dla kodu syntetycznego oraz biblioteki sortującej.

Plik	Loc	CycAVG	CycMAX	MN	NMP
synt01	10	2	2	6	1
synt02	13	3	3	12	2
synt03	16	4	4	18	3
synt04	19	5	5	24	4
synt05	22	6	6	30	5
synt06	26	8	8	36	6
synt07	35	5	8	42	7
synt08	39	5,5	8	43	8
sort01	46	2,25	5	12	2
sort02	72	2,60	5	18	3
sort03	89	2,86	5	30	5
sort04	152	3,11	6	42	7
sort05	208	3,17	6	60	10

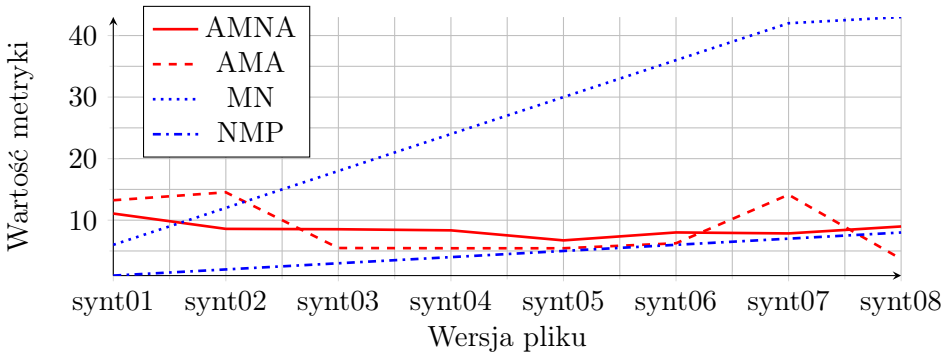
Tabela 5.5: Uśredniony całkowity czas wykonania mutacji, kompilacji oraz uruchomienia programów z agregacją mutantów w jednej kompilacji oraz bez agregacji dla kodu syntetycznego oraz biblioteki sortującej.

Plik	Wiele kompilacji	Jedna kompilacja
synt01	11 740,99	2 942,29
synt02	21 459,69	3 419,46
synt03	23 765,08	3 816,25
synt04	35 665,02	4 357,26
synt05	42 442,18	4 850,21
synt06	47 097,63	5 559,08
synt07	55 888,95	5 920,01
synt08	60 257,64	6 251,33
sort01	15 902,95	4 135,83
sort02	23 605,12	4 717,83
sort03	69 210,07	35 684,58
sort04	88 156,40	42 767,01
sort05	127 637,74	58 355,78

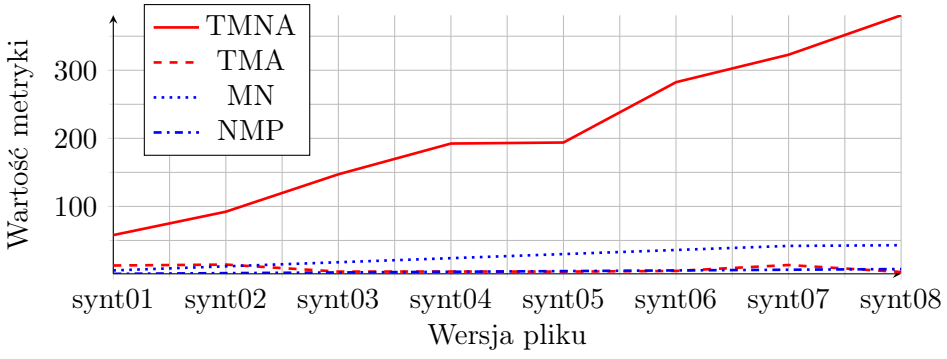
Dane w tabelach pokazują, że technika generacji wielu mutantów w jednej kompilacji ma

znaczącą przewagę nad tradycyjnymi metodami opartymi o wielokrotną kompilację plików źródłowych dla każdego mutanta. Wartość średnia złożoności czasowej generacji mutantów w przypadku agregacji kompilacji jest nieznacznie większa niż dla przypadku niezagregowanego. Podczas uruchamiania zmutowanych programów różnice między oboma podejściami są już zauważalne. Wykonywanie tego samego programu z rożnymi parametrami wywołań jest zdecydowanie szybsze, aniżeli uruchamianie kilkunastu zmutowanych wersji programu. Ta różnica wynika prawdopodobnie w głównej mierze z architektury zarządzania pamięcią operacyjną w systemie operacyjnym. W systemie Windows 10 na którym był przeprowadzany eksperyment program po wczytaniu do pamięci i uruchomieniu zostaje tam przez pewien czas, o ile jest to możliwe. To sprawia, że przy następnym odwołaniu się do danego programu w krótkim czasie od jego poprzedniego uruchomienia nie jest on ponownie ładowany z dysku.

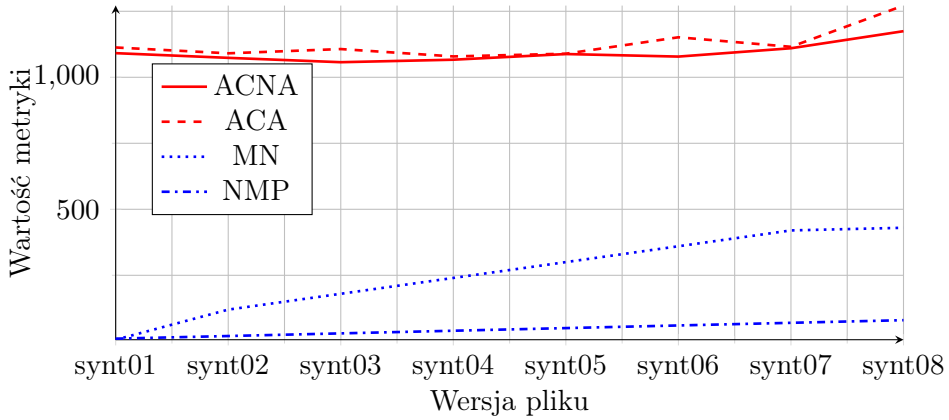
Największe różnice można zaobserwować w przypadku procesu kompilacji. Technika agregacji wielu mutantów w jednej kompilacji jest znacząco lepsza. W dodatku różnica zwiększa się wraz z wzrostem liczby mutantów. Na rysunkach 5.4-5.9 znajdują się wykresy obrazujące dynamikę zależności miedzy poszczególnymi metrykami. Wykres 5.4 obrazuje zależność między AMA, AMNA, MN oraz NMP. Wykres 5.5 obrazuje zależność między TMA, TMNA, MN oraz NMP. Wykres 5.6 obrazuje zależność między ACA, ACNA, MN oraz NMP. Wykres 5.7 obrazuje zależność między TCA, TCNA, MN oraz NMP. Wykres 5.8 obrazuje zależność między ARA, ARNA, MN oraz NMP. Wykres 5.9 obrazuje zależność między TRA, TRNA, MN oraz NMP.



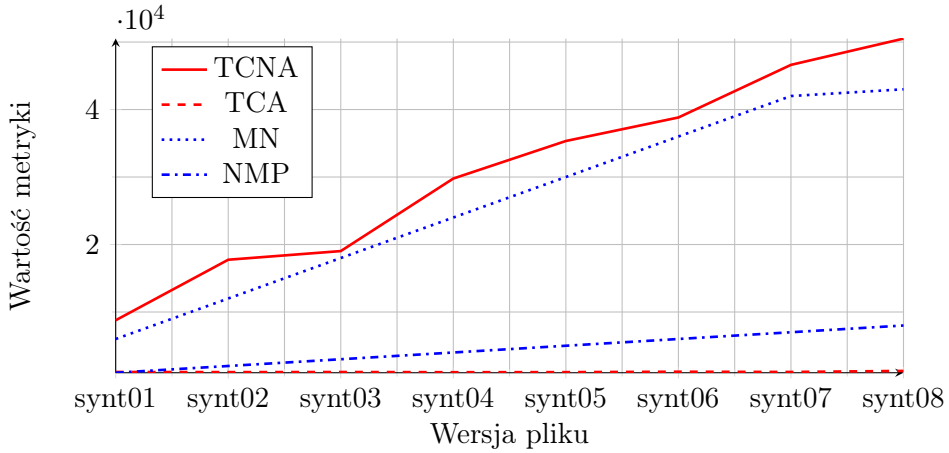
Rysunek 5.4: Stosunek wartości średnich metryk AMA oraz AMNA do MN oraz NMP.



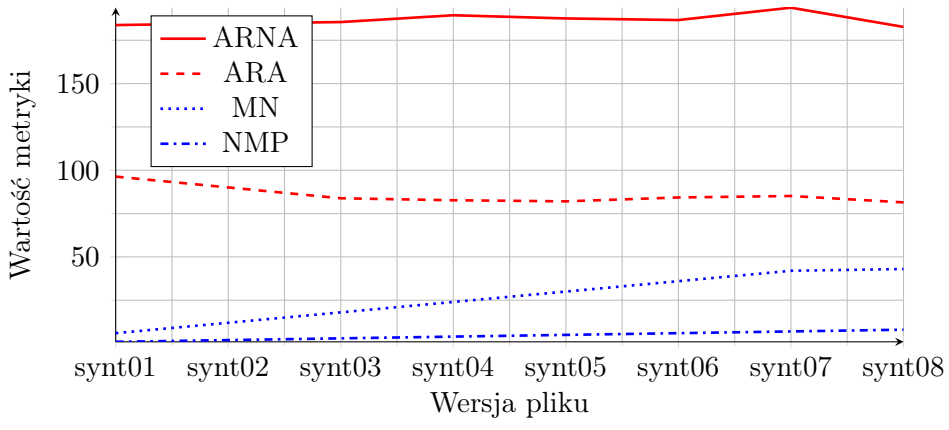
Rysunek 5.5: Stosunek wartości średnich metryk TMA oraz TMNA do MN oraz NMP.



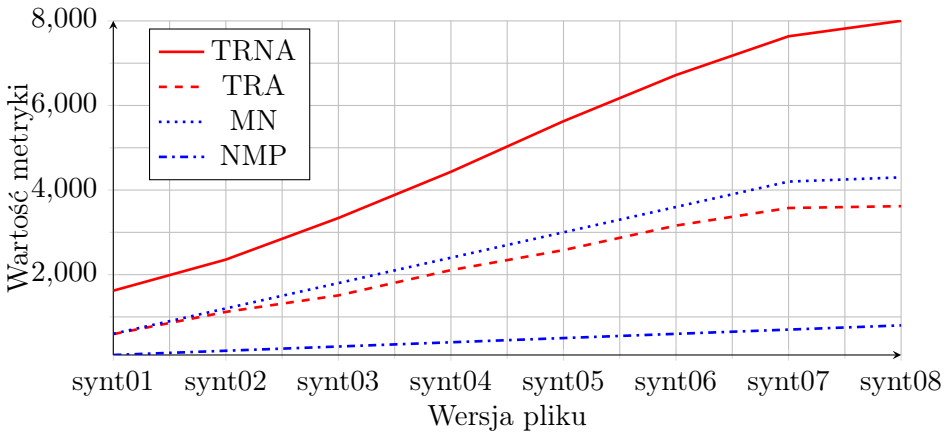
Rysunek 5.6: Stosunek wartości średnich metryk ACA oraz ACNA do MN oraz NMP (w celu lepszego zobrazowania zależności MN i NMP zostały przeskalowane x 10).



Rysunek 5.7: Stosunek wartości średnich metryk TCA oraz TCNA do MN oraz NMP (w celu lepszego zobrazowania zależności MN i NMP zostały przeskalowane x 1000).



Rysunek 5.8: Stosunek wartości średnich metryk ARA oraz ARNA do MN oraz NMP.



Rysunek 5.9: Stosunek wartości średnich metryk TMA oraz TMNA do MN oraz NMP (w celu lepszego zobrazowania zależności MN i NMP zostały przeskalowane x 100).

Analogiczne zależności można zaobserwować w danych wygenerowanych na podstawie plików sort01 do sort05.

5.5 Wnioski i podsumowanie

Proces generacji wielu mutantów w jednej kompilacji wykazuje znaczną efektywność jeśli chodzi o czas wykonania programu. Średni czas generacji mutantów jest porównywalny do tego jaki jest uzyskiwany w metodach tradycyjnych. Praktycznie dla wszystkich wskaźników czasowych agregacja mutacji w jedną kompilację ma przewagę nad generowaniem wielu kompilacji po jednej dla każdego mutanta. Stosowanie oraz rozwijanie tej technologii może znacząco przyspieszyć proces testowania mutacyjnego w takich językach jak C, C++ oraz innych, dla których proces kompilacji skutkuje generacją kodu wykonywalnego (object code). Według naszej najlepszej wiedzy taki model nie został jeszcze opracowany. Analizując możliwości oprogramowania do testów mutacyjnych MAJOR [40] oraz publikacje z nim związane widać, że podobny model mógł zostać zaimplementowany w tym oprogramowaniu. Niestety, w dokumentacji i publikacjach ciężko trafić na jego opis. Oczywiście proces ten nie ma zastosowania w językach skryptowych, w których obsługą kodu zajmuje się interpreter, a także w językach, w których kompilacja generuje kod pośredni uruchamiany na wirtualnej maszynie danego języka.

5.6 Identyfikacja zagrożeń dla poprawności badań

Głównym zagrożeniem dla poprawności wyników badań są czynniki związane z architekturą systemu oraz zarządzaniem czasem pracy procesora. Ten sam program lub komenda w dwóch różnych odstępach czasowych może dawać całkowicie inny czas wykonania. Mechanizmem chroniącym przed tym zagrożeniem było wielokrotne wykonywanie tego samego eksperymentu i uśrednianie wyników. Kolejnym zagrożeniem jest niedoskonałość platformy mutującej – jest to wersja prototypowa we wczesnej fazie wytwarzania. Innym zagrożeniem jest fakt, iż prototyp obejmuje tylko implementację niektórych operatorów mutacyjnych – nie wiadomo, czy wyniki przenoszą się na przypadek innych operatorów mutacyjnych, w tym operatorów obiektowych.

Rozdział 6

Mutation Churn Model

W tym rozdziale zaproponowany i opisany został model optymalizacji procesu testowania mutacyjnego przewidujący stosowanie analizy mutacyjnej tylko i wyłącznie na zmienionym lub nowym kodzie między wersjami danego oprogramowania. Rozdział zawiera opis założeń modelu optymalizacyjnego, jego praktyczne zastosowanie w eksperymentach oraz podsumowanie uzyskanych wyników potwierdzających poprawność działania modelu.

Treść rozdziału została napisana na podstawie publikacji [57] będącej pracą wspólną Adama Romana, Michała Mnicha oraz Piotra Wawrzyniaka. Adam Roman był pomysłodawcą badania i opracował je koncepcyjnie. Piotr Wawrzyniak odpowiedzialny był za wykonanie eksperymentów, zebranie danych oraz ich analizę na oprogramowaniu EasyMock. Autor niniejszej rozprawy odpowiedzialny był za redakcję tekstu, wykonanie eksperymentów, zebranie danych oraz ich analizę na oprogramowaniu JUnit. Jest również samodzielnym autorem podrozdz. 6.2. Informacje zawarte w publikacji zostały rozwinięte oraz dopracowane w treści tego rozdziału.

6.1 Założenia problemu

W obecnych czasach wytwarzanie oprogramowania jest procesem ciągłym, w którym kolejne wersje oprogramowania mogą być wypuszczane w stosunkowo bliskich odstępach czasowych. Co tydzień, co dzień lub nawet po każdej wprowadzonej zmianie w kodzie[1]. Taka częstotliwość wprowadzania zmian przy większych projektach powoduje, że analiza mutacyjna nie może być stosowana przed wypuszczeniem każdej nowej wersji oprogramowania, ze względu na potrzebną do jej wykonania dużą ilość zasobów obliczeniowych. Kompletna analiza mutacyjna znacznie zmniejszałaby bowiem wydajność procesu wytwarzania oprogramowania. Czasem zasoby, jakie musiałyby zostać użyte do przetestowania mutacyjnego oprogramowania po zmianie w kodze byłyby większe niż te wymagane do wprowadzenia samej zmiany. Wobec tego pojawia się istotne pytanie: czy stosować analizę mutacyjną w pełni i opóźniać wydawanie kolejnych wersji produktu, czy też przeprowadzać testowanie mutacyjne tylko na nowym lub zmodyfikowanym fragmencie oprogramowania, ryzykując ewentualne niewykrycie problemów regresji (tzn. sytuacji, w której zmiana w kodzie spowodowała problem w innym miejscu w systemie). Rozpatrzenie tej

[1] Aktualizacją po każdej zmianie w kodzie cechują się głównie serwisy internetowe oraz inne aplikacje sieciowe stosujące takie podejścia jak Continuous Integration czy Continuous Delivery.

kwestii może mieć zasadnicze znaczenie przy podnoszeniu wydajności procesów deweloperskich korzystających z technik testowania mutacyjnego.

6.2 Problematyczne przykłady

Początkowo można sądzić, że skoro testowanie mutacyjne zostało wykonane wcześniej na jakimś fragmencie oprogramowania, to wykonywanie go ponownie na niezmienionym fragmencie kodu nie ma żadnego znaczenia. Niestety, poniższy prosty, choć hipotetyczny przykład pokazuje nam, że to założenie nie zawsze musi być prawdziwe. W przykładzie z listingu 6.1 mamy dwie wersje kodu: wersję starą (N) oraz wersję nową ($N+1$), z niewielką zmianą. Pokażemy, że istnieje taki przypadek, w którym nawet trywialna zmiana w jednej linii kodu może wpływać na przeżywalność mutantów w innych częściach oprogramowania.

Listing 6.1: Wpływ zmian w kodzie na inne fragmenty

```
function f(int x) {     // wersja N
  x := x - 1
  if (x >= 1 || x <= -1) // fragment kodu bez zmian
    return 1
  else
    return 0
}

function f(int x) {     // mutant wersji N
  x := x - 1
  if (x > 1 || x <= -1) // mutant: zmiana >= na >
    return 1
  else
    return 0
}

function f(int x) {      // wersja N+1
  x := 1 - x             // zmieniony fragment kodu
  if (x >= 1 || x <= -1) // fragment kodu bez zmian
    return 1
  else
    return 0
}
```

Załóżmy, że mamy test, w którym $x = 2$. W wersji N programu wywoływana jest funkcja $f(2)$, która zwróci wartość 1. Następnie, dla tej funkcji został wygenerowany mutant dotyczący operatora relacyjnego w instrukcji warunkowej `if` zmieniający $\geq$ na $>$. W efekcie, ponawiając test dla $x = 2$ uzyskamy od funkcji wartość 0. W tym wypadku mutant został wykryty. W wersji programu $N + 1$ wprowadzono zmianę w pierwszej instrukcji: `x:=1-x`. Jeżeli użyjemy teraz tego samego przypadku testowego co w wersji N, to mutant nie zostanie wykryty.

Przypadek opisany powyżej jest dosyć trywialny, ale obrazuje możliwą do zaistnienia sytuację w

kodzie. Przykładowo, gdy w naszym oprogramowaniu zostanie użyty wzorzec fabryki[2] i zmiany w kolejnej wersji zostaną wprowadzone jedynie w jednym z produktów fabryki, na którego podstawie jakiś fragment oprogramowania generuje swój przepływ, to testując mutacyjnie tylko zmieniony fragment kodu, zostaną pominięte zmiany w stanie oprogramowania wynikające z tej zmiany. Co za tym idzie, wartości przechowywane w obiektach w kolejnej sesji mutacyjnej mogą być inne niż w poprzednich. Taka sytuacja może spowodować nagłe pojawienie się żywych mutantów w niezmienionym kodzie.

Fundamentalnym pozostaje jednak następujące pytanie: jak często taka sytuacja może mieć miejsce? Jeżeli prawdopodobieństwo jej wystąpienia jest relatywnie małe, być może sensownym jest stosować tylko częściową analizę mutacyjną a całościowo mutować projekt tylko przy znaczących zmianach oprogramowania? W poniższym podrozdziale opisano eksperyment, który miał na celu empiryczną odpowiedź na to pytanie.

6.3 Eksperyment

W eksperymencie została użyta zmodyfikowana wersja systemu PIT 1.1.7 [44], Jest to oprogramowanie służące do przeprowadzania analizy mutacyjnej w oprogramowaniu opartym o język Java. PIT posiada kilka mechanizmów optymalizacyjnych pozwalających na szybsze przeprowadzanie sesji testowej. Jeden z takich mechanizmów pozwala nie uruchamiać kolejnych testów dla mutanta, który został zabity przez bieżący test. Na potrzeby eksperymentu ten mechanizm został wyłączony, wobec czego uzyskane z eksperymentu dane zawierały pełną informację odnośnie każdego testu uruchomionego na każdym mutancie. Pozwoliło to uzyskać informację, jak wiele testów wykryło daną mutację.

Przebieg eksperymentu był następujący. Dla wybranego programu przeprowadzono mutacje jego poszczególnych wersji. Proces ten został wykonany w celu symulowania kilku iteracji wytwarzania oprogramowania. Następnie została wykonana analiza danych z raportów uzyskanych podczas procesu mutacji i testowania kodu. Raporty były analizowane pod kątem znajdowania takich mutantów, które w jednej wersji były wykrywane, a w kolejnej już nie lub na odwrót. Ponadto, na podstawie raportów został opisany przebieg życia mutantów pomiędzy wersjami danego programu. Ostatnim krokiem eksperymentu była analiza danych o przebiegu życiu mutantów, która miała potwierdzić bądź zaprzeczyć hipotezie mówiącej, że można skutecznie optymalizować proces testowania mutacyjnego, stosując tylko mutacje na zmienionych fragmentach kodu i zarazem nie obniżając skuteczności testowania mutacyjnego.

Opierając się na raportach wygenerowanych przez program PIT, zostało utworzone unikatowe nazewnictwo mutantów bazujące na nazwach metod/testów i ścieżek klas (ang. classpath) powiązanych z danym mutantem. Używając tego nazewnictwa, ręcznie przeanalizowano zmiany w testach oraz metodach w poszczególnych wersjach oprogramowania. Jako wynik uzyskano zestaw metod/testów, które uległy usunięciu, modyfikacji bądź zostały dodane.

[2]Kreacyjny wzorzec projektowy, którego celem jest dostarczenie funkcjonalności polegającej na tworzeniu różnych obiektów jednego typu lub implementujących ten sam interfejs bądź klasę bazową. W tym wzorcu do tworzenia produktów fabryki oddelegowuje się jeden obiekt bądź klasę statyczną. Typ produktów fabryki jest określany podczas działania programu poprzez odpowiednie żądanie wytworzenia obiektu.

Ręczna analiza była konieczna ze względu na to, że część mutantów wyglądała tak samo, lecz kod, na bazie którego mutant powstał, był zmieniony. W oprogramowaniu nastąpiło również wiele zmian w testach. Niektóre były istotne z punktu widzenia logiki testu, lecz cześć polegała tylko na zmianach kosmetycznych lub dodaniu komentarzy. Także część mutantów z wersji na wersję zmieniało swój stan z wykrytego na nie wykryty.

W następnym kroku eksperymentu zostały porównane mutanty z różnych wersji oprogramowania o tej samej sygnaturze zgodnej z naszym nazewnictwem. Tym sposobem zostały zidentyfikowane mutanty nowe, usunięte oraz zmodyfikowane. Na sam koniec została przeprowadzona analiza przyczyny zmiany stanu mutantów pomiędzy wersjami.

Godnym wspomnienia jest fakt, że eksperyment był przeprowadzany na oprogramowaniu, którego proces wytwórczy *nie korzystał* z testowania mutacyjnego. Z tego powodu można było zdecydować które mutanty są istotne z punktu widzenia badania. Jeżeli testowanie mutacyjne zostałoby użyte, to modyfikacje testów mogłyby wpływać na wyniki eksperymentu.

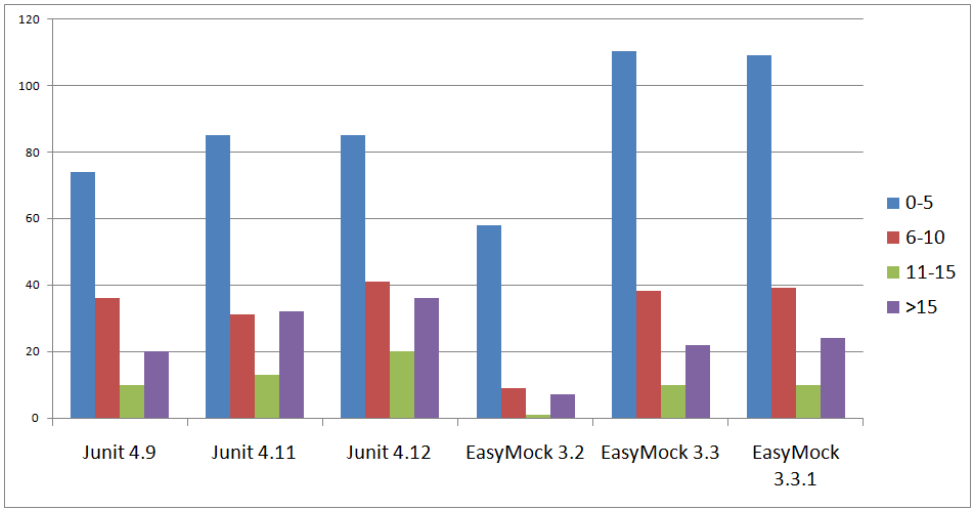
Oprogramowanie, które podlegało analizie to: JUnit w wersjach: 4.9, 4.11[3], 4.12 oraz EasyMock w wersjach: 3.2, 3.3, 3.3.1. Zarówno JUnit, jak i EasyMock są oprogramowaniem typu open sorce. Wszystkie projekty były pobrane z oficjalnych repozytoriów umieszczonych w serwisie GitHub. Tabela 6.1 reprezentuje podstawową charakterystykę badanych programów.

Tabela 6.1: Podstawowa charakterystyka testowanych aplikacji

Program	Wersja	LOC	liczba metod	liczba testów
JUnit	4.9	15440	106	409
JUnit	4.11	23794	111	506
JUnit	4.12	26079	135	727
EasyMock	3.2	12659	213	747
EasyMock	3.3	13158	233	764
EasyMock	3.3.1	13240	237	765

[3]Wersja 4.10 była wersją beta od 4.11, dlatego została pomięta.

Rys. 6.1 prezentuje rozkład liczby metod w klasach.



Rysunek 6.1: Rozkład liczby metod w klasach dla poszczególnych aplikacji

W tabeli 6.2 pokazane są wartości standardowych metryk kodu (tzw. metryk CK) dla każdego testowanego projektu. Metryki zostały opisane i zaproponowane przez Chidambera i Kemerera [58], a ich krótki opis znajduje się poniżej.

- CC (Cyclomatic Complexity) — złożoność cyklomatyczna kodu. Określona jest poprzez liczbę punktów decyzyjnych w oprogramowaniu + 1. Im wartość ta jest niższa, tym prostszą strukturę (w sensie logiki kodu) posiada program.

- WMC (Weighted Methods per Class). Definiowana jest przez liczbę metod zadeklarowanych w klasie. Im wartość WMC jest niższa, tym lepiej, ponieważ powszechnie wiadomo, że wysoka WMC prowadzi zwykle do większej liczby usterek. Ta cecha została wykazana między innymi w pracy Laing, Victor, Coleman, Charles [59].

- DIT (Depth of Inheritance Tree) — maksymalna długość ścieżki dziedziczenia od klasy do klasy głównej. DIT mierzy głębokość hierarchii klas. Im głębsze drzewo, tym więcej metod i zmiennych może dziedziczyć, co sprawia, że kod jest bardziej złożony, a więc bardziej podatny na defekty. Wartość DIT powinna być zatem możliwie mała.

- NOC (Number of Children) — liczba bezpośrednich podklas danej klasy. NOC mierzy szerokość hierarchii klas. Wysoka wartość NOC może wskazywać na: 1) częste ponowne użycia klasy bazowej, 2) to, że klasa bazowa może wymagać więcej testów, 3) niewłaściwe abstrakcje klasy macierzystej oraz 4) niewłaściwe wykorzystanie podklasy.

- LCOM (Lack of Cohesion of Methods) — brak kohezji metod. Wartość ta obliczana jest w następujący sposób[4]: zmienne P i Q ustawia się na 0 i rozważa się każdą parę metod w klasie. Jeśli uzyskują one dostęp do rozłącznych zbiorów zmiennych instancji, zwiększa

[4]W literaturze przedmiotu występuje kilka definicji tej metryki, najbardziej popularne oznaczane są jako LCOM1, LCOM2, LCOM3 i LCOM4. W niniejszym eksperymencie wykorzystano metrykę LCOM1.

się wartość P o jeden. Jeśli mają co najmniej jeden wspólny dostęp do zmiennej, zwiększa się Q o jeden. Wartość metryki definiuje się jako tzw. nieujemną różnicę tych wartości: $LCOM = P - Q$, jeśli $P > Q$ oraz $LCOM = 0$ jeśli $P \leq Q$. Jeśli $LCOM = 0$, klasa jest spójna. W przeciwnym razie może wymagać podziału na dwie lub więcej klas, ponieważ jej zmienne należą do zbiorów rozłącznych. Klasy z wysokim LCOM są uznawane za bardziej podatne na defekty.

Tabela 6.2: Podstawowe metryki analizowanych programów

		CC	Metryki CK			
Program	Wersja	$\mu \pm \sigma$	WMC	DIT	NOC	LCOM
JUnit	4.9	1.22	4.55	1.50	0.42	0.04
		±0.72	±8.1	±0.9	±3.27	±0.15
JUnit	4.11	1.22	4.59	1.45	0.38	0.04
		±0.67	±8.75	±0.85	±3.21	±0.16
JUnit	4.12	1.17	4.27	1.40	0.34	0.04
		±0.48	±0.19	±0.80	±2.94	±0.16
EasyMock	3.2	1.46	9.55	1.21	0.12	0.09
		±1.25	±15.37	±0.56	±0.71	±0.21
EasyMock	3.3	1.46	9.01	1.22	0.14	0.09
		±1.23	±15.11	±0.56	±1.1	±1.21
EasyMock	3.3.1	1.46	8.89	1.24	0.14	0.09
		±1.13	±15.01	±0.58	±0.92	±1.21

Dane w tabeli 6.2 pokazują, że kod programu JUnit ma tendencję do niskiego poziomu złożoności kodu. Pomimo wzrostu liczby klas średnia wartość CC spada z jednej wersji na drugą. Można także zaobserwować spadek odchylenia standardowego dla wartości CC. Tego typu zjawisko sugeruje, że kod był refaktoryzowany. W przypadku EasyMock średnia wartość złożoności cyklomatycznej utrzymuje się na podobnym poziomie. Natomiast odchylenie standardowe lekko spada. Te dane także sugerują, że kod programu EasyMock również podlegał refaktoryzacji.

W przypadku metryki WMC odchylenie standardowe jest wysokie w porównaniu ze średnią wartością. Oznacza to, że liczba metod różni się znacznie w zależności od klasy, zarówno w kodzie programu JUnit, jak i EasyMock. DIT, w obu aplikacjach i we wszystkich wersjach, pozostaje na niskim poziomie, pomiędzy 1 i 2. Pokazuje to, że hierarchia klas jest płaska we wszystkich analizowanych wersjach programów.

Tabela 6.3: Zmiany w testach jednostkowych przez wersje

			Testy	
Z wersji na wersję	dodane	usunięte	zmienione	niezmienione
JUnit z 4.9 na 4.11	118	21	30	358
JUnit z 4.11 na 4.12	285	64	41	401
EasyMock z 3.2 na 3.3	19	3	14	730
EasyMock z 3.3 na 3.3.1	1	0	0	764

Tabela 6.3 pokazuje statystyki dotyczące zmiany liczby testów z jednej wersji na drugą. Z

danych tych wynika, że zmiany z wersji 4.11 na 4.12 były o wiele większe, niż w przypadku zmiany z wersji 4.9 na 4.11. Z kolei zmiany w kolejnych wersjach EasyMock nie są zbyt duże. W przypadku wszystkich zmian wersji widać, że liczba niezmienionych testów jest bardzo duża.

Tabela 6.4: Ogólne statystyki analizy mutacyjnej

		Mutanty		
Aplikacja	liczba	wykryte	niewykryte	LOC/mutanta
JUnit 4.9	1673	954 (57%)	719 (43 %)	9.23
JUnit 4.11	1903	1091 (57%)	812 (43 %)	12.50
JUnit 4.12	2351	1494 (64%)	857 (36 %)	11.09
EasyMock 3.2	1374	1148 (84%)	226 (16 %)	9.21
EasyMock 3.3	1424	1196 (84%)	228 (16 %)	9.24
EasyMock 3.3.1	1424	1196 (84%)	228 (16 %)	9.30

Tabela 6.4 przedstawia ogólne statystyki analizy mutacyjnej dla wszystkich analizowanych wersji aplikacji. W tabeli jest pokazana całkowita liczba wygenerowanych mutantów dla każdej wersji programu, liczba wykrytych mutantów oraz tych niewykrytych. Ostatnia kolumna określa stosunek metryki LOC do liczby mutantów. Ten wskaźnik mówi, ile wynosi średnio liczba linii kodu na jednego mutanta. Jeżeli mutanty są generowane przez to samo narzędzie w taki sam sposób dla wszystkich wersji danej aplikacji, natychmiastowe zwiększenie tej metryki może sugerować istotną zmianę w strukturze kodu (na przykład, gdy do kodu zostanie dodany fragment, który nie ulega mutacji w odniesieniu do danego narzędzia mutacyjnego i operatorów mutacji).

Znaczący wzrost stosunku LOC do mutantów z JUnit 4.9 do 4.11 wynika z bardzo dużej zmiany kodu. Po pierwsze, rozmiar kodu znacznie wzrósł z 15 KLOC do prawie 24 KLOC. Po drugie, większość dodanego kodu to kod interfejsów oraz inne elementy, które nie podlegają mutacji.

Tabela 6.5: Szczegółowa analiza mutacji według typu operatora mutacji

Oper. →	CBM		IM		VMCM		RVM		MM		NCM	
Aplikacja	G	K	G	K	G	K	G	K	G	K	G	K
JU 4.9	59	25	40	19	475	238	587	366	53	12	459	297
JU 4.11	67	27	45	21	574	287	634	399	58	19	525	338
JU 4.12	78	47	60	39	586	291	808	549	61	23	680	488
EM 3.2	65	57	40	35	318	276	479	374	26	22	446	384
EM 3.3	64	56	39	34	332	289	503	397	29	25	457	395
EM 3.3.1	64	56	39	34	332	289	503	397	29	25	457	395

G = wygenerowany, K = zabity, JU = JUnit, EM = EasyMock

Tabela 6.5 przedstawia szczegółowe statystyki dotyczące współczynnika mutacji w stosunku do typu operatora mutacyjnego. W tym eksperymencie wykorzystano 6 różnych typów operatorów mutacyjnych dostępnych w narzędziu PIT[5]:

[5]PIT wprowadził własną nomenklaturę operatorów mutacyjnych. Pokrywa się ona w większości przypadków z tą zaproponowaną przez Offutta.

- CBM – Conditionals Boundary Mutator. Zastępuje operatory relacyjne $<$, $\leq$, $>$, $\geq$ dopełnieniami ich wartości granicznych (odp. $\leq$, $<$, $\geq$, $>$).
- IM – Increments Mutator. Mutuje inkrementacje i dekrementacje. Zastępuje przyrosty spadkami i na odwrót, na przykład: `i++` na `i--`.
- VMCM – Void Method Call Mutator. Usuwa wywołania metod typu `void`.
- RVM – Return Values Mutator. Mutuje zwracane wartości wywołań metod. W zależności od typu metody stosuje się inną mutację. Na przykład, dla typu boolean zastępuje on wartość `true` wartością `false`.
- MM – Math Mutator. Zastępuje operatory arytmetyczne innymi operatorami arytmetycznymi. Na przykład: `+` na `-`, `*` na `/`, `&` na `|` itd. W nomenklaturze stosowanej w pracy [14] oraz w rozdziale 3 ten operator nosi nazwę AOR.
- NCM – Negate Conditionals Mutator. Zamienia wartości warunkowe na ich negacje, na przykład: `==` na `!=`, `<=` na `>` oraz `>=` na `<`.

Tabela 6.6: Różne rodzaje typów sukcesji mutantów

Typ sukcesji	JUnit	EasyMock
-K	45	16
-S	3	2
KS	2	0
SK	2	0

W tabeli 6.6 przedstawiono analizę ilościową różnych typów przebiegu cyklu życia mutantów pomiędzy wersjami oprogramowania. Ten proces został nazwany *sukcesją*. Każda wartość w tabeli reprezentuje sumę liczby sukcesji danego typu od wersji pierwszej do drugiej i analogicznej liczby od wersji drugiej do trzeciej dla obu aplikacji. Rozważa się cztery interesujące rodzaje sukcesji:

- -K: W wersji N ten mutant nie występował. W wersji $N+1$ mutant pojawił się i został wykryty.
- -S: W wersji N ten mutant nie występował. W wersji $N+1$ mutant pojawił się i nie został wykryty.
- KS: W wersji N ten mutant był wykryty, ale w wersji $N+1$ nie został wykryty.
- SK: W wersji N ten mutant był nie wykryty, ale w wersji $N+1$ został wykryty.

Litery K i S pochodzą od angielskich słów Killed (mutant został zabity) oraz Survived (mutant przeżył, nie został wykryty). Istnieją również dwa inne rodzaje sukcesji: KK i SS, ale nie są one interesujące z punktu widzenia badań. KK oznacza po prostu, że mutant został wykryty w

obu wersjach, natomiast SS — że mutant nie został wykryty w obu wersjach. Niezerowa liczba sukcesji SS oznacza, że analiza mutacji nie była używana w przypadku JUnit i EasyMock. Dla projektu JUnit liczba SS wyniosła 44, (20.8% wszystkich sukcesji) a liczba KK – 115 (54.5% wszystkich sukcesji). Natomiast dla projektu EasyMock liczba SS wyniosła 9, (0.9% wszystkich sukcesji) a liczba KK – 74 (73% wszystkich sukcesji). Wobec tego obie aplikacje są dobrymi kandydatami do badań. Zastosowanie analizy mutacyjnej w wyżej wspomnianych programach spowodowałoby poprawę zestawu testów, a tym samym zniekształciłoby wyniki eksperymentu.

Typy -K i -S przedstawiają sytuacje, w których nowe mutanty zostały wprowadzone ze względu na zmianę kodu. Nowe mutanty można wprowadzić tylko wtedy, gdy dana część kodu została zmieniona lub gdy została napisana nowa część kodu. Interesujące jest, że w przypadku JUnit było w sumie 48 nowych mutantów, a nieznacznie zmodyfikowany zestaw testów[6] był w stanie zabić większość z nich (45, czyli około 93%). Również w przypadku EasyMock prawie wszystkie nowe mutanty zostały zabite (16 z 18, czyli około 89%).

Najciekawsze przypadki związane są z sukcesjami KS i SK. KS oznacza, że wcześniej wykryty mutant nie został wykryty w następnej wersji. Jest to niepożądana sytuacja, jeśli chcemy się skupić wyłącznie na analizie mutacji dla zmienionej części kodu. W przypadku EasyMock żadne mutanty nie należą do kategorii KS. W przypadku JUnit zaobserwowano dwie takie sytuacje. Liczba tego typu niepożądanych sukcesji jest raczej pomijalną wartością w odniesieniu do całkowitej liczby ponad 3500 mutantów wygenerowanych w wersjach 4.9 i 4.11. Można stwierdzić, że (niepożądana) sukcesja KS występuje incydentalnie.

Powyższe dwa przypadki sukcesji KS zostały dokładnie przeanalizowane. W JUnit od 4.11 do 4.12 nastąpiła sukcesja KS, gdy zmieniono metodę stosowaną przez zmutowaną metodę. Test i główna metoda pozostały niezmienione. Ten przypadek odpowiada hipotetycznym przykładom opisanym w podrozdz. 6.2. Drugi przykład KS zaobserwowano w zmianie JUnit z 4.9 do 4.11. Nastąpiło to poprzez przeniesienie ciała metody do innej metody, z niewielkimi modyfikacjami i bez aktualizacji testowych.

Typ sukcesji SK oznacza, że mutant, który przeżył w wersji starej, został zabity w następnej. Jest to pożądane (choć bardzo rzadkie, gdy nie są stosowane testy mutacyjne w celu ulepszenia danego zestawu testów), ponieważ uzyskuje się w ten sposób więcej zabitych mutantów zerowym kosztem. W przypadku JUnit były to dwa takie przypadki. W przypadku EasyMock nie zaobserwowano sukcesji SK. W JUnit wszystkie sukcesje SK były obserwowane tylko po dodaniu nowego testu.

6.4 Wnioski

W niniejszym rozdziale zbadano skuteczność analizy mutacji dla dynamicznie zmieniającego się kodu. Model ten dobrze pasuje do projektów zwinnych i open-source, a także do projektów wykorzystujących ciągłą integrację. Przeprowadzając analizę mutacyjną w kilku kolejnych wersjach

[6]W obu projektach między wersjami zdarzały się przypadki niewielkiej liczby nowych testów, a mimo wszystko testy pozostawały dalej skuteczne.

dwóch aplikacji: JUnit oraz EasyMock, policzono liczbę tak zwanych typów sukcesji wyników analizy mutacji.

Niska liczba mutantów typu KS sugeruje, że wykonanie analizy mutacji na niezmienionej części kodu nie jest skuteczne, ponieważ mutant wykryty w poprzedniej wersji z bardzo wysokim prawdopodobieństwem zostanie wykryty w wersji następnej. Dlatego podczas testowania nowej wersji można ograniczyć analizę mutacyjną tylko do zmienionej lub dodanej części kodu źródłowego. Takie podejście pozwala znacząco obniżyć koszty analizy mutacji przy zachowaniu niemal niezmienionego poziomu ryzyka związanego z testowaniem mutacji.

Wyniki powyższych badań sugerują, że rozsądne może być wykorzystanie wyżej wspomnianego podejścia w aplikacjach takich jak PIT. Jeśli aplikacja do testowania mutacji jest w stanie przeanalizować różnice między dwiema wersjami kodu źródłowego, może uruchomić analizę tylko na zmienionej lub dodanej części kodu źródłowego, skracając czas tej analizy. W takim przypadku prawdopodobieństwo, że pominięta zostanie generacja mutanta, który nie może być wykryty przez dany zestaw testów, jest bardzo niskie.

6.5 Identyfikacja zagrożeń dla poprawności badań

Głównym zagrożeniem dla poprawności wyników badań, które zostały opisane w tym rozdziale, jest mała liczba zbadanych programów (przeanalizowane zostały tylko dwa projekty: JUnit i EasyMock). Liczba przeanalizowanych wersji też była nieznaczna (tylko trzy na każdą aplikację; ostania wersja EasyMock tylko nieznacznie różniła się od poprzedniej pod względem liczby linii kodu). Niestety, przeprowadzanie eksperymentów na większą skalę było niemożliwe ze względu na ograniczone zasoby ludzkie potrzebne do ich przeprowadzenia — badania wymagały bowiem ręcznej, żmudnej analizy i nie było możliwe ich zautomatyzowanie.

Ponadto, badanie opierało się na „głównych” wersjach oprogramowania, które w procesie wytwórczym niewątpliwie podlegały wielu drobnym zmianom, których z oczywistych względów nie można było ująć w eksperymencie. Być może dla tych specyficznych, pojedynczych zmian w kodzie interesujące nas typy sukcesji mutantów pojawiały się częściej, ale w procesie testowania i usuwania defektów, w „głównych” wersjach kodu nie miały już miejsca.

Rozdział 7

Podejście bayesowskie do priorytetyzacji mutantów

W każdym projekcie prowadzonym w metodyce z zastosowaniem testowania mutacyjnego możemy zaobserwować propagację mutantów trywialnych (patrz def. 3.2) względem projektu. Proces ten polega na tym, że wykryty przez testy w pierwszej iteracji mutant jest także wykrywany w jego następnych iteracjach. Tego typu zjawisko sprawia, że marnowany jest czas oraz moc obliczeniowa na proces generacji i uruchamiania zmutowanego kodu, który nie wnosi żadnej nowej informacji do projektu.

Przyczyny tego zjawiska mogą być dwojakie: albo dane miejsce w kodzie zawsze generuje mutanty trywialne, albo dany operator mutacyjny jest trywialny[1] względem projektu. Rozwiązanie problemu z przypadku pierwszego sprowadza się jedynie do oznaczenia miejsca w kodzie generującego mutanty trywialne tak, żeby przy następnej iteracji nie były generowane mutanty związane z tym miejscem. Rozwiązanie przypadku drugiego jest problemem bardziej złożonym, ponieważ wyłączenie całego operatora mutacyjnego (patrz rozdział 4.1.1) nie mając pewności, że jest on trywialny, może skutkować generacją problemów, powodem których są usterki, a następnie awarie całego oprogramowania.

Optymalizacją dla tego problemu może być wnioskowanie *post factum* polegające na osłabianiu bądź wzmacnianiu siły danego operatora mutacyjnego poprzez zwiększanie, bądź zmniejszanie prawdopodobieństwa generowanych mutantów danego typu. Proces ten może zostać przeprowadzony za pomocą określenia prawdopodobieństwa losowania każdego mutanta w następnej iteracji na podstawie tego, czy w obecnej został on wykryty przez testy czy nie. Im częściej mutant danego typu jest wykrywany przez testy, tym niższe powinno być prawdopodobieństwo jego wyboru w następnej iteracji, ponieważ mutant taki jest najprawdopodobniej trywialny.

Narzędziem użytym do tego celu jest matematyczny mechanizm generacji mutantów oparty o bayesowski [60] model prawdopodobieństwa wiążącego prawdopodobieństwa warunkowe dwóch zdarzeń warunkujących się nawzajem.

[1] Trywialny operator mutacyjny to taki, dla którego każdy bądź prawie każdy mutant jest zawsze wykrywany przez testy. Trywialność operatorów zawsze jest zrelatywizowana do danego projektu, bowiem skuteczność operatora jest silnie skorelowana z testami użytymi w konkretnym projekcie.

7.1 Podejście bayesowskie w testowaniu mutacyjnym

W celu określenia wartości prawdopodobieństwa z jakim losowany jest mutant należący do jakiegoś operatora mutacyjnego należy określić, jaki jest stosunek wykrytych i niewykrytych mutantów w danym projekcie. Im liczba wykrytych mutantów będzie wyższa w stosunku do liczby mutantów niewykrytych, tym prawdopodobieństwo losowania takiego mutanta powinno być niższe. Wnioskowanie na temat obecnej mutacji musi odbywać się na podstawie danych z poprzedniej iteracji procesu testowania mutacyjnego[2]. Wobec tego ustalanie prawdopodobieństwa odbywa się a posteriori.

W podejściu bayesowskim estymacje parametrów rozkładów nie są określane punktowo, jak w podejściu klasycznym, ale zawsze są charakteryzowane poprzez cały rozkład prawdopodobieństwa. Rozkład prawdopodobieństwa zmiennej losowej definiującej prawdopodobieństwo losowania mutantów dla danego operatora mutacyjnego powinien być definiowany przez stosunek liczby mutantów wykrytych oraz niewykrytych lub niepokrytych przez testy[3]. Wobec powyższego, w modelu generującym prawdopodobieństwo losowania mutantów wnioskowanie powinno odbywać się post factum: wartość prawdopodobieństwa dla jednej sesji mutacyjnej powinna być liczona na podstawie wartości z sesji poprzedniej oraz wyników sesji bieżącej. Opisane cechy problemu wskazują, że do jego rozwiązania można zastosować wnioskowanie bayesowskie.

We wnioskowaniu bayesowskim do estymacji rozkładu zmiennej losowej wykorzystuje się twierdzenie Bayesa, które pozwala „odwrócić" warunkowanie dwóch zmiennych. Niech Θ oznacza wektor szukanych parametrów rozkładu, a D — obserwowane dane. Wtedy wzór Bayesa można zapisać następująco:

$$P(\Theta \mid D) = \frac{P(D \mid \Theta)\, P(\Theta)}{P(D)}.$$

Wartość $P(D)$ w mianowniku jest czynnikiem normalizującym i można go obliczyć ze wzoru na prawdopodobieństwo całkowite. Jedyną problematyczną kwestią w podejściu bayesowskim jest rozkład a priori $P(\Theta)$, który jest nieznany. Można go przyjąć na podstawie danych historycznych lub też zdefiniować jako np. rozkład jednostajny, reprezentujący całkowity brak wiedzy a priori o rozkładzie parametru Θ.

W definiowanym modelu celem jest określenie, czy wartość wcześniej[4] ustalonego prawdopodobieństwa generacji mutanta określonego typu jest dobrze dopasowana pod warunkiem, że w bieżącej iteracji przeżyło S mutantów tego typu. Innymi słowy, statystyki przeżywalności mutantów reprezentują obserwowane dane D, a wzór Bayesa pozwala nam uaktualnić rozkład prawdopodobieństwa dla prawdopodobieństw generacji określonych typów mutantów poprzez wyliczenie parametrów Θ używanego rozkładu prawdopodobieństwa. Definicja 7.1 wprowadza używane w dalszej części niniejszego rozdziału pojęcie zbioru żywych mutantów.

[2] Wnioskować można na podstawie danych zebranych w obrębie jednej klasy, przestrzeni nazw, czy nawet całego projektu. Wtedy dane zebrane z jednej wersji projektu będą służyć jako dane wejściowe w następnej wersji.

[3] Jeżeli testy nie pokrywają jakiegoś mutanta, powinien być on traktowany tak jakby był niewykryty.

[4] Wcześniej, czyli w poprzednim uruchomieniu testów mutacyjnych.

Definicja 7.1 (zbiór żywych mutantów)**.** Jest to zbiór wszystkich mutantów generowanych przez dany operator mutacyjny, które nie zostały wykryte przez testy.

Dla ustalonego zbioru żywych mutantów jego liczność będzie oznaczana jako S_i, gdzie i jest indeksem tego zbioru. W naszym modelu poszczególne komponenty wzoru Bayesa oznaczać będą następujące prawdopodobieństwa:

- $P(\Theta)$ określa wartość prawdopodobieństwa a priori parametrów rozkładu prawdopodobieństwa dla ustalonego typu operatorów mutacyjnych w danym momencie.
- $P(\Theta \mid D)$ określa wartość prawdopodobieństwa a posteriori dla parametru Θ, po zaobserwowaniu statystyk przeżywalności D mutantów danego typu.
- $P(D \mid \Theta)$ określa oczekiwaną wartość statystyki przeżywalności D pod warunkiem, że prawdopodobieństwo generacji mutantów odpowiedniego typu jest parametryzowane przez Θ.
- $P(D)$ oznacza prawdopodobieństwo wystąpienia określonych danych (można je policzyć ze wzoru na prawdopodobieństwo całkowite:

$$P(D) = \int P(D|\Theta)P(\Theta)d\Theta.$$

Parametr D reprezentuje stosunek żywych mutantów do martwych w tzw. zbiorze mutacyjnym, zdefiniowanym w def. 7.2.

Definicja 7.2 (zbiór mutacyjny)**.** Jest to zbiór wszystkich mutantów generowanych przez dany operator mutacyjny.

Liczność zbioru mutacyjnego odpowiadającego ustalonemu operatorowi mutacyjnemu będzie oznaczana jako N_i, gdzie i jest indeksem tego operatora. Przez $D_i = (S_i, N_i - S_i)$ oznaczać będziemy dane obserwowane dotyczące mutantów generowanych przez i-ty operator mutacyjny. Wartości S_i i N_i są obserwowane na podstawie przeprowadzenia cyklu testów mutacyjnych.

Ponieważ analizę bayesowską przeprowadzać będziemy osobno dla każdego typu operatora mutacyjnego, zatem wszystkie powyższe wzory należy rozważać w kontekście konkretnego operatora. Na przykład, dla i-tego operatora poszukujemy Θ_i i wzór Bayesa przyjmie wtedy postać

$$P(\Theta_i \mid D_i) = \frac{P(D_i \mid \Theta_i) \cdot P(\Theta_i)}{P(D_i)}.$$

Funkcja dopasowująca prawdopodobieństwo losowania mutanta (dla i-tego operatora) na podstawie stosunku żywych i martwych mutantów będzie zatem wyrażona za pomocą prawdopodobieństwa warunkowego

$$P(\Theta_i \mid S_i, N_i - S_i).$$

Ten wzór jednak zawiera pewną wadę: dane ustalające prawdopodobieństwo Θ_i pochodzą tylko z poprzednio wykonanych testów mutacyjnych. Innymi słowy, jeżeli przykładowo efektem przeprowadzania testów mutacyjnych będzie stosunek żywych i martwych mutantów wynoszący 1:50, na tej podstawie zostanie ustalone prawdopodobieństwo Θ_i, a w następnym kroku ten sam stosunek wyniesie 2:1, to przy ustalaniu nowej wartości prawdopodobieństwa dla Θ_i zostanie uwzględniony tylko obecny stosunek, co sprawi, że wartość Θ_i może nagle znacząco wzrosnąć. Taki model nie byłby modelem uczącym się, gdyż wnioskowanie odbywałoby się jedynie na podstawie ostatnio wygenerowanych danych. Ten problem należy rozwiązać poprzez uwzględnienie we wzorze wartości z poprzednich testów mutacyjnych, co uzyskamy, stosując podejście bayesowskie.

Podejście bayesowskie będziemy stosować krokowo. Po przeprowadzeniu sesji testowania mutacyjnego otrzymujemy dane o przeżywalności mutantów, na podstawie których obliczamy prawdopodobieństwo a priori parametru Θ_i. W kolejnym kroku wartość ta staje się wartością a priori. Niech zatem $k = 1, 2, ...$ będzie krokiem oznaczającym daną sesję mutacyjną. Niech $X_{i,k}$ oznacza zmienną losową przyjmującą dwie możliwe wartości: $P(X_{i,k} = 1) = p$ oraz $P(X_{i,k} = 0) = 1 - p$, gdzie wartość 1 oznacza, że w k-tym kroku procesu testowania mutacyjnego i-ty operator mutacyjny będzie generowany z prawdopodobieństwem p.

Interesujące nas rozkłady zmiennych $X_{i,k}$ są rozkładami dwupunktowymi (Bernoulliego), parametryzowanymi jedną wartością: prawdopodobieństwem sukcesu w pojedynczej próbie Bernoulliego $p_{i,k}$. Tej wartości będziemy poszukiwać przy użyciu podejścia bayesowskiego.

Rozkładem sprzężonym[5] z rozkładem Bernoulliego jest rozkład beta [61], zatem $\Theta_{i,k}$ będzie reprezentować prawdopodobieństwo p rozkładu Bernoulliego dla i-tego operatora mutacyjnego w k-tym kroku:

$$\Theta_{i,k} = p_{i,k},$$

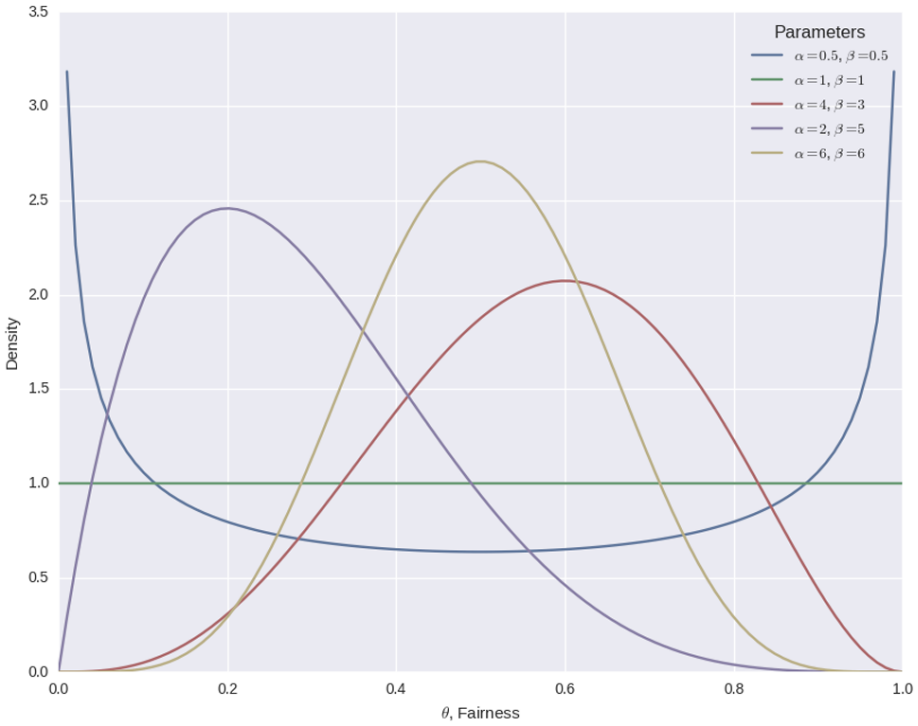
natomiast rozkład $\Theta_{i,k}$ będzie parametryzowany przez hiper-parametry rozkładu beta:

$$\Theta_{i,k} \sim Beta(\alpha_{i,k}, \beta_{i,k}).$$

Na rys. 7.1 przedstawiono kilka przykładowych rozkładów beta dla różnych parametrów[6].

[5]W statystyce bayesowskiej zasadniczym problemem jest odpowiedni dobór rozkładu a priori $P(\Theta)$. Zwykle zależy nam, aby po obliczeniu rozkładu a posteriori $P(\Theta \mid D)$ (uwzględniając $P(D \mid \Theta)$), był on tego samego typu co rozkład a priori. Mówimy, że rozkład a priori jest sprzężony z rozkładem $P(D \mid \Theta)$, jeśli wynikowy rozkład a posteriori pochodzi z tej samej rodziny rozkładów, co rozkład a priori. Dzięki temu w kolejnych krokach stosowania reguły Bayesa mamy do czynienia z rozkładam z tej samej rodziny, a zatem zmiana wynikająca z obserwacji jakichś danych w danej iteracji sprowadza się zazwyczaj do modyfikacji parametrów rozkładu a priori.

[6]Grafika została pobrana ze strony: https://www.quantstart.com/articles/Bayesian-Inference-of-a-Binomial-Proportion-The-Analytical-Approach



Rysunek 7.1: Przykładowe rozkłady beta dla różnych parametrów

W całej dalszej części niniejszego rozdziału dla uproszczenia posługiwać będziemy się jedynie parametrem k oznaczającym numer kroku, a pomijać będziemy indeks i oznaczający typ operatora, dla którego przeprowadzamy analizę. Będziemy zakładać *implicite*, że cała analiza wykonywana jest dla jednego, ustalonego typu i operatora mutacyjnego.

Gęstość rozkładu beta opisana jest w definicji 7.3. Jak wspomnieliśmy, jest on parametryzowany dwoma parametrami α, β, które z naszego punktu widzenia są hiper-parametrami modelu (gdyż służą do opisania rozkładu parametru innego rozkładu prawdopodobieństwa).

Definicja 7.3 (funkcja gęstości rozkładu beta)**.** Funkcja gęstości prawdopodobieństwa dla rozkładu beta zadana jest następującym wzorem [62]:

$$f(\Theta; \alpha, \beta) = \frac{\Theta^{\alpha-1}(1-\Theta)^{\beta-1}}{B(\alpha, \beta)},$$

gdzie α, β są hiperparametrami rozkładu, a B jest tzw. funkcją beta i we wzorze pełni rolę czynnika normalizującego: $B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt$.

Wartość oczekiwana zmiennej losowej dla tego rozkładu wyraża się następującym wzorem:

$$E\Theta = \int_0^1 \Theta f(\Theta; \alpha, \beta)\, d\Theta = \int_0^1 \Theta \frac{\Theta^{\alpha-1} \cdot (1-\Theta)^{\beta-1}}{B(\alpha, \beta)} = \frac{\alpha}{\alpha+\beta}.$$

Zakres całkowania przebiega wszystkie możliwe wartości szukanego przez nas parametru p rozkładu Bernoulliego, a zatem zbiór $[0, 1]$. Wartość oczekiwaną $E\Theta$ będziemy w k-tym kroku procesu interpretować jako estymator bayesowski wartości oczekiwanej dla parametru p_k wspomnianego wcześniej rozkładu Bernoulliego:

$$p_k = \frac{\alpha_k}{\alpha_k + \beta_k}.$$

Wprowadzamy zatem parametry α_k i β_k parametryzujące rozkład beta w k-tym kroku naszej procedury. Z teorii wiadomo, że wartości te można obliczyć z prostych wzorów, uwzględniających dane obserwowane oraz wartości tych parametrów z kroku poprzedniego:

$$\begin{aligned} \alpha_k &= S_k + \alpha_{k-1}, \\ \beta_k &= N_k - S_k + \beta_{k-1}. \end{aligned}$$

7.2 Model

Zgodnie z powyżej opisanymi założeniami odnośnie statystycznego wnioskowania bayesowskiego można zaproponować model matematyczny, który na podstawie obserwacji wyników mutacji będzie modyfikował prawdopodobieństwa generowania mutantów tak, aby częściej generować mutanty o wyższym prawdopodobieństwie niewykrycia. Jak wspomnieliśmy wcześniej, w modelu parametr p_k rozkładu dwupunktowego — czyli prawdopodobieństwo wykorzystania operatora mutacyjnego dla danego typu operatora — jest liczony oddzielnie i wyrażany jest „po bayesowsku", czyli jako rozkład beta.

7.2.1 Założenia modelu

Wartość prawdopodobieństwa dla zadanego operatora mutacyjnego będzie wyrażana poprzez wartość oczekiwaną rozkładu beta opisanego w podrozdz. 7.1. Zakłada się, że w procesie wytwarzania oprogramowania testowanie mutacyjne zostanie przeprowadzone K razy, za każdym razem na kolejnej wersji programu. Podczas procesu testowania mutacyjnego można wyróżnić I operatorów mutacyjnych. Parametry dla rozkładu beta w k-tym kroku będą zatem następujące:

- parametry α_k oraz β_k dla rozkładu będą takie, jak zdefiniowano w podrozdz. 7.1:

$$\begin{aligned} \alpha_k &= S_k + \alpha_{k-1} \\ \beta_k &= N_k - S_k + \beta_{k-1}. \end{aligned}$$

- sam rozkład będzie miał zatem postać

$$f(\Theta_k; S_k + \alpha_{k-1}, N_k - S_k + \beta_{k-1}).$$

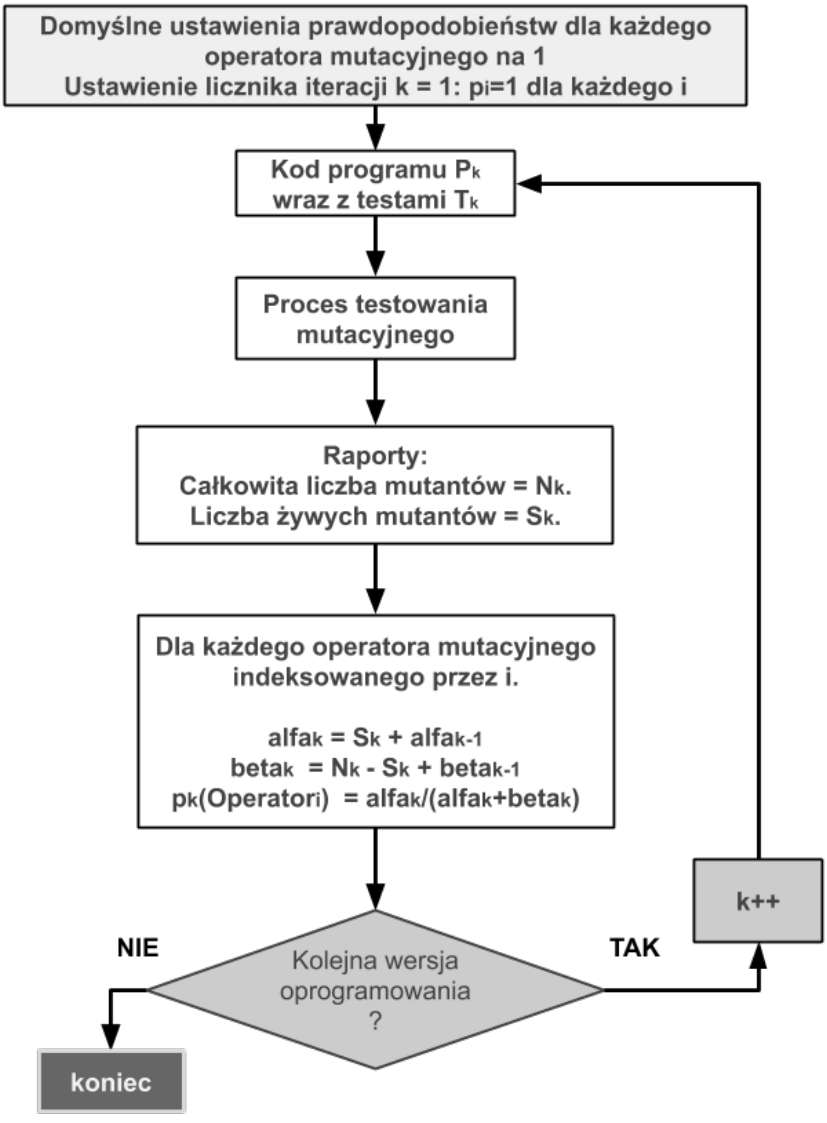
Zgodnie z założeniami, parametry powyższego rozkładu beta obrazują relację między liczbą żywych a martwych mutantów w danej iteracji. W każdej iteracji na podstawie S_k i N_k generowane są nowe wartości dla parametrów α_{k+1} i β_{k+1} rozkładu a posteriori. Modelują one „skalę trywialności" (patrz def. 3.2) operatora mutacyjnego, tzn. prawdopodobieństwo wykrycia mutanta wygenerowanego przy pomocy tego operatora na podstawie liczby wykrytych żywych mutantów.

Wobec powyższego, wartość oczekiwana zmiennej losowej Θ_k reprezentującej prawdopodobieństwo losowania mutantów danego typu w k-tym kroku będzie wyrażona następującym wzorem:

$$E\Theta_k = \frac{S_k + \alpha_{k-1}}{N_k - S_k + \beta_{k-1} + S_k + \alpha_{k-1}} = \frac{S_k + \alpha_{k-1}}{N_k + \beta_{k-1} + \alpha_{k-1}}.$$

7.2.2 Algorytm

Zachowanie modelu oraz definiowany przez niego algorytm przedstawiony został na schemacie na rys. 7.2.



Rysunek 7.2: Algorytm modelu bayesowskiego

7.3 Wykorzystane operatory mutacyjne

W celu weryfikacji tez dotyczących powyższego modelu oraz w celu sprawdzenia skuteczności generacji mutantów przy użyciu podejścia bayesowskiego został przeprowadzony szereg eksperymentów analizujących bilans zysków i strat, jakie niesie ze sobą ta technika.

Podczas przeprowadzania eksperymentów użyto następujących operatorów mutacyjnych:

Tabela 7.1: Działanie operatora CONDITIONALS BOUNDARY

Oryginalny kod	Mutant
<	<=
<=	<
>	>=
>=	>

Tabela 7.2: Działanie operatora INCREMENTS

Oryginalny kod	Mutant
+	-
-	+
*	/
/	*
%	*
&	|
|	&
^	&
<<	>>
>>	<<
>>>	<<

Tabela 7.3: Działanie operatora NEGATE_CONDITIONALS

Oryginalny kod	Mutant
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

- CONDITIONALS_BOUNDARY — zgodnie z opisaną w podrozdz. 3.3 nomenklaturą ten operator mutacyjny jest modyfikacją operatora mutacyjnego ROR. CONDITIONALS_BOUNDARY generuje mutanty według schematu opisanego w tabeli 7.1.
- INCREMENTS — jest to szczególny przypadek operatora ASR, którego działanie polega na zamianie inkrementacji na dekrementację i na odwrót.
- MATH — jest to połączenie operatora mutacyjnego ASR oraz COR. Operator działa zgodnie ze schematem opisanym w tabeli 7.2.
- NEGATE_CONDITIONALS — jest to zmodyfikowana wersja połączonych operatorów COR oraz ROR. Schemat działania operatora obrazuje tabela 7.3.
- RETURN_VALS — jest to odpowiednik operatora IREM. Działa w taki sam sposób.

Zgodnie z założeniami modelu oczekiwane jest zmniejszenie zasobów obliczeniowych[7] potrzeb-

[7] Przez zmniejszenie zasobów obliczeniowych rozumiemy tutaj zmniejszenie liczby danych, które program musi

nych do wykonywania operacji testowania mutacyjnego.

Użycie podejścia bayesowskiego powinno doprowadzić do zmniejszenia ogólnej liczby mutantów, które będą losowane w taki sposób, że dla danego operatora mutacyjnego prawdopodobieństwo losowania mutanta wzrośnie wraz z każdym niewykrytym mutantem, a zmaleje wraz z każdym wykrytym. Zarazem współczynnik pominiętych, niewylosowanych mutantów, które były niewykrywane przez testy będzie stosunkowo mały. W celu zbadania tego procesu przeprowadzono szereg eksperymentów opisanych w kolejnych podrozdziałach.

7.4 Eksperyment „Bayes per klasa"

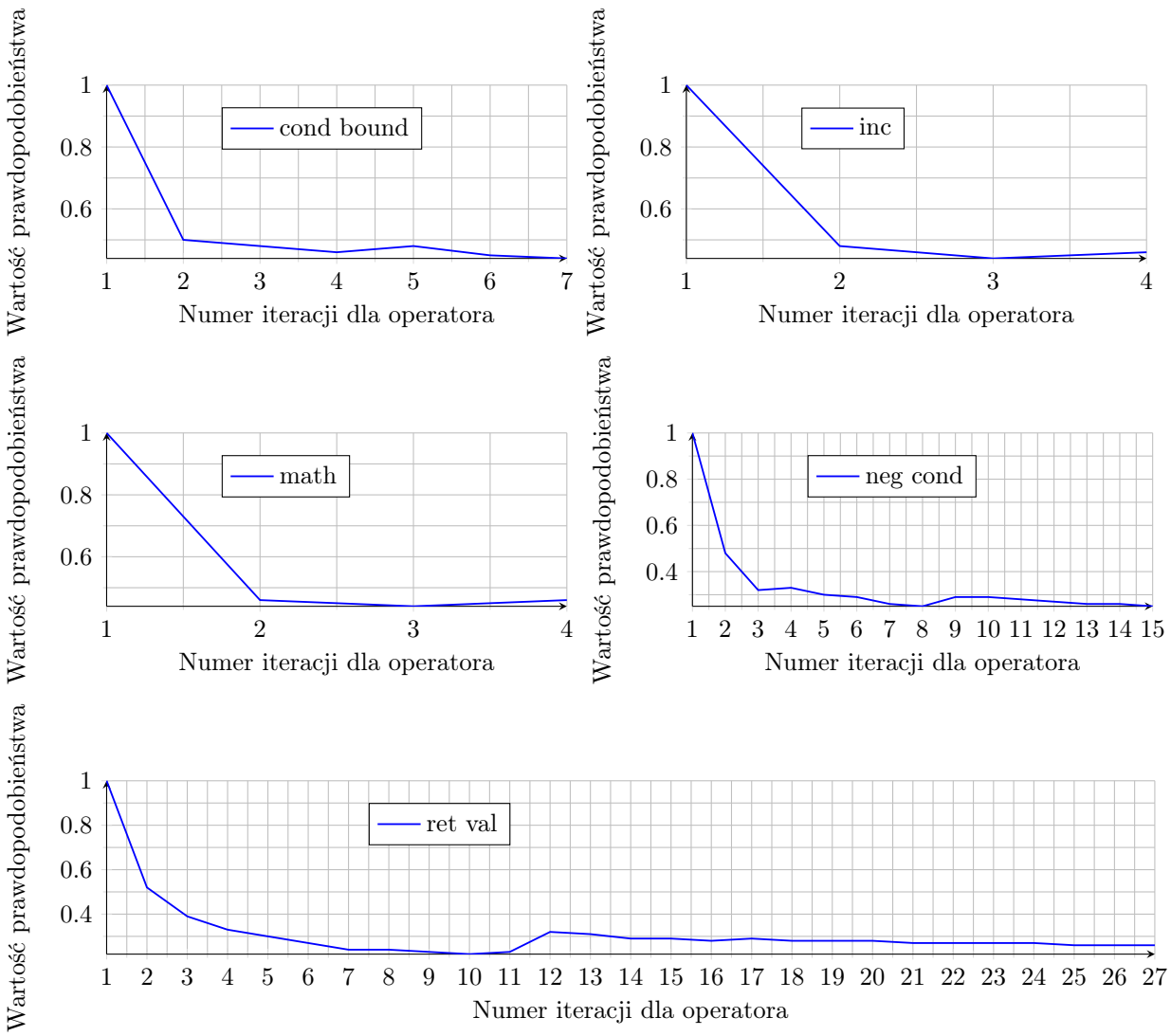
Eksperyment został przeprowadzony na oprogramowaniu JUnit v. 4.12.3. Testy mutacyjne były przeprowadzane per klasa. Po przetestowaniu każdej klasy aktualizowane były wartości parametrów odpowiadającego jej rozkładowi beta. Każdy operator mutacyjny był analizowany osobno i dlatego miał odpowiadającą mu parę (α_k, β_k). Mutacja całego projektu odbyła się tylko raz, wobec czego model bayesowski uczył się tylko raz na każdej klasie. W celu weryfikacji poprawności wyników z mutacji przy użyciu modelu bayesowskiego przeprowadzono także mutacje dla każdej z klas bez użycia tego modelu. Porównanie wyników z obu sesji mutacyjnych obrazuje liczbę pominiętych istotnych mutantów oraz skalę zwiększenia wydajności mutacji w związku z mniejszą liczbą mutantów, które były uruchamianie przez testy. Początkowe wartości hiperparametrów (α, β) ustawione są dla każdego operatora na $(12, 12)$.

7.4.1 Wyniki

Tabela 7.4.1 przedstawia wyniki przeprowadzenia testów mutacyjnych z użyciem modelu bayesowskiego na platformie JUnit 4.12.3. Użyte w tabeli symbole oznaczają odpowiednio: N – liczbę mutantów, Nc – liczbę niepokrytych mutantów, S – liczbę żywych mutantów, K – liczbę mutantów zabitych, p_{k-1} – prawdopodobieństwo obecne, p_k – nowe prawdopodobieństwo, cond_bound – operator CONDITIONALS_BOUNDARY, inc – operator INCREMENTS, math – operator MATH, neg_cond – operator NEGATE_CONDITIONALS, ret_val – operator RETURN_VALS.

przeanalizować podczas swojej pracy. W tym wypadku danymi jest liczba mutantów, na których muszą zostać wykonane testy jednostkowe. Uruchomienie każdego testu na danym mutancie zajmuje określoną jednostkę czasu, jak i mocy procesora. Zmniejszenie liczby mutantów jest w tym wypadku optymalizacją.

Operator	N	β	α	S	Nc	p_{k-1}	p_k	klasa	K
cond_bound	2	13	13	1	0	1.00	0.50	c01	1
cond_bound	1	14	13	0	0	0.50	0.48	c02	1
cond_bound	1	15	13	0	0	0.48	0.46	c03	1
cond_bound	1	15	14	0	1	0.46	0.48	c04	0
cond_bound	2	17	14	0	0	0.48	0.45	c05	2
cond_bound	1	18	14	0	0	0.45	0.44	c06	1
inc	1	13	12	0	0	1.00	0.48	c07	1
inc	2	15	12	0	0	0.48	0.44	c08	2
inc	1	15	13	0	1	0.44	0.46	c09	0
math	2	14	12	0	0	1.00	0.46	c10	2
math	1	15	12	0	0	0.46	0.44	c11	1
math	1	15	13	1	0	0.44	0.46	c12	0
neg_cond	5	15	14	0	2	1.00	0.48	c13	3
neg_cond	15	30	14	0	0	0.48	0.32	c14	15
neg_cond	1	30	15	1	0	0.32	0.33	c15	0
neg_cond	5	35	15	0	0	0.33	0.30	c16	5
neg_cond	2	37	15	0	0	0.30	0.29	c17	2
neg_cond	5	42	15	0	0	0.29	0.26	c18	5
neg_cond	4	46	15	0	0	0.26	0.25	c19	4
neg_cond	4	46	19	0	4	0.25	0.29	c20	0
neg_cond	1	47	19	0	0	0.29	0.29	c21	1
neg_cond	1	48	19	0	0	0.29	0.28	c22	1
neg_cond	4	52	19	0	0	0.28	0.27	c23	4
neg_cond	1	53	19	0	0	0.27	0.26	c24	1
neg_cond	1	54	19	0	0	0.26	0.26	c25	1
neg_cond	2	56	19	0	0	0.26	0.25	c26	2
ret_val	3	13	14	0	2	1.00	0.52	c27	1
ret_val	9	22	14	0	0	0.52	0.39	c28	9
ret_val	1	23	14	0	0	0.39	0.38	c29	1
ret_val	10	33	14	0	0	0.38	0.30	c30	10
ret_val	5	38	14	0	0	0.30	0.27	c31	5
ret_val	6	44	14	0	0	0.27	0.24	c32	6
ret_val	1	45	14	0	0	0.24	0.24	c33	1
ret_val	1	46	14	0	0	0.24	0.23	c34	1
ret_val	8	53	15	1	0	0.23	0.22	c35	7
ret_val	3	55	16	1	0	0.22	0.23	c36	2
ret_val	10	55	26	0	10	0.23	0.32	c37	0
ret_val	4	59	26	0	0	0.32	0.31	c38	4
ret_val	5	64	26	0	0	0.31	0.29	c39	5
ret_val	1	65	26	0	0	0.29	0.29	c40	1
ret_val	1	66	26	0	0	0.29	0.28	c41	1
ret_val	2	67	27	1	0	0.28	0.29	c42	1
ret_val	1	68	27	0	0	0.29	0.28	c43	1
ret_val	1	69	27	0	0	0.28	0.28	c44	1
ret_val	1	70	27	0	0	0.28	0.28	c45	1
ret_val	2	72	27	0	0	0.28	0.27	c46	2
ret_val	1	73	27	0	0	0.27	0.27	c47	1
ret_val	1	74	27	0	0	0.27	0.27	c48	1
ret_val	1	75	27	0	0	0.27	0.27	c49	1
ret_val	1	76	27	0	0	0.27	0.26	c50	1
ret_val	1	77	27	0	0	0.26	0.26	c51	1
ret_val	2	78	28	1	0	0.26	0.26	c52	1



Rysunek 7.3: Prawdopodobieństwa dla poszczególnych operatorów mutacyjnych

Wykresy 7.3 obrazują jak zmieniała się wartość prawdopodobieństwa poszczególnych operatorów mutacyjnych z każda iteracją mutacji przy zastosowaniu modelu bayesowskiego per klasa. Na wykresach można zauważyć tendencje do stabilizowania się wartości prawdopodobieństwa po kilku iteracjach. Dla operatora *RETURN_ VALS* widać że wraz z wzrostem liczby żywych mutantów wartość prawdopodobieństwa rośnie a następnie stabilizuje się na wyższej wartości.

Tabela 7.4 przedstawia sumaryczne statystyki dla poszczególnych operatorów mutacyjnych.

Tabela 7.4: Mutacja Bayes JUnit 4.12.3 sumaryczna

Operator	Wszystkie mutanty	Żywe	Brak pokrycia	Zabite	Przekroczony czas
CONDITIONALS BOUNDARY	8	1	1	6	
INCREMENTS	4	0	1	3	
MATH	4	1	0	3	
NEGATE CONDITIONALS	52	1	6	44	1
RETURN VALS	83	4	12	66	1
RAZEM	**151**	**7**	**20**	**122**	**2**

Tabela 7.5 przedstawia wyniki testów mutacyjnych z wyłączeniem modelu bayesowskiego także dla JUnit 4.12. W tej tabeli zostały pogrubione wartości wykazujące różnicę w stosunku do danych z tabeli 7.4 w liczbie mutantów żywych, z brakiem pokrycia oraz przekroczonym czasem wykonania.

Tabela 7.5: Mutacja bez Bayesa JUnit 4.12.3

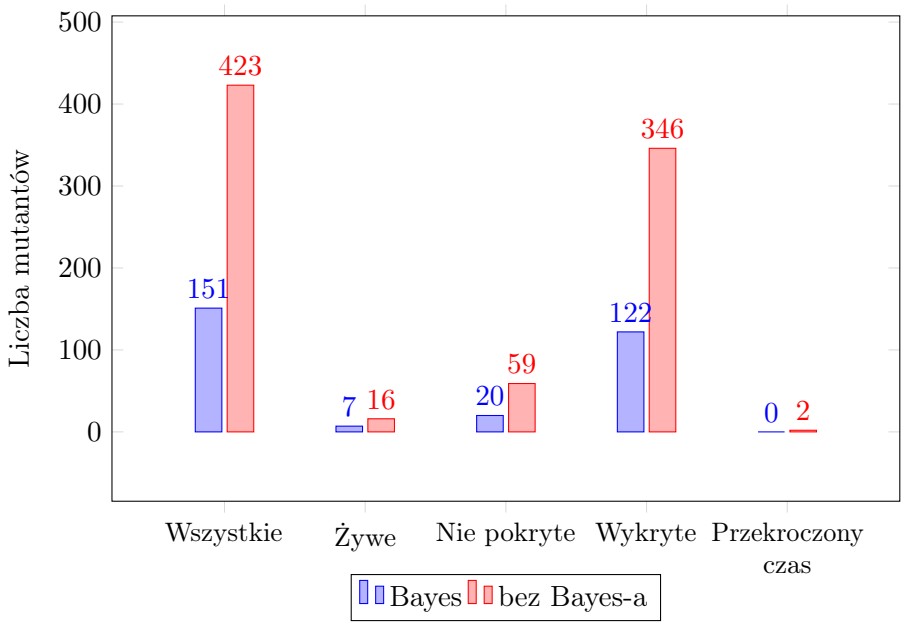
Operator	Wszystkie mutanty	Żywe	Brak pokrycia	Zabite	Przekroczony czas
CONDITIONALS BOUNDARY	9	1	1	7	
INCREMENTS	10	**1**	1	8	
MATH	8	1	**1**	6	
NEGATE CONDITIONALS	168	**4**	**15**	148	1
RETURN VALS	228	**9**	**41**	177	1
RAZEM	**423**	**16**	**59**	**346**	**2**

7.4.2 Wnioski z eksperymentu

Dane z tabel oraz wykresu z rys. 7.4 pokazują, że już przy pierwszym użyciu metody beyesowskiej można obserwować znaczny wzrost wydajności. Uruchomiono tylko 35% wszystkich mutantów, podczas gdy wykryto 43% z wszystkich żywych mutantów. Mimo wszystko współczynnik utraconych danych związanych z niewykrytymi mutantami jest znaczący. Problem ten będzie zanikał z czasem kolejnych sesji mutacyjnych, w związku z tym, że część mutantów niewylosowana w tej sesji, zostanie wylosowana w następnej. Ponadto, losowanie przy użyciu prawdopodobieństwa bayesowskiego jest procesem uczącym się, zatem współczynnik prawdopodobieństwa losowania mutanta będzie dopasowywał się do zadanego projektu coraz lepiej z każdą iteracją.

7.5 Eksperyment „Bayes dla wszystkich wersji"

Eksperyment drugi został przeprowadzony na takich samych zasadach jak pierwszy, z tym że tym razem mutacji z użyciem modelu bayesowskiego zostało poddanych sześć wersji oprogramowania

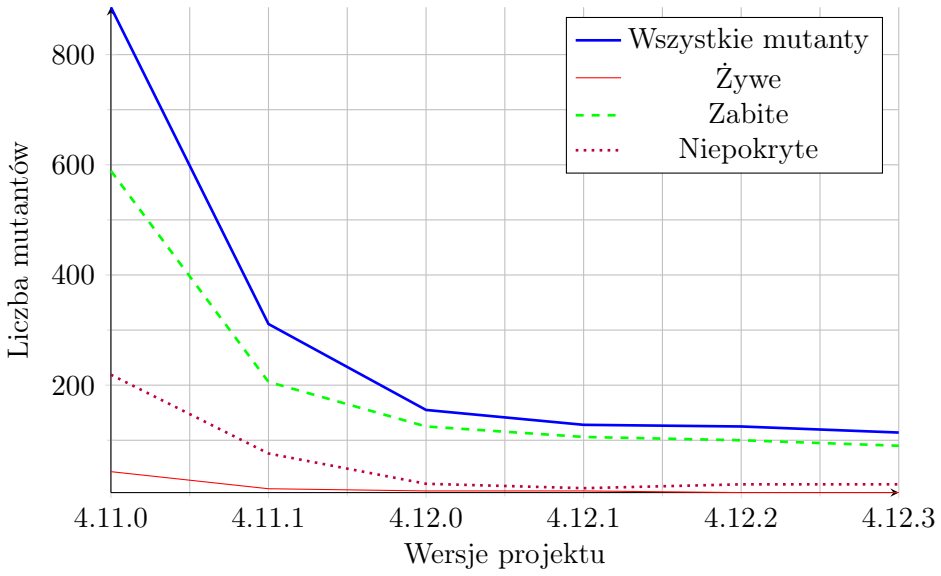


Rysunek 7.4: Porównanie mutacji bayesowskiej i zwykłej dla eksperymentu pierwszego

JUnit. Były to wersje: 4.11.0, 4.11.1, 4.12.0, 4.12.1, 4.12.2, 4.12.3. Taki szereg poszczególnych wersji programu był ciągiem uczącym dla modelu bayesowskiego.

7.5.1 Wyniki

Rysunek 7.5 przedstawia zmieniający się stosunek całkowitej liczby mutantów do mutantów, które przeżyły na przestrzeni wszystkich sześciu wersji projektu. Wartości na osi X odpowiadają kolejnym wersjom projektu JUnit.



Rysunek 7.5: Przebieg zmienności liczby mutantów dla eksperymentu drugiego

7.5.2 Wnioski z eksperymentu

Wykres z rys. 7.4 pokazuje, że liczba żywych mutantów podczas mutacji z użyciem modelu bayesowskiego uruchomionym tylko raz na projekcie JUnit 4.12.3 jest porównywalna z liczbą żywych mutantów w eksperymencie drugim, gdzie model bayesowski był nauczany od wersji 4.11.0 do 4.12.3. Natomiast liczba mutantów wykrytych spadła o 27%. Wskazuje to na wzrost efektywności modelu bayesowskiego wraz z kolejnymi iteracjami sesji testów mutacyjnych. Niestety, liczba niewykrytych żywych mutantów dalej sięga około 50%. Rozwiązania tego problemu należy szukać w odpowiedniej konfiguracji testów mutacyjnych z użyciem modelu bayesowskiego.

7.6 Eksperyment „Bayes – zbiór uczący jako cały projekt"

W obu powyższych eksperymentach można zaobserwować bardzo szybki spadek wartości prawdopodobieństwa, z jakim jest losowany mutant. Już po pierwszej iteracji ta wartość spada czasem aż o 0.5. Jest to spowodowane tym, że gdy model jest uczony per klasa, to początkowy zestaw klas może nie zawierać żadnych żywych mutantów bądź zawierać ich bardzo mało. To sprawia, że wartość prawdopodobieństwa zaczyna gwałtownie spadać, zgodnie z zasadą modelu bayesowskiego.

W celu poprawienia skuteczności modelu, czyli zmniejszenia liczby niewykrytych żywych mutantów, w tym eksperymencie nauczanie odbywało się na całym projekcie, a nie tak jak poprzednio per klasa.

7.6.1 Wyniki

Tabela 7.6: Wyniki po sesji testowej na projekcie JUnit v.4.11.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	29	22	1	6	0	1.00	0.36
INCREMENTS	23	17	2	4	0	1.00	0.38
RETURN VALS	479	305	25	149	0	1.00	0.35
MATH	6	4	0	2	0	1.00	0.47
NEGATE CONDITIONAL	318	241	15	58	4	1.00	0.24
RAZEM	**855**	**589**	**43**	**219**	**4**		

Tabela 7.7: Wyniki po sesji testowej na projekcie JUnit v.4.11.1

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	2	1	1	0	0	0.36	0.36
INCREMENTS	2	2	0	0	0	0.38	0.37
RETURN VALS	61	46	1	14	0	0.35	0.34
MATH	3	2	1	0	0	0.47	0.45
NEGATE CONDITIONALS	28	25	0	3	0	0.24	0.23
RAZEM	**96**	**76**	**3**	**17**	**0**		

Tabela 7.8: Wyniki po sesji testowej na projekcie JUnit v.4.12.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
INCREMENTS	2	1	1	0	0	0.36	0.36
CONDITIONALS BOUNDARY	2	2	0	0	0	0.38	0.37
RETURN VALS	61	46	1	14	0	0.35	0.34
MATH	3	2	1	0	0	0.47	0.45
NEGATE CONDITIONALS	28	25	0	3	0	0.24	0.23
RAZEM	**96**	**76**	**3**	**17**	**0**		

Tabela 7.9: Wyniki po sesji testowej na projekcie JUnit v.4.12.1

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
INCREMENTS	6	6	0	0	0	0.36	0.32
CONDITIONALS BOUNDARY	4	4	0	0	0	0.36	0.33
RETURN VALS	81	60	2	18	1	0.33	0.32
MATH	4	2	1	1	0	0.42	0.43
NEGATE CONDITIONALS	41	34	3	4	0	0.21	0.21
RAZEM	**136**	**106**	**6**	**23**	**1**		

Tabela 7.10: Wyniki po sesji testowej na projekcie JUnit v.4.12.2

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
INCREMENTS	2	2	0	0	0	0.32	0.31
CONDITIONALS BOUNDARY	3	3	0	0	0	0.33	0.32
RETURN VALS	85	65	3	17	0	0.32	0.31
MATH	5	4	1	0	0	0.43	0.40
NEGATE CONDITIONALS	32	29	1	2	0	0.21	0.20
RAZEM	**127**	**103**	**5**	**19**	**0**		

Tabela 7.11: Wyniki po sesji testowej na projekcie JUnit v.4.12.3

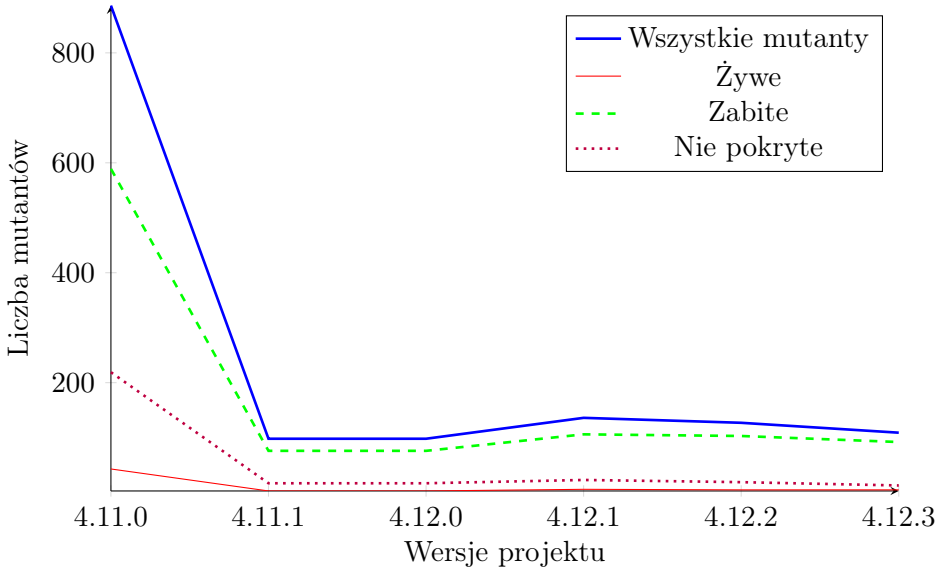
Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	1	1	0	0	0	0.32	0.31
INCREMENTS	3	2	1	0	0	0.31	0.31
RETURN VALS	66	53	2	10	1	0.31	0.30
MATH	3	1	1	1	0	0.40	0.41
NEGATE CONDITIONALS	38	35	1	2	0	0.20	0.19
RAZEM	**111**	**92**	**5**	**13**	**1**		

7.6.2 Wnioski z eksperymentu

Eksperyment trzeci dla ostatniej wersji JUnit (4.12.3) dał wyniki w postaci 109 przeanalizowanych mutantów, 5 żywych oraz 13 mutantów bez pokrycia testami. W stosunku do mutacji tej samej wersji bez użycia optymalizacji bayesowskiej nastąpił spadek wykrywalności żywych mutantów o 69%, mutantów niepokrytych testami o 78% oraz ogólny wzrost wydajności o 75% (wydajność rozumiemy tu w terminach ogólnej liczby analizowanych mutantów – im mniejsza ta liczba, tym wydajniejszy proces). Jest to poprawa w stosunku do eksperymentu drugiego i pierwszego, ale nadal liczba utraconych danych jest niezadowalająca.

7.7 Eksperyment Bayes dla $\alpha = 250$ oraz $\beta = 10$

Eksperyment trzeci, opisany w podrozdz. 7.6, pokazuje zwiększoną skuteczność modelu przy nauczaniu wykonanym na całym projekcie. Jednakże wartość prawdopodobieństwa w obrębie jed-



Rysunek 7.6: Przebieg zmienności liczby mutantów dla eksperymentu trzeciego

nego operatora mutacyjnego dalej może spadać zbyt szybko. Jest to związane z niedostosowaniem wartości parametrów α oraz β do ogólnej liczby mutantów.

Szybkość spadku lub wzrostu wartości zmiennej losowej dla modelu bayesowskiego opisanego w rozdziale 7.2 może być sterowana za pomocą stosunku α do β oraz stosunku obu parametrów do ogólnej liczby mutantów. W związku z powyższym, eksperyment czwarty został przeprowadzony ze zmienionymi wartościami parametrów α oraz β. Tak samo jak w eksperymencie trzecim, nauczanie odbywało się na całym projekcie.

7.7.1 Wyniki

Dla parametrów α i β zaproponowano początkowe wartości $\alpha = 250$ oraz $\beta = 10$. Wartość 250 dla parametru α została zaproponowana na podstawie średniej liczby mutantów dla różnych wersji oprogramowania JUnit[8].

Tabela 7.12: Wyniki po sesji testowej na projekcie JUnit v.4.11.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	29	22	1	6	0	1.00	0.94
INCREMENTS	23	17	2	4	0	1.00	0.95
RETURN VALS	479	305	25	149	0	1.00	0.67
MATH	6	4	0	2	0	1.00	0.95
NEGATE CONDITIONAL	318	241	15	58	4	1.00	0.67
RAZEM	**855**	**589**	**43**	**219**	**4**		

[8]W tym eksperymencie średnia wartość wynosi około 600 mutantów. Użyto około 40% tej wartości, bowiem przy wyższych wartościach dla α zmiana prawdopodobieństwa byłaby zbyt wolna.

Tabela 7.13: Wyniki po sesji testowej na projekcie JUnit v.4.11.1

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	7	5	1	1	0	0.94	0.88
INCREMENTS	6	3	2	1	0	0.95	0.90
RETURN VALS	184	132	4	48	0	0.67	0.54
MATH	5	3	1	1	0	0.95	0.94
NEGATE CONDITIONALS	143	121	3	19	0	0.67	0.52
RAZEM	**361**	**264**	**27**	**70**	**0**		

Tabela 7.14: Wyniki po sesji testowej na projekcie JUnit v.4.12.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	7	6	1	0	0	0.88	0.87
INCREMENTS	9	7	1	1	0	0.90	0.88
RETURN VALS	131	106	4	20	1	0.54	0.49
MATH	7	5	1	1	0	0.94	0.92
NEGATE CONDITIONALS	89	78	2	9	0	0.52	0.47
RAZEM	**243**	**202**	**9**	**31**	**1**		

Tabela 7.15: Wyniki po sesji testowej na projekcie JUnit v.4.12.1

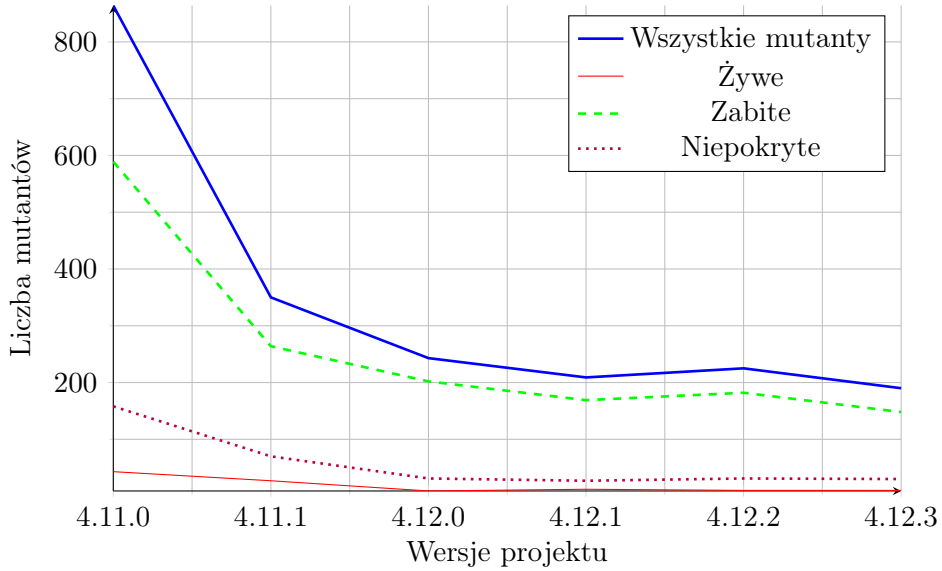
Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	8	6	1	1	0	0.87	0.85
INCREMENTS	8	6	1	1	0	0.88	0.86
RETURN VALS	115	92	7	15	1	0.49	0.46
MATH	7	5	1	1	0	0.92	0.91
NEGATE CONDITIONALS	71	60	2	9	0	0.47	0.44
RAZEM	**209**	**169**	**12**	**27**	**1**		

Tabela 7.16: Wyniki po sesji testowej na projekcie JUnit v.4.12.2

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	8	6	1	1	0	0.85	0.84
INCREMENTS	8	7	0	1	0	0.86	0.84
RETURN VALS	119	92	7	19	1	0.46	0.43
MATH	8	6	1	1	0	0.91	0.89
NEGATE CONDITIONALS	82	71	1	9	1	0.44	0.41
RAZEM	**225**	**182**	**10**	**31**	**2**		

Tabela 7.17: Wyniki po sesji testowej na projekcie JUnit v.4.12.3

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	8	6	1	1	0	0.84	0.82
INCREMENTS	9	7	1	1	0	0.84	0.83
RETURN VALS	95	69	4	21	1	0.43	0.42
MATH	7	6	0	1	0	0.89	0.87
NEGATE CONDITIONALS	71	60	4	6	1	0.41	0.39
RAZEM	**190**	**148**	**10**	**30**	**2**		



Rysunek 7.7: Przebieg zmienności liczby mutantów dla eksperymentu czwartego

7.7.2 Wnioski z eksperymentu

W eksperymencie czwartym liczba żywych mutantów jest tylko o 38% mniejsza w stosunku do liczby żywych mutantów w przypadku, gdy nie stosowano optymalizacji opartej na modelu bayesowskim.

Wzrost wydajności mierzony za pomocą ogólnej liczby analizowanych mutantów wynosi 55% w stosunku do mutacji klasycznej, a liczba niewykrytych mutantów, które nie były pokryte przez testy, jest mniejsza o 50%.

Na podstawie tych danych możemy wnioskować, że przy odpowiednim ustawieniu parametrów w modelu możemy uzyskać zadowalający wzrost wydajności przy stosunkowo niskim ryzyku utraty istotnych danych.

Istotną cechą modelu jest to, że działa on lepiej wraz z coraz większą liczbą projektów, jakie są przez niego analizowane. Przy małej liczbie projektów lub wersji tego samego projektu w modelu kluczową rolę będzie odgrywać początkowa konfiguracja. Gdy konfiguracja będzie sprawiała, że model będzie dostosowywał się bardzo gwałtownie zmieniając znacznie wartość prawdopodobieństwa stosowanego w losowaniu mutantów już po pierwszej iteracji, to ze względu na małą liczbę iteracji model nie będzie miał czasu na ustabilizowanie się. Jeżeli natomiast parametry zostaną ustawione w taki sposób, że wartość prawdopodobieństwa będzie się zmieniać tylko nieznacznie, to model będzie dostosowywał się zbyt wolno. W praktyce ten problem będzie sprawiał, że w początkowych fazach tworzenia oprogramowania, gdy iteracji będzie mało, pożądanym może być poprawianie parametryzacji modelu po każdej iteracji w celu dostosowania tempa adaptacji modelu.

7.8 Eksperyment „Bayes dopasowywany do projektu"

Na podstawie wyników z eksperymentu czwartego można zauważyć, że ogólna liczba mutantów w pierwszej iteracji projektu będzie rzutować na sposób, w jaki będzie się zmieniać prawdopodobieństwo losowania mutantów. Zgodnie ze wzorem na wartość oczekiwaną prawdopodobieństwa opisaną w rozdziale 7.2 widzimy, że bardzo duża liczba mutantów dla danego operatora staje się dominującym współczynnikiem we wzorze, co sprawia, że spadek lub wzrost wartości prawdopodobieństwa może znacznie przyspieszać.

W efekcie, model dostosowuje się w różny sposób dla różnych operatorów. Dla tych z małą liczbą mutantów proces uczenia jest zbyt wolny, a dla tych z dużą — zbyt szybki, co sprawia, że wartości prawdopodobieństwa mogą zmienić się drastycznie w obrębie jednej iteracji.

Zgodnie z powyższym, w tym eksperymencie model bayesowski został skonfigurowany po uprzedniej analizie liczby mutantów dla każdego operatora. Przy wartościach od 1 do 100 parametr α został ustawiony na 99. Przy wartościach powyżej 100 parametr α został ustawiony na dokładną liczbę mutantów danego operatora. Parametr β zawsze jest ustawiany na 10.

7.8.1 Wyniki

Tabela 7.18: Wyniki po sesji testowej na projekcie JUnit v.4.11.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	29	22	1	6	0	1.00	0.77
INCREMENTS	23	17	2	4	0	1.00	0.80
RETURN VALS	479	305	25	149	0	1.00	0.67
MATH	6	4	0	2	0	1.00	0.88
NEGATE CONDITIONALS	318	241	15	58	4	1.00	0.60
RAZEM	**855**	**589**	**43**	**219**	**4**		

Tabela 7.19: Wyniki po sesji testowej na projekcie JUnit v.4.11.1

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	5	4	0	1	0	0.77	0.75
INCREMENTS	4	2	1	1	0	0.80	0.79
RETURN VALS	125	83	1	41	0	0.67	0.63
MATH	5	3	1	1	0	0.88	0.86
NEGATE CONDITIONALS	87	74	1	11	1	0.60	0.55
RAZEM	**226**	**166**	**4**	**55**	**1**		

Tabela 7.20: Wyniki po sesji testowej na projekcie JUnit v.4.12.0

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	7	5	1	1	0	0.75	0.73
INCREMENTS	10	8	1	1	0	0.79	0.75
RETURN VALS	150	111	6	32	1	0.63	0.59
MATH	7	5	1	1	0	0.86	0.83
NEGATE CONDITIONALS	93	83	2	8	0	0.55	0.50
RAZEM	**267**	**212**	**11**	**43**	**1**		

Tabela 7.21: Wyniki po sesji testowej na projekcie JUnit v.4.12.1

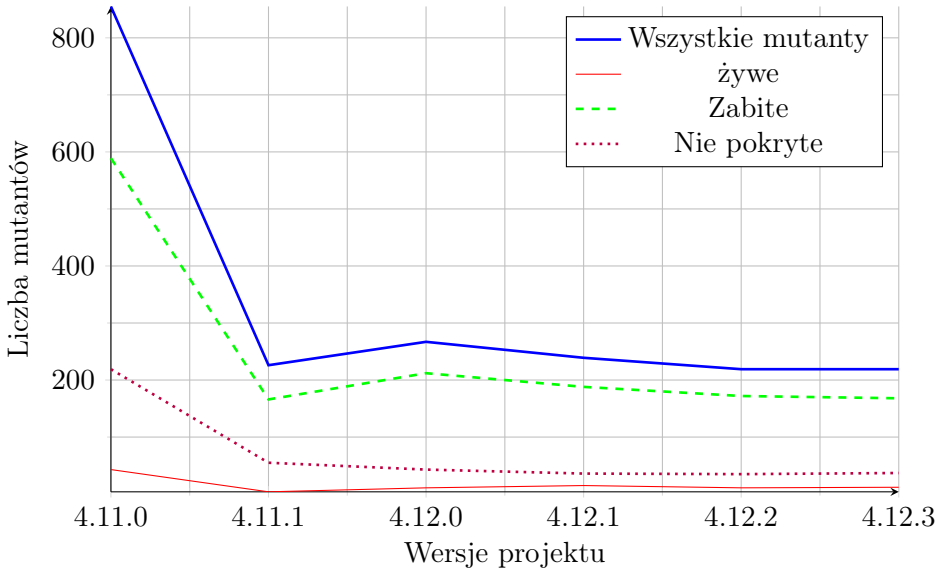
Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	8	6	1	1	0	0.73	0.70
INCREMENTS	10	8	1	1	0	0.75	0.71
RETURN VALS	121	91	8	22	0	0.59	0.56
MATH	7	5	1	1	0	0.83	0.80
NEGATE CONDITIONALS	93	78	4	11	0	0.50	0.47
RAZEM	**239**	**188**	**15**	**36**	**0**		

Tabela 7.22: Wyniki po sesji testowej na projekcie JUnit v.4.12.2

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	4	3	0	1	0	0.70	0.69
INCREMENTS	7	5	1	1	0	0.71	0.69
RETURN VALS	132	101	5	25	1	0.56	0.53
MATH	6	4	1	1	0	0.80	0.78
NEGATE CONDITIONALS	70	59	4	7	0	0.47	0.44
RAZEM	**219**	**172**	**11**	**35**	**1**		

Tabela 7.23: Wniki po sesji testowej na projekcie JUnit v.4.12.3

Operator	Wszystkie mutanty	Zabite	Żywe	Brak pokrycia	Przekroczony czas	p. stare	p. nowe
CONDITIONALS BOUNDARY	6	5	0	1	0	0.69	0.67
INCREMENTS	9	7	1	1	0	0.69	0.67
RETURN VALS	124	91	6	26	1	0.53	0.51
MATH	6	4	1	1	0	0.78	0.76
NEGATE CONDITIONALS	74	61	4	8	1	0.44	0.42
RAZEM	**219**	**169**	**11**	**37**	**2**		



Rysunek 7.8: Przebieg zmienności liczby mutantów dla eksperymentu piątego

7.8.2 Wnioski z eksperymentu

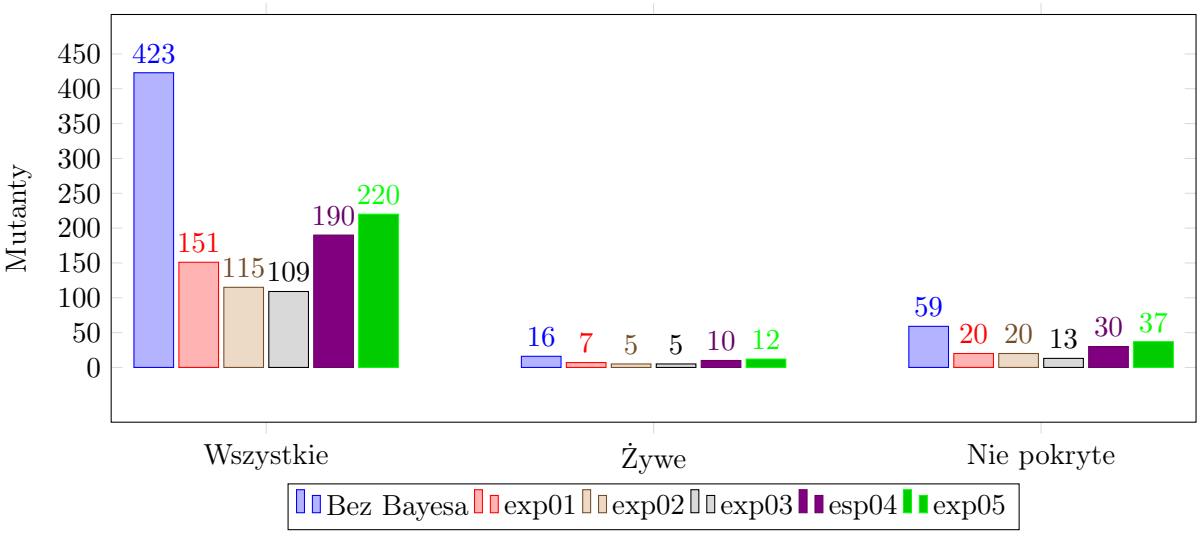
Porównując wyniki zebrane w eksperymencie piątym dla ostatniej wersji programu JUnit (tabela 7.23) oraz dane z wykonania mutacji na tym samym oprogramowaniu, ale bez użycia modelu bayesowskiego (tabela 7.5) widzimy, że utrata danych w wyniku niewylosowania żywych mutantów wynosi 25%. W wyniku niewylosowania odsetek niepokrytych mutantów wynosi 48%, natomiast wzrost wydajności wynosi 52%. Ten eksperyment pokazuje, że przy odpowiednim dobraniu parametrów α oraz β jest możliwe zaprojektowanie rozsądnego modelu optymalizującego testowanie mutacyjne oparte na prawdopodobieństwie bayesowskim.

7.9 Wyniki zbiorcze

W tabeli 7.24 zostały przedstawione wyniki zbiorcze z wszystkich eksperymentów. Wyniki zostały porównane ze sobą oraz z procesem mutacji nieuwzględniającej optymalizacji bayesowskiej. Szczególnie dobry wynik optymalizacji można obserwować w danych z eksperymentu piątego. W tym eksperymencie utrata danych związana z niewylosowaniem żywego mutanta w najgorszym wypadku wyniosła 8 a w najlepszym 2. Dla eksperymentu piątego średnia liczba pominiętych żywych mutantów wyniosła 5.2. W eksperymencie pierwszym operacja testowania mutacyjnego przebiegała tylko dla projektu 4.12.3.

7.10 Wnioski

Wydajność procesu testowania mutacyjnego szczególnie dla dużych projektów może zostać podniesiona poprzez odpowiednie dopasowanie modelu optymalizacyjnego stosującego prawdopodobieństwo bayesowskie. Proces dopasowania modelu możemy obserwować na rysunku 7.9. Ważnym elementem w procesie dopasowywania modelu jest pilnowanie, by nauczanie nie odbywało się lokalnie – model powinien uczyć się na przestrzeni kilku wersji oprogramowania.



Rysunek 7.9: Porównanie mutacji bayesowskiej i zwykłej na przestrzeni wszystkich eksperymentów

Tabela 7.24: Wyniki zbiorcze wszystkich eksperymentów

Wersja/eksperyment	Wszytkie	Żywe	Brak pokrycia	Błąd uruchomienia	Przekroczony czas	Zabite
JUnit 4.11.0						
Bez Bayesa	**886**	**43**	**219**	**31**	**4**	**589**
eksperyment 2	886	43	219	31	4	589
eksperyment 3	886	43	219	31	4	589
eksperyment 4	886	43	219	31	4	589
eksperyment 5	886	43	219	31	4	589
JUnit 4.11.1						
Bez Bayesa	**385**	**20**	**103**	**13**	**1**	**248**
eksperyment 2	311	12	76	14	3	206
eksperyment 3	98	3	17	2	0	76
eksperyment 4	361	27	70	0	0	264
eksperyment 5	226	4	55		1	166
JUnit 4.12						
Bez Bayesa	**423**	**17**	**63**	**0**	**1**	**342**
eksperyment 2	128	8	13	0	1	106
eksperyment 3	98	3	17	2	0	76
eksperyment 4	243	9	31	0	1	202
eksperyment 5	267	11	43	0	1	212
JUnit 4.12.1						
Bez Bayesa	**423**	**17**	**59**	**0**	**1**	**346**
eksperyment 2	125	5	20	0	0	100
eksperyment 3	136	6	23	0	1	106
eksperyment 4	209	12	27	0	1	169
eksperyment 5	239	15	36	0	0	188
JUnit 4.12.2						
Bez Bayesa	**423**	**16**	**59**	**0**	**2**	**346**
eksperyment 2	155	8	21	0	1	125
eksperyment 3	127	5	19	0	0	103
eksperyment 4	225	10	31	0	2	182
eksperyment 5	219	11	35	0	1	172
JUnit 4.12.3						
Bez Bayesa	**423**	**16**	**59**	**0**	**2**	**346**
eksperyment 1	151	7	20	0	2	122
eksperyment 2	115	5	20	0	0	90
eksperyment 3	111	5	13	0	1	92
eksperyment 4	190	10	30	0	2	148
eksperyment 5	219	11	37	0	2	169

Dane przedstawiają wartości uzyskane dla końcowej iteracji dla każdego eksperymentu oraz wartości bez stosowania optymalizacji.

Na podstawie danych z eksperymentu podsumowanych w tabeli 7.24 można stwierdzić, że wykazano, iż stosowanie statystycznych model uczących opartych na podejściu bayesowskim może znacznie zwiększyć wydajność procesu testowania mutacyjnego poprzez redukcję liczby mutantów często wykrywanych przez testy. W praktycznych zastosowaniach prawdopodobieństwo, z jakim jest losowany mutant w modelu, powinno być cały czas kontrolowane po każdej iteracji tak, żeby jego wartość cały czas odpowiadała naszym potrzebom. Pewne straty wynikające z faktu utraty danych poprzez niewylosowanie istotnych mutantów mogą być korygowane przy następnych losowaniach stosowanych do kolejnych iteracji.

Gdy prawdopodobieństwo dla danego operatora mutacyjnego spada do wartości bliskich zeru,

może to sugerować, że mutanty powiązane z tym operatorem mutacyjnym są mutantami trywialnymi.

Model bayesowski można próbować usprawniać, wprowadzając do niego między innymi następujące modyfikacje:

- Zapamiętywanie żywych i niepokrytych mutantów w celu gwarancji losowania ich w następnych iteracjach[9].
- Ustanowienie stałej w modelu w miejsce odpowiadające liczbie żywych mutantów co sprawi, że jedynym czynnikiem dostosowującym parametry modelu będzie liczba żywych i niepokrytych mutantów[10].
- Kolejną ciekawą modyfikacją modelu może być zastąpienie funkcji dopasowującej z opartej na prawdopodobieństwie bayesowskim na inną funkcję, również generującą prawdopodobieństwo losowania.

Prace przygotowawcze związane z przeprowadzeniem badań oraz dostosowaniem oprogramowania w taki sposób, żeby pozwalało na zastosowanie modeli predykcyjnych podczas procesu testowania mutacyjnego trwały ponad 7 miesięcy.

7.11 Przyszłe badania

Badania nad optymalizacją i usprawnianiem wyżej opisanego modelu są dalej prowadzone. Obecnie prace polegają na wprowadzaniu do systemu modułu, dzięki któremu można będzie deklarować własne funkcje optymalizacyjne determinujące prawdopodobieństwo losowania mutanta. Trwają także prace nad zastosowaniem w modelu skryptów mutacyjnych opisanych w rozdziale 4.1.1, które pozwolą na rozpatrywanie prawdopodobieństwa losowania mutanta nie tylko ze względu na typ operatora mutacyjnego, ale także ze względu na elementy składowe tego operatora. Wprowadzenie skryptów mutacyjnych pozwoli na bardziej precyzyjne określanie prawdopodobieństwa losowania mutantów. Planowane jest także (przy współpracy z SJSI[11]) zbudowanie klastra obliczeniowego umożliwiającego mutacje kodu, który będzie zawierał w sobie optymalizacje procesu testowania oparte na prawdopodobieństwie bayesowskim.

Zastosowanie takiego modelu generacji mutantów opartego na prawdopodobieństwie bayesowskim poza zwiększeniem wydajności testowania mutacyjnego może mieć także zastosowanie w orzekaniu czy dany mutant lub operator jest trywialny czy też nie (patrz def. 3.2).

[9] Szczególnie istotne może się to okazać przy mutantach niepokrytych testami, bowiem po pokryciu takiego mutanta testami w następnej iteracji nie jest wiadome czy testy są silne.

[10] Taki model jest modelem, w którym możemy kontrolować, jak bardzo istotny dla dopasowania będzie każdy żywy mutant.

[11] Stowarzyszenie Jakości Systemów Informatycznych

7.12 Identyfikacja zagrożeń dla poprawności badań

Głównym zagrożeniem dla poprawności wyników badań jest liczba iteracji projektu, jaka została użyta do przeprowadzenia eksperymentu. Model oparty na prawdopodobieństwie bayesowskim dostosowuje się coraz lepiej wraz ze wzrostem danych w zbiorze uczącym. Ponadto model był użyty na oprogramowaniu, gdzie proces testowania mutacyjnego nie był używany w procesie jego wytwarzania, co skutkowało tym, że mutanty przenosiły się z wersji na wersję, zaburzając wyniki eksperymentu. Przeprowadzenie eksperymentu na większą skalę ze względu na ograniczoną liczbę zasobów nie było możliwe.

Rozdział 8

TDD+M — metodyka wytwarzania oprogramowania wykorzystująca testowanie mutacyjne

Badania opisane w poniższym rozdziale zostały opublikowane w Software Quality Journal [63].

Analiza opisu efektywnych systemów mutacyjnych sugeruje, że takie systemy mogą mieć trwałe i praktyczne zastosowanie w deweloperskich procesach wytwarzania oprogramowania. Profesjonalnie wytwarzane oprogramowanie zawsze opiera się o jakąś dobrze zdefiniowaną i (zazwyczaj) empirycznie zweryfikowaną pod kątem skuteczności metodę[1]. W kontekście weryfikacji skuteczności technik mutacyjnego testowania oprogramowania szczególnie istotne są metodyki zwinne oparte na koncepcji wielu krótkich iteracji, takie jak: Scrum, Kanban, XP, Lean IT itp., a także praktyki deweloperskie wykorzystujące iteracyjne podejście "najpierw test" (ang. test first), takie jak wytwarzanie sterowane testami (ang. TDD – Test-Driven Development).

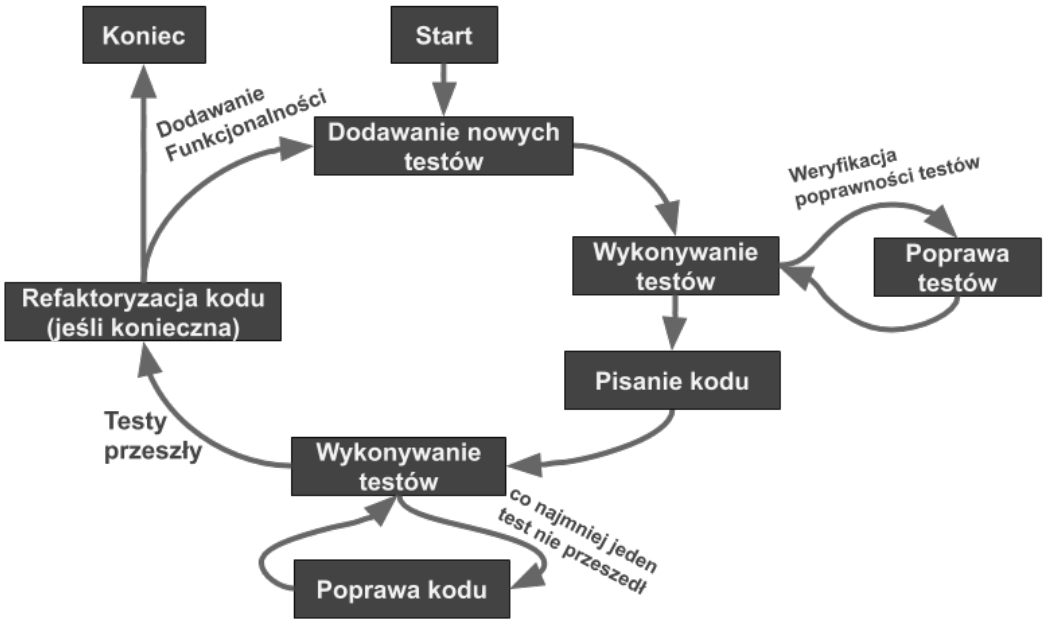
Według najlepszej wiedzy autora do tej pory nie interesowano się tematyką testowania mutacyjnego jako integralnego elementu zwinnych metod wytwarzania oprogramowania. W literaturze przedmiotu istnieje niewielka liczba prac naukowych podejmujących tę tematykę. Jedną z nich jest praca [64], która podnosi problem zastosowania testów mutacyjnych jako wzbogacenie metodyk wytwarzania oprogramowania, ale rozważa go jedynie spekulatywnie i teoretycznie, bez jakichkolwiek empirycznych badań skuteczności podejścia w praktyce. W Internecie można jednak znaleźć szereg informacji wskazujących na to, że część firm zajmujących się wytwarzaniem oprogramowania zaczyna stosować testowanie mutacyjne jako uzupełnienie używanych do tej pory metodyk wytwarzania oprogramowania. Jedną z takich informacji można znaleźć na stronie twórcy systemu PIT w dziale związanym z doświadczeniami użytkowników systemu [65]. Inne przykłady z ostatnich lat można znaleźć w pracach [66, 67, 68].

[1] W języku polskim często na określenie metody błędnie używa się pojęcia „metodyka", które oznacza naukę o metodach. Metoda wytwarzania oprogramowania to zbiór reguł i zasad definiujących sposób, w jaki oprogramowanie jest projektowane, wytwarzane, testowane oraz wdrażane. Najpopularniejsze obecnie stosowane metody to metody zwinne. Test-Driven Development jest praktyką deweloperską często wykorzystywaną właśnie w modelach zwinnych.

8.1 Test-Driven Development

Test-Driven Development jest zwinną metodyką wytwarzania oprogramowania stworzoną przez Kenta Becka [69], polegającą na wytwarzaniu kodu w sposób iteracyjny zaczynając od testów, które tworzą zarys przyszłych funkcjonalności oprogramowania. Kolejne kroki w podejściu TDD to:

1. Napisanie testu dla planowanej do implementacji funkcjonalności.
2. Uruchomienie testów (nowy test nie powinien przejść, ponieważ nie ma jeszcze dla niego kodu).
3. Implementacja nowych funkcjonalności pokrywających się z założeniami zawartymi w testach aż do momentu, gdy wszystkie testy będą zwracać wynik pozytywny.
4. Refaktoryzacja kodu w celu poprawiania struktury i czytelności oprogramowania.
5. Powrót do punktu pierwszego.



Rysunek 8.1: Metodyka Test-Driven Development

Krok czwarty (refaktoryzacja) związany jest z tym, że oprogramowanie tworzone jest w wielu krótkich iteracjach. Każda z nich ma przyrostowy charakter, tzn. w każdej iteracji do kodu zostaje dodana minimalna ilość nowego kodu, który pozwala zdać nowe testy, zaprojektowane na początku iteracji. Aby częste wprowadzanie wielu zmian nie wprowadziło do kodu źródłowego nieporządku, konieczny jest krok „porządkujący", czyli właśnie refaktoryzacja. Diagram na rys. 8.1 opisuje sposób stosowania metodyki TDD.

Istnieje bogata literatura na temat metod i podejść związanych z TDD. Przede wszystkim cytowana już wcześniej książka Kenta Becka [69], który jest autorem metodyki. Kolejną ciekawą

pozycją jest książka Dave'a Astelsa [20], będąca praktycznym poradnikiem stosowania TDD z punktu widzenia programisty. Źródłem wiedzy o TDD nie są jednak tylko książki. Powstał szereg publikacji związanych z tą tematyką. Szczególnie interesujące są te pozycje, które badają TDD od strony praktycznej oraz weryfikują, jak ten teoretyczny model metodyki sprawdza się w realiach profesjonalnego wytwarzania oprogramowania. Tego typu pracą jest na przykład artykuł [70] oceniający wpływ rozwoju TDD na jakość oprogramowania. Autor szczególną uwagę zwraca na jakość projektowania oprogramowania, uwzględniając również implikacje pedagogiczne. Kolejną równie ciekawą pracą jest artykuł [71]. Omawia on rozwój oprogramowania wpierany przez metodykę TDD dla dwóch różnych aplikacji firmy Microsoft: Windows i MSN.

Duża liczba publikacji na temat TDD sugeruje, że jest to metoda często używana, która stała się *de facto* standardową praktyką iteracyjnego wytwarzania oprogramowania. Jednak pewne meta-analizy zdają się wskazywać, że wpływ TDD na różne aspekty procesu wytwórczego jest niejednoznaczny. W tabeli 8.1 przedstawiono (za [1]) przegląd wyników eksperymentów pochodzących z różnych prac i dotyczących wpływu TDD na różnorakie parametry procesu. Znak '<' oznacza wykazanie negatywnego wpływu TDD na dany parametr (np. produktywność jest niższa w grupie stosującej TDD niż w grupie nie stosującej tego podejścia). Znak '>' oznacza efekt pozytywny. Znak 'o' oznacza, że nie wykazano istotnej różnicy pomiędzy grupami. Rozważane w niniejszej analizie typy eksperymentów to: eksperyment kontrolowany (ang. controlled experiment, CE), eksperyment quasi-kontrolowany (ang. quasi-controlled experiment, QCE) oraz studium przypadku (ang. case study, CS).

Tabela 8.1: Przegląd postrzeganych efektów TDD na wybrane aspekty procesu wytwarzania oprogramowania (za [1]))

Publikacja	Typ studium	PR	ZJ	Z	PK	MSI
Madeyski [72]	CE	<	<		>	>
George i Williams [73]	QCE	<	>			
Bhat i Nagappan [74]	CS	<			>	
Erdogmus [75]	CE	>	o			
Flohr i Schneider [76]	QCE	>			<	
Gupta i Jalote [77]	CE	>	>			
Huang i Holcombe [78]	CE	>	o			
Janzen i Saedian [79]	QCE, CS			>		
Janzen i Saedian [80]	CE		>			
Crispin [81]	CS		>			
Geras et al. [82]	QCE		o			
Pančur et al. [83]	CE		<		<	
Siniaalto [84]	CS				>	
Mueller i Hagner [85]	QCE				<	
Madeyski [86]	CE		<			

Prod - produktywność, ZJ - zewnętrzna jakość, Z - złożoność, PK - pokrycie kodu

Cztery spośród siedmiu opisanych studiów pokazały pozytywny wpływ TDD na produktywność, ale trzy pozostałe wykazały wpływ negatywny. Wobec chyba najbardziej interesującego parametru – zewnętrznej jakości (ang. external quality) 4 (na 10) eksperymenty wykazały efekt pozytywny, 3 – negatywny, a 3 – brak efektu. Jedno studium wykazało pozytywny efekt TDD na złożoność

oprogramowania. Trzy spośród 6 analizowanych prac wskazują na pozytywny efekt TDD na pokrycie kodu, trzy inne – negatywny.

W jednej pracy zaraportowano również wyniki dotyczące tzw. wskaźnika MSI (ang. Mutation Score Indicator) zdefiniowanego jako dolna granica ilorazu liczby zabitych mutantów do całkowitej liczby mutantów nierównoważnych. Metryka ta, w niniejszej pracy zwana pokryciem mutacyjnym, służy często jako komplementarna miara pokrycia kodu. Wykorzystuje się ją w celu oceny jakości i mocy zaprojektowanych testów. Wspomniana praca wykazuje pozytywny efekt TDD na tę miarę w porównaniu z techniką 'test-last'.

Wspomnieliśmy już wcześniej, że TDD jest bardzo popularną dziedziną badawczą. Obecnie indeksy bibliometryczne wskazują istnienie kilku tysięcy praz naukowych związanych z metodyką TDD[2]. Zazwyczaj jednak źródła te pomijają chyba najistotniejszą zaletę stosowania tej praktyki. W teorii testowania oprogramowania istnieje pojęcie „psychologii testowania" i związanego z nią problemu tzw. niezależności testowania [4]. Chodzi o to, że programista, który napisał *własny* kod, ma do tego kodu emocjonalny stosunek i posiada w związku z tym pewne uprzedzenia (ang. bias). Jeśli *po* napisaniu kodu programista zostanie poproszony o dopisanie do niego testów jednostkowych, to ze względu na brak owej niezależności (programista testuje bowiem swój własny produkt) jego testy będą zazwyczaj słabe, gdyż będą jedynie starały się potwierdzić założenia, jakie programista poczynił przed rozpoczęciem implementacji.

Podejście TDD pozwala programiście uzyskać nieco większą niezależność — jeśli testy ma napisać *zanim* zaczął tworzyć kod, może przemyśleć to, jak w najlepszy sposób weryfikować założenia specyfikacji wymagań, które ma zaimplementować. Dzięki temu napisany *później* kod będzie lepszej jakości. Z powyższych rozważań wynika, że podejście TDD może w istotny sposób wpłynąć na poprawę jakości końcowej tworzonego oprogramowania. Warto zatem przeprowadzić ten proces w sposób możliwie wydajny pod kątem zapewnienia i kontroli jakości. Obserwację tę wspierają np. wyniki prac [87, 88]. W kolejnych podrozdziałach opiszemy podejście wykorzystujące testowanie mutacyjne, które modyfikuje proces TDD, pozwalając na efektywniejsze jego użycie. Następnie omówimy wyniki eksperymentów, które przeprowadziliśmy z użyciem tej zmodyfikowanej metodyki, pokazując, że rzeczywiście może ona dawać lepsze wyniki niż stosowanie "czystego" podejścia TDD.

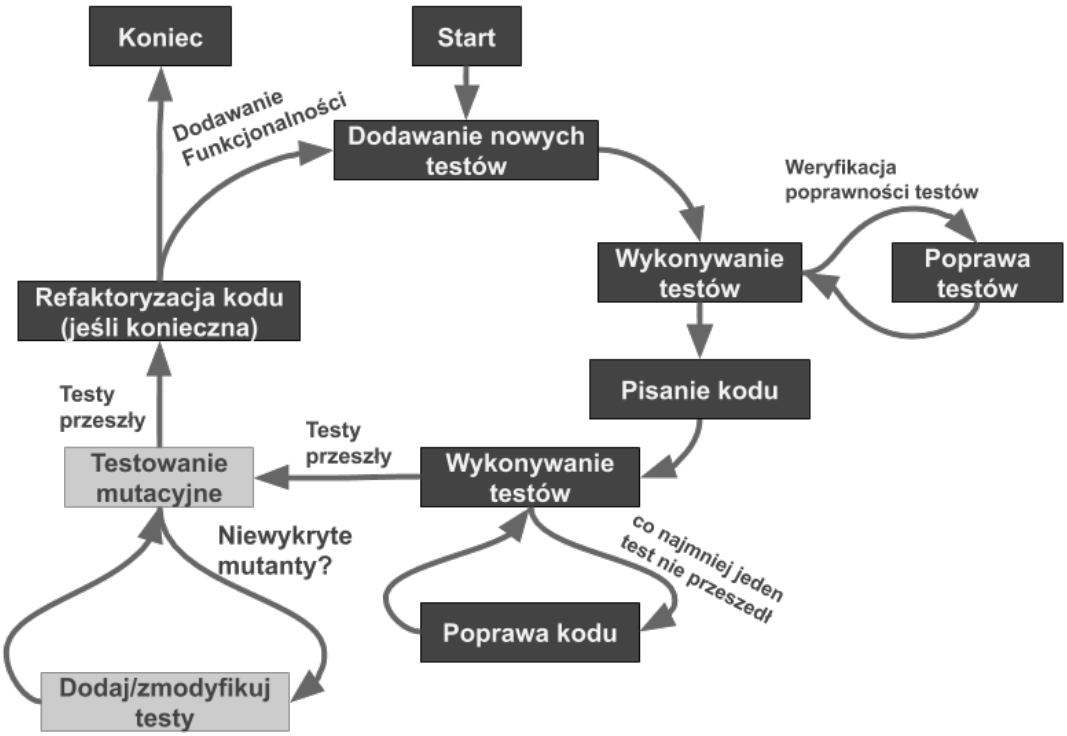
8.2 Test-Driven Development + Mutacje

Udoskonaleniem metodyki TDD może być metodyka TDD+M polegająca na dodaniu dodatkowego kroku procesu dotyczącego przeprowadzenia testów mutacyjnych przed etapem refaktoryzowania kodu. W ten sposób można znacząco podnieść jakość wytwarzanego oprogramowania, a także jego bezpieczeństwo[3]. Diagram z rys. 8.2 opisuje sposób użycia metodyki TDD+M.

Dodatkowy krok – testowanie mutacyjne – znajduje się pomiędzy wykonaniem testów a

[2]Np. w bazach danych Springra słowo kluczowe „Test-Driven Development" zwraca ok. 4500 prac

[3]Kluczową sprawą podczas podnoszenia metodyki z poziomu TDD do TDD+M jest pamiętanie, że test mutacyjny jest weryfikacją poprawności i siły testu, ale może także weryfikować poprawność implementacji metod, których test dotyczy.



Rysunek 8.2: Metodyka Test-Driven Development + Mutatacje

refaktoryzacją kodu. Po zaliczeniu wszystkich testów przeprowadzany jest proces testowania mutacyjnego. W jego trakcie może się okazać, że chociaż wszystkie testy przechodzą, to nie są zbyt silne. Jeśli żaden z nich nie jest w stanie zabić niektórych z wygenerowanych mutantów, programista jest niejako zmuszony dodać nowy test (lub przynajmniej zmodyfikować jakiś już istniejący) tak, aby ostatecznie wszystkie mutanty zostały zabite. Tylko po osiągnięciu tego kroku można rozpocząć refaktoryzację kodu oraz kolejną iterację procesu.

8.3 Porównanie TDD z TDD+M

W celu weryfikacji tez związanych z testowaniem mutacyjnym oraz dołączania komponentu mutacyjnego do technik zwinnych takich jak TDD zostały przeprowadzone dwa eksperymenty na grupach deweloperów programujących na poziomie podstawowym i średnio-zaawansowanym.

W obu eksperymentach użyto następujących operatorów mutacyjnych.

- ReturnValsMutator – modyfikuje zwracaną wartość (dla zmiennej bool zamienia TRUE na FALSE; dla zmiennych typu integer, byte oraz short zamienia 1 na 0 i 0 na wartość inną niż 0; for long zamienia x na $x + 1$; dla zmiennej typu float zastępuje x na $-(x + 1, 0)$, jeśli x nie jest NAN (Not a number) to zamienia NAN na 0; dla obiektu zamienia niezerowe zwracane wartości na null, a następnie zgłasza wyjątek java.lang.RuntimeException, jeśli niezmutowana metoda zwróci wartość null;
- IncrementsMutator – zamienia inkrementacje na dekrementacje oraz na odwrót. Przykładowo `i++` zostanie zamienione na `i-`;

- MathMutator – zamienia arytmetyczne binarne operatory (int oraz float) między sobą;
- NegateConditionalsMutator – zamienia operator relacyjny na jego negację: `==` na `!=`, `<=` na `>`, `>` na `<=` itd.;
- InvertNegsMutator – „odwraca" liczby całkowite oraz zmiennoprzecinkowe, przykładowo: `i = j+1` zostanie zamienione na `i = -j+1`;
- ConditionalsBoundaryMutator – zamienia ostre domknięcia warunków na nieostre i na odwrót, tzn. `<` na `<=`, `>=` na `>`;
- VoidMethodCallMutator – usuwa wywołanie metod nie zwracających wartości (metody typu void).

8.3.1 Cel eksperymentów

Celem badań było wykazanie poprawy jakości wytwarzanego oprogramowania w technikach zwinnych opartych na TDD poprzez modyfikację tej metodyki do TDD+M. Wedle tego założenia programiści pracujący w metodyce TDD+M uzyskają mniejszą liczbę nieprzetestowanych przepływów sterowania oraz pokryją znacznie większą część implementowanego przez siebie kodu.

Celem eksperymentów jest zweryfikowanie Tezy 3 (patrz podrozdz. 1.2), którą w niniejszym rozdziale podzielimy na trzy podtezy:

- Teza 3.1: testy napisane w podejściu TDD+M dają lepsze pokrycie kodu niż te pisane w czystej metodyce TDD, bez udziału mutacji.
- Teza 3.2: testy napisane w podejściu TDD+M są silniejsze niż te pisane w czystej metodyce TDD.
- Teza 3.3: jakość zewnętrzna oprogramowania jest wyższa, gdy zamiast TDD stosuje się podejście TDD+M.

Jak zauważyliśmy wcześniej (patrz podrozdz. 8.1), badania nad efektywnością TDD nie są konkluzywne. Dlatego, aby porównać TDD z TDD+M użyjemy najprostszych, podstawowych metryk, aby uniknąć nadmiernego skomplikowania metodologicznego w naszym eksperymencie. W celu weryfikacji tezy 3.1 wykorzystamy metrykę pokrycia instrukcyjnego (patrz def. 2.5. Do zbadania tezy 3.2 posłużymy się metryką pokrycia mutacyjnego (patrz def. 3.1. W celu weryfikacji tezy 3.3 użyjemy z kolei metryki liczby defektów znalezionych w kodzie podczas fazy testów. W ten sposób model dla porównania obu podejść będzie możliwie prosty, przez co unikniemy różnego rodzaju niebezpieczeństw i stronniczości (ang. bias), które mogłyby wyniknąć ze zbyt dużej złożoności modelu.

8.4 Eksperyment pierwszy

Do eksperymentu pierwszego została wybrana ośmioosobowa grupa programistów będących studentami trzeciego roku studiów informatycznych na specjalności „inżynieria oprogramowania" na

Wydziale Matematyki i Informatyki na Uniwersytecie Jagiellońskim w Krakowie. Przeprowadzenie eksperymentu zajęło około jednego miesiąca. Celem eksperymentu było wstępne potwierdzenie skuteczności metodyki TDD+M. Ze względu na mały rozmiar grupy uczestniczącej w badaniu, jego wyniki nie mogą być weryfikowane formalnymi, statystycznymi metodami.

Programiści zostali podzieleni na dwie czteroosobowe grupy. Pierwsza grupa pracowała w czystej metodyce TDD. W jej skład wchodziło dwóch programistów, jeden tester i szef zespołu nadzorujący proces oraz projektujący oprogramowanie.

Druga grupa pracowała w metodyce TDD+M w podobnym składzie, czyli dwóch programistów, jeden tester (wykonujący także testy mutacyjne) oraz szef zespołu, który tak jak w grupie pierwszej nadzorował procesy wytwórcze, a także testy mutacyjne. Ponadto, szef zespołu wraz z testerem przeprowadzał przeglądy kodu w oparciu o wyniki testów mutacyjnych.

Obie grupy pracowały nad tym samym projektem, który polegał na wytworzeniu oprogramowania realizującego proste operacje macierzowe, takie jak:

- dodawanie dwóch macierzy ($M_1 + M_2$);
- odejmowanie macierzy ($M_1 - M_2$);
- transponowanie macierzy M^T;
- obliczanie macierzy odwrotnej M^{-1}.

Wszystkie macierze miały wymiar maksymalnie 3×3. Założenia projektu były opisane w dostarczonym pliku typu JavaDoc. Plik zawierał szkielet aplikacji w postaci deklaracji wszystkich interfejsów, których implementacje miały być następnie wykonane przez zespoły programistyczne. Zespoły nie mogły modyfikować interfejsów ani dopisywać nowych tak, ażeby testy wykonywane w grupie TDD mogły być przeniesione do grupy TDD+M i na odwrót.

Grupa TDD wykonała 8 pełnych iteracji, natomiast grupa TDD+M wykonała 9 pełnych iteracji. Obie grupy pracowały zgodnie z wyznaczonymi im metodykami.

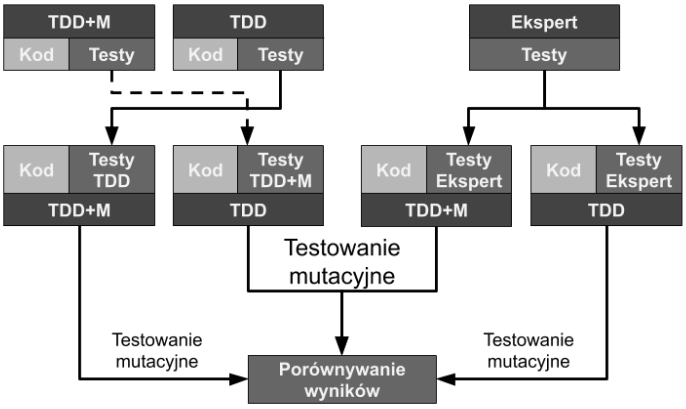
Aby umożliwić głębszą analizę wyników dodany został dodatkowy czynnik polegający na zleceniu zaprojektowania testów doświadczonemu programiście z ponad pięcioletnim stażem pracy. Następnie jego testy zostały użyte na kodzie obu grup w celu weryfikacji jakości wytworzonego przez nie oprogramowania, a także w celu zweryfikowania, czy któraś z grup studenckich, złożonych z niedoświadczonych programistów, była w stanie osiągnąć porównywalne lub lepsze rezultaty programisty zawodowego. Testy były wykonane w ostatniej iteracji TDD+M.

8.4.1 Wyniki eksperymentu pierwszego

Po zakończeniu eksperymentu przez obie grupy przystąpiono do opracowania wyników. Dla grupy TDD+M zostały zebrane i opracowane wyniki z testów mutacyjnych dla wszystkich dziewięciu iteracji. Następnie testy tej grupy podmieniono na testy grupy TDD oraz została wykonana dodatkowa sesja testów mutacyjnych na ostatniej iteracji kodu TDD+M z testami z grupy TDD

w celu późniejszego porównania wyników. Ten projekt został także przetestowany mutacyjnie przy użyciu testów napisanych przez doświadczonego programistę.

Dla grupy TDD została wykonana sesja testów mutacyjnych dla każdej z dostarczonych iteracji[4]. Podobnie jak w poprzedniej grupie przeprowadzono dodatkową sesję mutacyjną przy użyciu implementacji programu w metodyce TDD oraz testów z metodyki TDD+M. Przebieg eksperymentu schematycznie obrazuje diagram z rys. 8.3.



Rysunek 8.3: Układ eksperymentalny w eksperymencie pierwszym

8.4.2 Wyniki

Tabela 8.2: Statystyki kodu i testów dla obu grup w eksperymencie pierwszym

Grupa	Projekt LOC	Testy LOC	Kod LOC	test-to-code-ratio
TDD	216	214	430	0.99
TDD+M	321	536	857	1.67

Oba projekty składają się z dwóch interfejsów oraz dwóch klas je implementujących. Statystyki dotyczące kodu programu dla pierwszej i drugiej grupy znajdują się w tabeli 8.2. Projekt pisany zgodnie z podejściem TDD+M zawiera 321 linii kodu programu bez testów oraz 857 z testami. Projekt pisany zgodnie z TDD zawiera 216 linii kodu programu bez testów oraz 430 z testami. W grupie TDD ilość kodu testów jest porównywalna z ilością kodu właściwego – ich stosunek wynosi 0.99. W grupie TDD+M proporcja kodu testowego jest zauważalnie wyższa (1.67), co nie powinno dziwić – testowanie mutacyjne wymusza bowiem na programiście dodawanie kodu testowego wykrywającego niezabite dotychczas mutanty. W czystym podejściu TDD mechanizm ten nie występuje, dlatego kod testowy jest o wiele uboższy.

[4]Grupa TDD nie stosowała mutacji, więc w celu weryfikacji testów została ona przeprowadzona post-factum.

Tabela 8.3: Wyniki dla grupy TDD

wersja	Pl %	Pm %	testy	mutanty	metody
1	0	0	0	4	6
2	79,31	39,13	2	23	8
3	79,49	82,69	5	52	10
4	79,49	82,69	5	52	10
5	89,74	82,69	7	52	10
6	66,67	63,24	8	68	16
7	41,86	41,35	8	104	16
8	**60,47**	**64,42**	11	104	16
Testy z grupy TDD+M	**86,76**	**75,96**	16	104	16
Testy eksperta	**91,42**	**82,32**	18	104	16

Tabela 8.3 przedstawia wyniki testów mutacyjnych na grupie TDD, niestosującej metodyki TDD+M. *Pl* oznacza pokrycie instrukcyjne, *Pm* – pokrycie mutacyjne. Kluczowe metryki do porównania zaznaczono pogrubioną czcionką. Testy z projektu wykonanego zgodnie z TDD+M zabiły 79 ze 104 mutantów, co daje 75.96% pokrycia mutacyjnego. Jest to o 12 mutantów więcej oraz o 11 punktów procentowych więcej w stosunku do wyniku grupy TDD osiągniętego ich własnymi testami. Pokrycie kodu testami grupy TDD+M jest z kolei o ponad 26 punktów procentowych większe. Testy grupy TDD+M osiągnęły niewiele niższe pokrycie niż testy eksperta, ale wyższe od testów własnych grupy TDD.

Zmniejszenie pokrycia pomiędzy iteracjami 5 i 6 było wynikiem zwiększenia liczby zaimplementowanych metod. Dodano 6 nowych metod i tylko jeden nowy test. Z kolei zmniejszenie pokrycia pomiędzy iteracjami 6 i 7 wynikało z faktu wprowadzenia nowego kodu (i – jako skutek tego działania – nowych mutantów), przy jednoczesnym braku zamian w liczbie testów. Wzrost pokrycia pomiędzy dwoma ostatnimi iteracjami wynika z dodania pięciu nowych testów.

Tabela 8.4: Wyniki dla grupy TDD+M

Wersja	Pl %	Pm %	Testy	Mutanty	Metody
1	43,90	30,30	3	33	16
2	36,11	34,33	4	134	16
3	39,81	38,06	6	134	16
4	39,81	39,55	6	134	16
5	39,81	38,81	6	134	16
6	47,22	44,03	7	134	16
7	49,07	44,03	8	134	16
8	93,52	87,31	12	134	16
9	**81,45**	**73,58**	12	134	16
Testy z grupy TDD	**39,52**	**40,25**	12	134	16
Testy eksperta	**90,32**	**82,39**	18	134	16

Tabela 8.4 prezentuje z kolei wyniki przeprowadzenia testów mutacyjnych w grupie stosującej metodykę TDD+M. Oznaczania są takie same jak w tabeli 8.3. W tym przypadku metody

implementujące interfejsy były bardziej obszerne, co sprawiło, że testy z grupy TDD pokryły je tylko w 39% oraz wykryły tylko 40% mutantów. To o 47.2% (odp. 35.7%) mniej niż w przypadku testów grupy TDD+M pokrywających kod grupy TDD (patrz tabela 8.3).

Grupa TDD+M była w stanie osiągnąć – na swoim własnym kodzie – 81.4% (odp. 73.5%) pokrycia instrukcyjnego (odp. mutacyjnego). To o 21% (odp. 13.1%) więcej niż w przypadku testów grupy TDD na swoim własnym kodzie, które osiągnęły odpowiednio 60.4 i 64.4%. O ile można było się spodziewać, że w układzie krzyżowym (testy jednej grupy na kodzie drugiej) grupa TDD+M osiągnie znacznie lepsze rezultaty, gdyż stosowała wprost podejście mutacyjne, o tyle w przypadku testów danej grupy na własnym kodzie powyższe wyniki są zdecydowanie bardziej interesujące. Wskazują bowiem na pozytywny wpływ technik mutacyjnych na jakość testów.

Testy wykonane przez eksperta miały podobną skuteczność co w przypadku kodu grupy TDD: wykryły 82% mutantów oraz pokryły kod w 90%. Osiągnęły one wyniki nieco lepsze niż testy własne grupy TDD+M. Mimo wszystko, ekspert nie był w stanie przewidzieć wszystkich przypadków nawet w tak prostym programie. Spadek pokrycia pomiędzy dwoma ostatnimi iteracjami wynikał z dodania nowej porcji kodu bez dodania jakichkolwiek nowych testów.

Godnym uwagi jest również fakt, że liczba mutantów była większa w przypadku grupy TDD+M (134 wobec 104). Oznacza to, że kod tej grupy był większy, bardziej skomplikowany, a co za tym idzie – trudniejszy do pokrycia w terminach zarówno pokrycia instrukcyjnego jak i mutacyjnego. Mimo to, grupa TDD+M była w stanie osiągnąć lepsze wyniki w terminach tych metryk niż grupa TDD.

W eksperymencie pierwszym grupa TDD+M osiągnęła o wiele lepsze wyniki niż grupa TDD, zarówno w terminach jakości własnych testów stosowanych do własnego kodu, jak i w terminach mocy własnych testów zastosowanych do cudzego kodu. Najlepsze wyniki osiągnął ekspert, choć były one tylko nieznacznie lepsze od wyników grupy TDD+M. Zauważmy ponadto, że nawet ekspert nie był w stanie osiągnąć pełnego pokrycia instrukcyjnego w tym relatywnie prostym kodzie. Niektóre mutanty przeżyły. Były to mutanty odpowiedzialne za zmiany w mnożeniu macierzy. Powodem było to, że w niektórych przypadkach wyniki działania kodu były poprawne nawet dla złej implementacji formuły mnożenia (np. $2 \cdot 2 = (-2) \cdot (-2)$, zatem jeśli testy uruchomiono na kodzie mnożącym dwie macierze 1×1 równe [2] i operator mutacyjny zmienił mnożenie na dodawanie, test dawał wynik fałszywie pozytywny).

8.4.3 Szczegółowe informacje o testach i znalezionych defektach

Dla obu grup została przeprowadzona analiza wykonanych testów. Raport został podzielony względem testowanych funkcjonalności oprogramowania. Każda funkcjonalność została opatrzona opisem sposobu w jaki poprawność jej działania jest sprawdzana przez testy.

Dla grupy niemutującej (TDD) analiza testów wygląda sposób następujący.

- Proste dodawanie macierzy: Do testów zostały użyte tylko dwie macierze. Test polega na wykonaniu dodawania na powyższych dwóch macierzach oraz sprawdzeniu czy wynik jest

zgodny z oczekiwanym.

- Mnożenie macierzy: test nie został wypełniony.
- Odejmowanie macierzy: Proces testowanie został wykonany przy użyciu tych samych macierzy co w przypadku z dodawaniem. Mechanika testu jest analogiczna jak w przypadku dodawania.
- Transpozycja: Brak wyplenionego testu.
- Mnożenie przez skalar: Brak wyplenionego testu.
- Macierz identycznościowa: Do testów użyto jednego przykładu macierzy wraz z weryfikacją, czy wpisywane wartości do macierzy są takie jak oczekiwano.
- Wyznacznik: Wyznacznik przetestowany dla trzech przypadków macierzy 2x2.
- Wstawianie wartości do macierzy: Napisano dwa testy, które sprawdzają czy macierze mają poprawną wartość. Sprawdzanie odbywa się dla łącznie 4 przypadków.

Analiza testów grupy mutującej (TDD+M) wykazała znacznie większą liczbę testów. Testy w grupie niemutującej znajdowały się w dwóch klasach, natomiast w grupie mutującej już w czterech. Dla grup mutujących analiza testów wygląda w sposób następujący.

- Dodawanie macierzy: Badane przez dwa testy. pierwszy na macierzach 3x3, Drugi weryfikujący poprawność testów za pomocą trzech asercji sprawdzających poprawność dodawania siedmiu macierzy w różnych kombinacjach.
- Mnożenie macierzy: Testowanie zostało wykonane tak samo jak w przypadku dodawania, ale na innym zestawie macierzy.
- Odejmowanie macierzy: Tak samo jak w przypadku dodawania i mnożenia lecz na innych zestawach danych.
- Transpozycja: Jeden test sprawdzający poprawność transpozycji macierzy 3x3.
- Odwracanie macierzy: W celu sprawdzenia tej funkcjonalności użyto dwóch testów: pierwszy wykorzystuje macierz 3x3, drugi odwraca dwie inne macierze 3x3 i sprawdza ich poprawność.
- Macierz identycznościowa: Test opracowany na podstawie 2 innych testów związanych z testowaniem poprawności wymiarów macierzy. Używał macierzy 2x2 oraz macierzy niepoprawnej.
- Wyznacznik: testowany przez cztery macierze (3x3 , 2x2 ,2x2, 2x1). Napisano tylko jeden test, ale z czterema asercjami.
- Wstawianie wartości do macierzy: Sprawdzanie poprawności wpisywanych danych odbywało się w jednym teście z pętlą w środku dla siedmiu macierzy.

Grupa przygotowała sobie dwa zestawy po siedem macierzy do testów oraz w paru testach generowała dodatkowe macierze na bieżąco. Macierze były używane w testach w różnych kombinacjach.

Grupa niemutująca za pomocą swoich testów znalazła następujące defekty:

- Problem w liczeniu wyznacznika.
- Brak odporności metody analizy macierzy na wartości null.
- Defekt w zaokrąglaniu wartości (ponieważ zamiast zmiennych zmiennoprzecinkowych użyto zmiennych całkowitych).

W grupie niemutującej dwa razy pojawił się defekt w testach. W obu przypadkach został on zauważony. W grupie mutującej taka sytuacja nie miała miejsca. Prawdopodobnie wynika to z samej istoty procesu testowania mutacyjnego, który z zasady eliminuje defekty w testach.

Grupa mutująca za pomocą swoich testów znalazła następujące defekty.

- W pierwszej iteracji po testach naprawione zostało rzucanie wyjątków. Podczas procesu testowania i mutacji skupiono się na wykryciu i eliminacji wszystkich mutantów.
- W trzeciej iteracji poprawione zostało wypisywanie wypisywanie wartości na konsolę. W tej iteracji została również naprawiona kolejność nadpisywania wartości w obiektach macierzy. Wszystkie poprawki zostały wykonane po testach mutacyjnych.
- W kolejnych iteracjach skupiono się na zmniejszeniu liczby żywych mutantów. Podczas tego procesu został wykryty i naprawiony problem w metodzie obliczania macierzy identycznościowej.

Grupa na przestrzeni 58 commitów skupiła się na detekcji i eliminacji żywych mutantów. Problem związany z wypisywaniem niewłaściwych danych na konsolę został wykryty bez użycia testów mutacyjnych. Po wykryciu wszystkich mutantów w ostatniej iteracji, w związku z brakiem dalszych defektów oraz wysokim pokryciem mutacyjnym zadecydowano o zakończeniu prac nad programem.

8.4.4 Wnioski z pierwszego eksperymentu

Podczas eksperymentu pierwszego grupa mutacyjna wypadła znacznie lepiej niż grupa stosująca zwykłą technikę TDD. Testy grupy TDD+M w dla własnego kodu były o 33% silniejsze niż testy TDD dla swojego kodu. Natomiast w grupie TDD testy z TDD+M okazały się o 11% silniejsze. Najlepszy jednak okazał się doświadczony programista niestosujący TDD+M. Jednak i w tym wypadku testy nie okazały się w 100% skuteczne. Cześć mutantów we fragmentach kodu odpowiadających mnożeniu macierzy przeżywała. Okazało się, że testy mutacyjne doskonale nadają się do weryfikacji poprawności testów metod numerycznych. Kilka uwag w tej kwestii opisano w kolejnym punkcie.

8.4.5 Eksperyment pierwszy – dodatkowe uwagi

Testowanie mutacyjne daje całkiem dobre wyniki w przypadku testowania bibliotek numerycznych. Powinno więc stanowić wystarczające środki do przetestowania poprawności implementacji różnych operacji numerycznych, w szczególności związanych z operacjami na wielomianach. Jest to zagadnienie kluczowe z punktu widzenia obliczeń komputerowych, ze względu na rolę wielomianów w algebrze liniowej, metodach numerycznych czy interpolacji.

Aby lepiej zobrazować powyższą tezę, rozważmy prosty przykład. Załóżmy, że poprawną implementacją pewnego wielomianu jest $P(x) = ax^3 + bx^2 + cx + d$. Jeśli rozważymy następujące trzy mutacje:

1. $P_1(x) = ax^3 \times bx^2 + cx + d$,
2. $P_2(x) = ax^3 + bx^2 \times cx + d$,
3. $P_3(x) = ax^3 - bx^2 + cx + d$,

jedynym przypadkiem, w którym wszystkie te trzy wielomiany są równe jest przypadek $a = b = c = 0$, co jest mało prawdopodobne w rzeczywistym wykonaniu programu. Zatem prawdopodobieństwo wygenerowania mutantów równoważnych jest w tym przypadku niskie.

Ponadto, jeśli mamy przypadek testowy weryfikujący poprawność implementacji takiego wielomianu, okazuje się, że wystarczy bardzo niewielka liczba mutantów, aby wykryć nieprawidłową implementację, *nawet jeśli test na oryginalnym kodzie daje wynik fałszywie pozytywny.*

Na przykład, załóżmy, że poprawną implementacją wielomianu jest $P(x) = x^2 - x + 2$, ale – ze względu na popełnioną przez programistę pomyłkę – implementacja rzeczywista to $P_{wrong}(x) = x^2 + x - 2$. Załóżmy ponadto, że mamy przypadek testowy t dla $x = 2$, dla którego oczekiwany wynik to $P(2) = 4$. Zauważmy, że w rzeczywistej, niepoprawnej implementacji również zachodzi $P_{wrong}(2) = P(2) = 4$, co oznacza, że test jest zdany i jest to wynik fałszywie pozytywny. Nasz test nie wykryje więc defektu w kodzie.

Jednak zazwyczaj wystarczy wygenerować tylko jednego mutanta zmieniającego operator arytmetyczny, aby wykryć tego typu defekt. W tabeli 8.5 zebrano kilka typów mutacji (wygenerowanych przy użyciu dwóch różnych operatorów mutacyjnych) dla $P(x)$ wygenerowanych przy użyciu różnych operatorów mutacyjnych. Zauważmy, że dla każdego z nich test wykrywa różnicę pomiędzy implementacją oryginału i mutanta, chociaż test ten daje wynik fałszywie pozytywny na oryginalnej, niepoprawnej implementacji.

8.5 Eksperyment drugi

Drugi eksperyment został przeprowadzony w podobny sposób jak pierwszy, tylko na większą skalę. W badaniu wzięły udział 22 osoby (również studenci informatyki, jak w eksperymencie pierwszym), które zostały podzielone na 8 grup. 4 grupy pracowały w metodyce TDD+M, a 4 grupy w metodyce TDD. Rozkład osób w grupach obrazuje tabela 8.6.

Tabela 8.5: Mutacje wielomianu $P_{wrong}(x) = x^2 + x - 2$ z $P(2) = 4$

Operator	Zmutowany P_{wrong}	$P_{wrong}(2)$
Arithmetic Operator Replacement	$x^2 - x - 2$	0
Arithmetic Operator Replacement	$x^2 \cdot x - 2$	6
Arithmetic Operator Replacement	$x^2/x - 2$	0
Arithmetic Operator Replacement	$x^2 + x + 2$	8
Arithmetic Operator Replacement	$x^2 + x \cdot 2$	8
Arithmetic Operator Replacement	$x^2 + x/2$	5
Unary Operator Insertion	$x^{-2} + x - 2$	0.25
Unary Operator Insertion	$-x^2 + x - 2$	-4
Unary Operator Insertion	$-(x^2 + x - 2)$	-4

Program przeznaczony do implementacji był rozwinięciem programu z pierwszego eksperymentu. Programiści biorący udział w eksperymencie implementowali bibliotekę operacji macierzowych użytą także w poprzednim eksperymencie, bibliotekę do prostych obliczeń geometrycznych, interfejs webowy do obu bibliotek oraz serwer obsługujący żądania http, na podstawie których dostarczany był interfejs webowy do użytkownika. Eksperyment trwał 3 tygodnie, w trakcie których grupy implementowały program oraz tworzyły testy podczas krótkich iteracji.

Po zakończeniu kodowania powstałe programy przygotowano do eksperymentu krzyżowego, analogicznego do tego w eksperymencie pierwszym. W tym celu stworzono 64 kopii oprogramowania dla każdego wariantu par: (testy z grupy X, program z grupy Y), dla $X, Y \in \{1, ..., 8\}$.

Osoby biorące udział w eksperymencie, przed jego rozpoczęciem, wypowiedziały się w ankiecie na temat swoich zdolności programistycznych oraz testerskich. Ankieta zawierała dwa pytania:

1. Jak dobrze oceniasz swoje zdolności programistyczne?

2. Jak dobrze oceniasz swoje zdolności testerskie?

Obie odpowiedzi należało wyrazić w pięciostopniowej skali. Średnią ocenę[5] z ankiety dla każdej z grup można obserwować w tabeli 8.6. Ze względu na małą liczebność grup tabela podaje również wprost odpowiedzi uczestników.

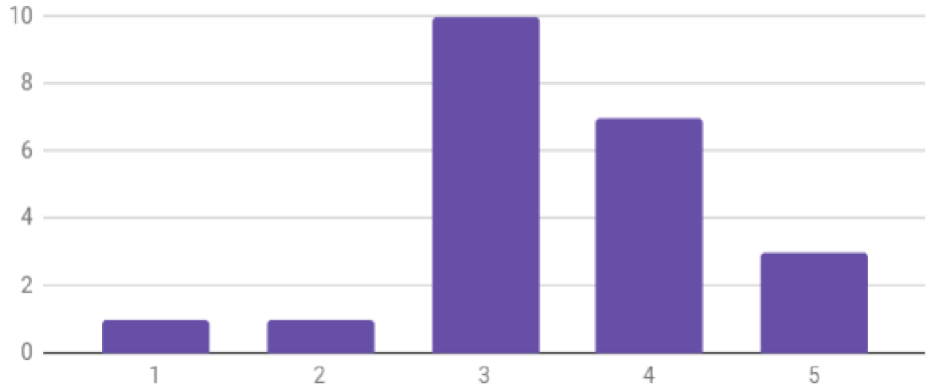
[5]Należy wyraźnie zaznaczyć, że wykonywanie operacji uśredniania na wartościach pochodzących ze skali porządkowej jest pewnym nadużyciem metodologicznym, z którego zdajemy sobie sprawę. Dlatego w wynikach podajemy również wprost odpowiedzi uczestników.

Tabela 8.6: Dane zbiorcze grup

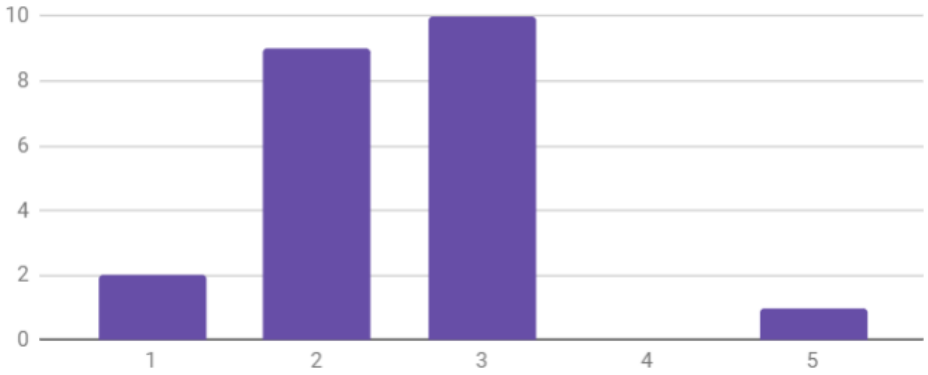
Id Grupy	Typ Grupy	Liczba osób	Zdolności programistyczne	Zdolności testerskie
Grupa 01	mutująca	3	4 (4, 4, 4)	2,3 (2, 2, 3)
Grupa 02	mutująca	3	3,3 (4, 5, 1)	2,3 (3, 3, 1)
Grupa 03	mutująca	3	3 (3, 3, 3)	1,7 (2, 2, 1)
Grupa 04	mutująca	3	4,3 (5, 5, 3)	3,3 (3, 2, 5)
Grupa 05	niemutująca	3	3,3 (4, 3, 3)	2,7 (2, 3, 3)
Grupa 06	niemutująca	3	3,3 (3, 4, 3)	2,7 (3, 2, 3)
Grupa 07	niemutująca	2	2,5 (3, 2)	2,5 (3, 2)
Grupa 08	niemutująca	2	3,5 (3, 4)	2,5 (2, 3)

Rozkład odpowiedzi uczestników eksperymentu dotyczących samooceny w zakresie programowania i testowania przedstawiają wykresy na rys. 8.4 oraz 8.5.

Rysunek 8.4: Odpowiedzi indywidualne — ocena zdolności programistycznych

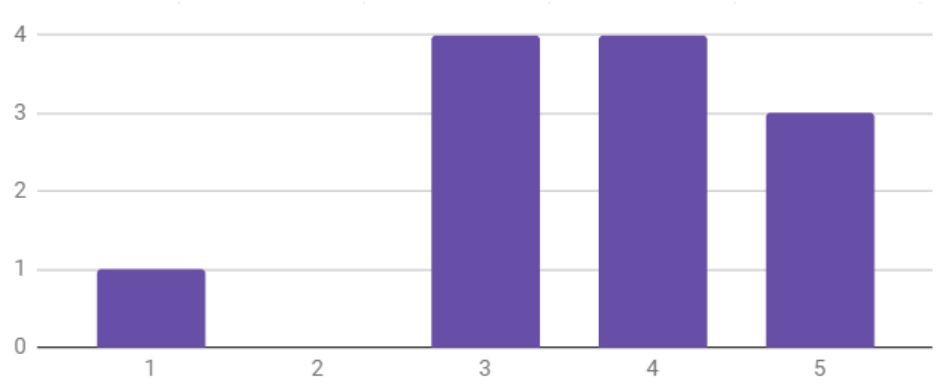


Rysunek 8.5: Odpowiedzi indywidualne — ocena zdolności testerskich

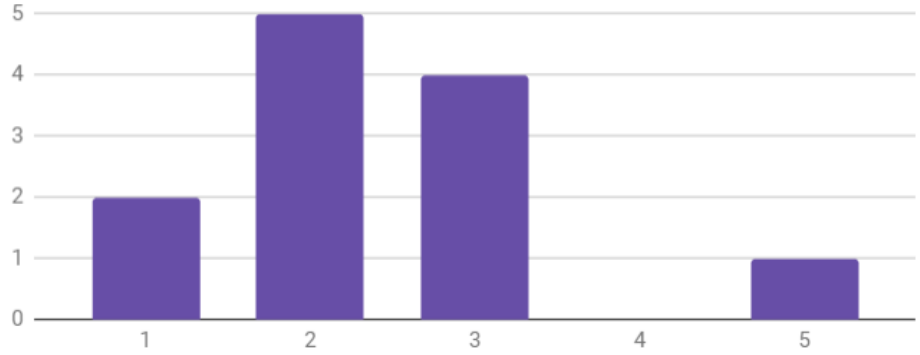


Wyniki ankiet dla uczestników należących do grup stosujących podejście TDD+M prezentują wykresy na rys. 8.6 oraz 8.7.

Rysunek 8.6: Odpowiedzi indywidualne grupy mutującej — Ocena zdolności programistycznych

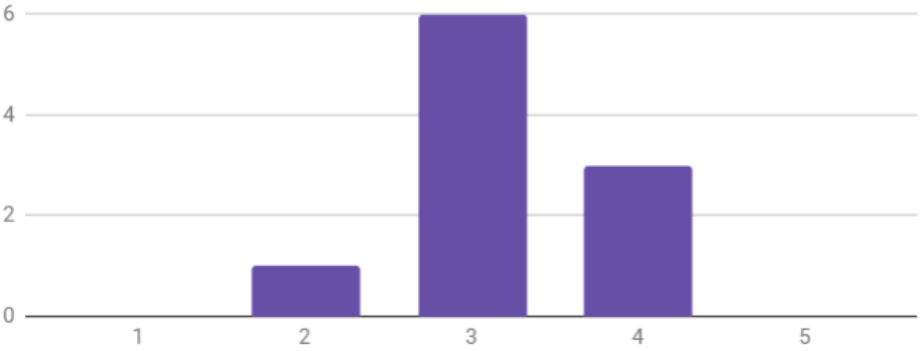


Rysunek 8.7: Odpowiedzi indywidualne grupy mutującej — Ocena zdolności testerskich

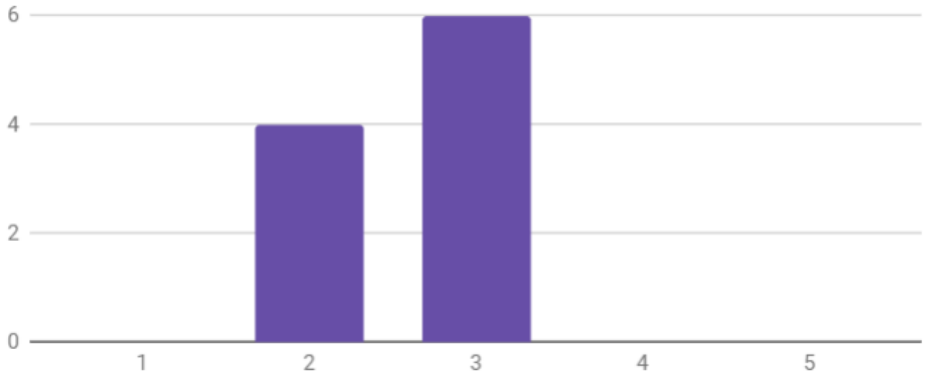


Wyniki ankiet dla uczestników należących do grup niestosujących podejścia TDD+M prezentują wykresy na rys. 8.8 oraz 8.9.

Rysunek 8.8: Odpowiedzi indywidualne grupy niemutującej — Ocena zdolności programistycznych



Rysunek 8.9: Odpowiedzi indywidualne grupy niemutującej — Ocena zdolności testerskich



Po zakończeniu eksperymentu nastąpiła analiza wyników, przeprowadzona w sposób analogiczny jak w eksperymencie pierwszym. Ze względu na liczbę grup, podczas tego eksperymentu przeprowadzono także statyczną analizę kodu dla każdej z grup. Narzędziem użytym do tego celu była platforma SonarQube w wersji 6.3.

8.5.1 Wyniki

W celu lepszego przybliżenia struktury programów będących przedmiotem eksperymentu poniżej zaprezentowane zostały metryki kodu dla każdej z grup oraz ich sumy i średnie wartości. Symbol „(M)” przy nazwie grupy oznacza, że stosowała ona podejście TDD+M. Symbol kreski oznacza, że grupa nie zaimplementowała danej klasy.

Tabela 8.7: Statystyki dla projektów

Grupa / metryka →	**01 (M)**		**02 (M)**		**03 (M)**		**04 (M)**		**05**		**06**		**07**		**08**	
Plik ↓	CC	LOC	CC	LOC	CC	LOC	CC	LOC	CC	LOC	CC	LOC	CC	LOC	CC	LOC
Bitmap.java	11	35	8	23	5	21	4	15	11	34	4	16	7	25	10	24
Circle.java	7	35	7	36	8	36	10	51	7	38	5	43	8	38	7	35
IGemoetricObj.java	0	5	0	5	0	5	0	5	0	5	0	5	0	5	0	5
IMatrix.java	0	11	0	11	0	14	0	11	0	11	0	11	0	12	0	12
IMatrixMath.java	0	9	0	9	0	9	0	9	0	9	0	9	0	9	0	9
InvalidDimensionException.java	2	9	2	9	2	9	2	9	2	9	2	9	2	9	2	9
Line.java	–	–	–	–	–	–	–	–	–	–	–	–	18	53	–	–
main.java	2	20	2	19	2	20	2	19	2	21	2	19	2	21	2	19
Matrix.java	22	78	30	84	16	66	28	81	32	106	22	84	24	86	18	62
MatrixMath.java	41	117	41	142	14	32	48	127	48	158	38	91	38	121	30	76
MatrixOperations.java	–	–	–	–	–	–	–	–	–	–	–	–	–	–	34	121
MatrixRequestType.java	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	5
MyHttpServer.java	18	194	8	69	11	132	9	165	10	118	11	84	21	218	8	104
PointXY.java	10	22	1	8	2	9	5	12	1	8	2	9	4	22	1	8
QeryType.java	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0	4
Rectangle.java	14	41	3	36	6	46	26	88	12	82	7	48	13	45	10	60
Shape.java	6	39	8	31	0	5	0	5	0	5	0	5	0	5	0	5
ShapeType.java	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0	4
Triangle.java	6	32	3	32	12	55	32	96	10	60	13	63	28	88	8	54
WebContentFabric.java	55	188	60	217	41	138	32	112	25	117	10	63	71	187	56	244
Sum	194	843	173	739	119	605	198	813	160	789	116	567	236	952	186	860
Mean	11.4	49.6	10.2	43.5	7.0	35.6	11.6	47.8	9.4	46.4	6.8	33.4	13.1	52.9	9.8	45.3

Złożoność oraz linie kodu wskazują, że są to programy znacznie większe niż te z eksperymentu pierwszego. Zgodnie z wynikami z tabeli 8.7 największy program pod względem linii kodu napisała grupa siódma. Jest to program, którego średnia złożoność cyklomatyczna wynosi 13.1, a dwie klasy: `MatrixMath.java` oraz `Matrix.java` miały złożoność cyklomatyczną na poziomie 38 i 24. Wskazuje to (zgodnie z komentarzem przy def. 5.2), że kod w tych dwóch klasach jest stosunkowo niestabilny, posiada wiele zapętleń, a graf przepływu sterowania jest bardzo złożony. Dla grupy siódmej można się spodziewać dużej liczby mutantów. Grupa siódma jest zarazem grupą, która nie pracowała w metodyce TDD+M, tylko w zwykłym TDD.

Grupa nr 6, na podstawie metryk cyklometrycznych, jest najlepiej prezentującą się grupą pod względem złożoności oprogramowania. Kod tej grupy jest mało obszerny oraz jego złożoność jest niska. Jednak po wstępnej analizie kodu można stwierdzić, że wynika to z nie dość dokładnej i pobieżnej implementacji zarówno testów, jak i samych klas. Ta grupa także nie używała testów mutacyjnych.

Na podstawie przeglądu kodu i testów najlepszą grupą niestosującą technik mutacyjnych wydaje się być grupa nr 5. Jej średnia złożoność cyklomatyczna wynosi 9.4, a całkowita liczba linii kodu w projekcie to 789. Grupa ta w ankiecie opisanej w tabeli 8.6 nie oceniła się zbyt wysoko[6]. Niepokojącą dla tej grupy jest wysoka wartość złożoności klas `MatrixMath.java` oraz `Matrix.java`. Zgodnie z tabelą 8.7 wynoszą one odpowiednio 48 i 32[7].

Na podstawie ręcznej analizy jakości kodu w grupach mutacyjnych można stwierdzić, że najlepszą grupą jest grupa nr 7 a najgorszą grupa nr 2. Zgodnie z danymi z metryk ta zależność nie jest obserwowalna. Jedynie porównanie metryk dla klasy `Matrix.java` może sugerować, że grupa nr 7 posiada lepszy kod.

Zgodnie z celem eksperymentu oraz metodyk powiązanych z TDD jakość kodu jest tylko skutkiem ubocznym dobrze wykonanych testów, które podnoszą zaufanie do poprawności działania oprogramowania. Toteż nawet przy metrykach wskazujących na dużą złożoność kodu, a co za tym idzie potencjalną jego niestabilność, dobre przetestowanie go może kompensować zagrożenie pojawienia się usterki w skomplikowanym kodzie[8].

Zgodnie z powyższym, poniżej zaprezentowane są wyniki analizy mutacyjnej kodu dla poszczególnych grup. Ponadto, tak jak w eksperymencie pierwszym każdej z grup wymieniono testy, ażeby lepiej i bardziej obiektywnie porównać ich siłę. Testy wymieniono na wszystkie możliwe sposoby, tzn. program każdej grupy był testowany testami wszystkich ośmiu grup. Zatem sumarycznie wykonano $8 \cdot 8 = 64$ cykli testowych.

Dane w tabelach 8.8, 8.9, 8.10 oraz 8.11 są uporządkowane w następujący sposób: W kolumnie „testy” znajduje się oznaczenie informujące, że dane w wierszu należą do konkretnej wersji projektu danej grupy lub do wersji projektu z testami z innej grupy. Przykładowo, w tabeli 8.8 o nazwie „kod z Grupy 2”, wiersz oznaczony przez „gr 2.1” oznacza, że dane w nim umieszczone reprezentują

[6] Praktycznie żadna z grup nie oceniła się wysoko. Osoby poddające się samoocenie często nie są obiektywne.

[7] Z reguły dla tych dwóch klas złożoność cyklomatyczna jest wysoka, bowiem są to pod względem obliczeniowym najbardziej rozbudowane klasy.

[8] Czasem skomplikowanego kodu nie da się uniknąć.

wyniki testów mutacyjnych na pierwszej wersji projektu grupy 2. Oznaczenie „gr 2.2" mówi, że chodzi o dane z drugiej wersji oprogramowania itd. Oznaczenie „gr 2" wskazuje, że wiersz zawiera dane z ostatniej iteracji oprogramowania. Następnie, oznaczenia od „gr 1" do „gr 8" mówią, że dany wiersz zawiera wyniki testów mutacyjnych przeprowadzonych na ostatniej wersji oprogramowania z grupy 2, ale z testami z poszczególnych innych grup. Dane są posortowane względem numerów grup, ale w taki sposób, że na początku zawsze są dane z grupy, do której należy kod programu, poczynając od pierwszej do ostatniej wersji, a dopiero następnie dane reprezentujące wyniki po zamianie testów. Grupy, które podczas procesu wytwarzania oprogramowania nie korzystały z TDD+M, nie posiadają wersji oprogramowania. W tym przypadku mutowana była tylko ostatnia wersja programu. *Pl* oznacza procentowe instrukcyjne, a *Pm* — mutacyjne.

Tabela 8.8: Mutacje dla poszczególnych grup TDD+M (1–2)

(a) kod z Grupy 1

testy	Pl %	Pm%	Testy	Mutanty
gr 1.1	59%	45%	56	343
gr 1.2	59%	48%	56	343
gr 1	67%	70%	54	334
gr 2	17%	21%	15	334
gr 3	45%	58%	32	334
gr 4	47%	59%	76	334
gr 5	50%	65%	75	334
gr 7	11%	9%	25	334
gr 6	16%	20%	3	334
gr 8	25%	32%	17	334

(b) kod z Grupy 2

testy	Pl %	Pm%	Testy	Mutanty
gr 2.1	0%	0%	0	44
gr 2.2	0%	0%	0	40
gr 2.3	0%	0%	0	305
gr 2	21%	24%	15	353
gr 1	54%	80%	54	334
gr 3	49%	73%	32	333
gr 4	46%	65%	76	341
gr 5	51%	72%	75	341
gr 6	17%	12%	3	341
gr 7	30%	50%	23	341
gr 8	28%	33%	17	341

Tabela 8.9: Mutacje dla poszczególnych grup TDD+M (3–4)

(a) kod z Grupy 3

testy	Pl %	Pm%	Testy	Mutanty
gr 3.1	0%	0%	0	165
gr 3.2	16%	10%	16	197
gr 3.3	39%	40%	22	180
gr 3.4	47%	62%	36	199
gr 3.5	43%	59%	32	204
gr 3	45%	62%	32	196
gr 1	38%	60%	54	194
gr 2	30%	55%	15	186
gr 4	32%	55%	76	186
gr 5	35%	58%	75	186
gr 6	27%	49%	3	186
gr 7	23%	47%	25	186
gr 8	31%	55%	17	186

(b) kod z Grupy 4

testy	Pl %	Pm%	Testy	Mutanty
gr 4.1	0%	0%	0	41
gr 4.2	68%	69%	76	204
gr 4	84%	90%	76	381
gr 1	70%	85%	54	401
gr 2	35%	44%	15	390
gr 3	63%	78%	32	392
gr 5	67%	82%	75	392
gr 6	33%	35%	3	392
gr 7	16%	16%	25	390
gr 8	39%	45%	17	392

Tabela 8.10: Mutacje dla poszczególnych grup TDD (5–6)

(a) kod z Grupy 5

testy	Pl %	Pm%	Testy	Mutanty
gr 5	64%	82%	75	339
gr 1	75%	90%	54	339
gr 2	25%	30%	4	325
gr 3	62%	81%	14	339
gr 4	61%	79%	70	325
gr 6	18%	18%	3	339
gr 7	11%	10%	25	339
gr 8	32%	36%	17	339

(b) kod z Grupy 6

testy	Pl %	Pm%	Testy	Mutanty
gr 6	31%	21%	3	228
gr 1	67%	85%	54	237
gr 2	34%	29%	15	228
gr 3	65%	83%	32	229
gr 4	73%	85%	76	228
gr 5	71%	88%	75	228
gr 7	15%	12%	25	228
gr 8	43%	37%	17	229

Tabela 8.11: Mutacje dla poszczególnych grup TDD (7–8)

(a) kod z Grupy 7

testy	Pl %	Pm%	Testy	Mutanty
gr 7	42%	49%	25	367
gr 1	54%	66%	52	381
gr 2	25%	33%	15	376
gr 3	46%	60%	32	372
gr 4	56%	64%	76	372
gr 5	50%	60%	75	372
gr 6	22%	27%	3	373
gr 8	30%	35%	17	372

(b) kod z Grupy 8

testy	Pl %	Pm%	Testy	Mutanty
gr 8	28%	25%	17	319
gr 1	61%	84%	54	328
gr 2	21%	25%	15	319
gr 3	48%	63%	32	319
gr 4	62%	82%	76	319
gr 5	52%	79%	75	319
gr 6	16%	10%	3	319
gr 7	11%	11%	25	319

Dane w powyższych tabelach pokazują wartości pokryć testami dla każdej z 8 grup poprzez wszystkie iteracje oraz po przeniesieniu testów z innych grup. Tylko w grupach mutujących występują raporty mutacyjne z poszczególnych iteracji. Dla grup niemutujących mutacja została wykonana tylko raz na samym końcu.

W tabelach od 8.12 do 8.19 są przedstawione dane opisujące szczegółowo proces mutacji wykonany dla każdej z poszczególnych grup. Każda z poniższych tabel zawiera wyniki procesu mutacji z testami rodzimymi danej grupy oraz z testami od innych grup. Dane są podzielone względem modułu projektu (tzw. przestrzeni nazw). Dla każdej z przestrzeni nazw jest zaprezentowana liczba klas, które zawiera, oraz pokrycie testami względem mutacji i pokrycie linii kodu. *Pl* oznacza pokryte linie kodu, *Pm* — pokryte mutanty.

Tabela 8.12: Grupa 1 — mutacje w podziale na zbiory klas

	core.Comon	core.Math	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	1	2	4	2
Gr1.1 Pl	56% 9/16	80% 16/20	95% 102/107	83% 48/58	24% 39/163
Gr1.1 Pm	0% 0/17	4% 1/28	89% 119/133	28% 23/81	11% 11/84
Gr1.2 Pl	56% 9/16	80% 16/20	96% 103/107	83% 48/58	24% 39/163
Gr1.2 Pm	0% 0/17	7% 2/28	95% 126/133	30% 24/81	13% 11/84
Gr1 Pl	100% 16/16	90% 18/20	100% 109/109	84% 49/58	33% 53/163
Gr1 Pm	76% 13/17	82% 23/28	95% 129/136	65% 53/81	26% 22/84
Gr2 Pl	50% 8/16	80% 16/20	0% 0/109	69% 40/58	0% 0/163
Gr2 Pm	53% 9/17	100% 26/26	0% 0/130	67% 49/73	0% 0/88
Gr3 Pl	50% 8/16	100% 20/20	91% 99/109	66% 38/58	0% 0/163
Gr3 Pm	53% 9/17	100% 26/26	94% 122/130	49% 36/73	0% 0/88
Gr4 Pl	50% 8/16	100% 20/20	90% 98/109	69% 40/58	3% 5/163
Gr4 Pm	53% 9/17	100% 26/26	85% 110/130	67% 49/73	5% 4/88
Gr5 Pl	50% 8/16	100% 20/20	95% 104/109	69% 40/58	6% 10/163
Gr5 Pm	53% 9/17	100% 26/26	99% 129/130	67% 49/73	5% 4/88
Gr6 Pl	44% 7/16	80% 16/20	0% 0/109	64% 37/58	0% 0/163
Gr6 Pm	47% 8/17	100% 26/26	0% 0/130	44% 32/73	0% 0/88
Gr7 Pl	100% 16/16	90% 18/20	100% 109/109	84% 49/58	33% 53/163
Gr7 Pm	76% 13/17	81% 21/26	96% 125/130	64% 47/73	32% 28/88
Gr8 Pl	50% 8/16	80% 16/20	24% 26/109	69% 40/58	0% 0/163
Gr8 Pm	53% 9/17	100% 26/26	17% 22/130	67% 49/73	0% 0/88

Grupa 1 jest grupą mutującą, której testy są prawie tak samo silne, jak te napisane przez grupę 4. Średnie pokrycie mutacyjne testów grupy 1 w innych grupach wynosi 66,86%. Jest to drugi najlepszy wynik zaraz za grupą 4. Zdolności programistyczne grupy 1 były znacznie mniejsze niż w przypadku grupy 4. Programiści byli mało obeznani z językiem Java, w którym był pisany projekt, ale nadrabiali to stosunkowo skrupulatnym trzymaniem się metodyki TDD+M, co — jak widać — przyniosło bardzo dobre efekty. Wyniki prac grupy przedstawione są w tabelach 8.12 oraz 8.8. Ciekawym faktem jest to, że mimo iż grupa 1 nie przykładała wagi do testów warstwy webowej, ich testy okazały się stosunkowo silne w innych grupach. Ze względu na rozbudowaną warstwę webową także testy innych grup nie radziły sobie najlepiej na tej klasie.

Tabela 8.13: Grupa 2 — mutacje w podziale na zbiory klas

	core.Comon	core.Math	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	1	2	3	2
Gr2.1 Pl	0% 0/4	–	0% 0/16	0% 0/12	0% 0/51
Gr2.1 Pm	0% 0/1	–	0% 0/11	0% 0/6	0% 0/23
Gr2.2 Pl	0% 0/4	–	0% 0/16	0% 0/13	0% 0/50
Gr2.2 Pm	0% 0/1	–	0% 0/11	0% 0/6	0% 0/22
Gr2.3 Pl	0% 0/12	0% 0/16	0% 0/139	0% 0/65	0% 0/76
Gr2.3 Pm	0% 0/10	0% 0/27	0% 0/170	0% 0/64	0% 0/34
Gr2 Pl	100% 12/12	100% 16/16	0% 0/139	97% 60/62	0% 0/194
Gr2 Pm	55% 6/11	89% 24/27	0% 0/170	91% 53/58	0% 0/87
Gr1 Pl	63% 10/16	100% 16/16	91% 126/139	97% 60/62	10% 20/194
Gr1 Pm	61% 11/18	100% 24/24	96% 165/172	100% 50/50	24% 17/70
Gr3 Pl	54% 7/13	100% 16/16	87% 122/141	92% 67/73	0% 0/194
Gr3 Pm	20% 2/10	100% 24/24	96% 165/172	89% 51/57	0% 0/70
Gr4 Pl	35% 6/17	100% 16/16	83% 117/141	89% 64/72	0% 0/194
Gr4 Pm	5% 1/19	100% 24/24	85% 146/172	89% 50/56	0% 0/70
Gr5 Pl	35% 6/17	100% 16/16	91% 129/141	89% 64/72	5% 9/194
Gr5 Pm	5% 1/19	100% 24/24	97% 166/172	89% 50/56	6% 4/70
Gr6 Pl	35% 6/17	94% 15/16	0% 0/141	72% 52/72	0% 0/194
Gr6 Pm	5% 1/19	63% 15/24	0% 0/172	45% 25/56	0% 0/70
Gr7 Pl	24% 4/17	13% 2/16	87% 123/141	26% 19/72	4% 7/194
Gr7 Pm	5% 1/19	17% 4/24	86% 148/172	23% 13/56	7% 5/70
Gr8 Pl	41% 7/17	100% 16/16	24% 34/141	89% 64/72	0% 0/194
Gr8 Pm	11% 2/19	100% 24/24	20% 35/172	89% 50/56	0% 0/70

W tym wypadku dwa testy z grupy 7 musiały zostać zakomentowane. Dane sumaryczne znajdują się w tabeli 8.7. Grupa 2 jest najsłabszą grupą mutacyjną. Jej członkowie nie zastosowali się do zasad TDD+M ani nawet TDD. Praktycznie wszystkie testy zostały napisane w ostatniej iteracji.

Tabela 8.14: Grupa 3 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	2	3	2
Gr3.1 Pl	–	0% 0/40	0% 0/75	–
Gr3.1 Pm	–	0% 0/26	0% 0/139	–
Gr3.2 Pl	0% 0/7	73% 29/40	0% 0/75	0% 0/51
Gr3.2 Pm	0% 0/1	73% 19/26	0% 0/139	0% 0/26
Gr3.3 Pl	44% 4/9	100% 40/40	39% 30/77	0% 0/50
Gr3.3 Pm	33% 1/3	100% 26/26	37% 45/121	0% 0/25
Gr3.4 Pl	44% 4/9	100% 40/40	100% 73/73	0% 0/117
Gr3.4 Pm	33% 1/3	100% 26/26	92% 94/102	0% 0/65
Gr3.5 Pl	44% 4/9	100% 40/40	100% 73/73	0% 0/137
Gr3.5 Pm	33% 1/3	100% 26/26	92% 94/102	0% 0/70
Gr3 Pl	44% 4/9	100% 40/40	100% 73/73	0% 0/137
Gr3 Pm	33% 1/3	100% 26/26	92% 94/102	0% 0/65
Gr1 Pl	93% 13/14	10% 4/40	100% 73/73	7% 9/137
Gr1 Pm	91% 10/11	0% 0/25	100% 102/102	9% 5/56
Gr2 Pl	44% 4/9	0% 0/40	100% 73/73	0% 0/137
Gr2 Pm	33% 1/3	0% 0/25	100% 102/102	0% 0/56
Gr4 Pl	78% 7/9	10% 4/40	99% 72/73	1% 1/137
Gr4 Pm	33% 1/3	0% 0/25	99% 101/102	0% 0/56
Gr5 Pl	44% 4/9	10% 4/40	100% 73/73	7% 9/137
Gr5 Pm	33% 1/3	0% 0/25	100% 102/102	9% 5/56
Gr6 Pl	33% 3/9	0% 0/40	90% 66/73	0% 0/137
Gr6 Pm	0% 0/3	0% 0/25	90% 92/102	0% 0/56
Gr7 Pl	33% 3/9	10% 4/40	62% 45/73	5% 7/137
Gr7 Pm	0% 0/3	0% 0/25	80% 82/102	9% 5/56
Gr8 Pl	44% 4/9	8% 3/40	100% 73/73	0% 0/137
Gr8 Pm	33% 1/3	0% 0/25	100% 102/102	0% 0/56

Grupa 3 prowadziła eksperyment zgodnie z modelem iteracyjnym TDD+M. Nie występowały znaczne skoki w liczbie mutantów i testów. Każda iteracja bazowała na efektach poprzedniej. Na koniec, podczas ostatniej iteracji grupa przeprowadziła refaktoring kodu i testów czyszcząc niepotrzebne metody i „testy widmo” (testy puste, które nic nie sprawdzają). Ten zabieg doprowadził do zmniejszenia liczby mutantów. Podczas użycia testów z grupy 4 koniecznym było zakomentowanie pięciu testów, co zmniejszyło procentowe pokrycie. Testy z grupy 6 udało się dopasować dobrze jedynie do kodu w części `core.Math.Shapes`. Dla innych modułów większość testów musiała zostać zakomentowana bądź bardzo ograniczona.

Tabela 8.15: Grupa 4 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	2	3	2
Gr4.1 Pl	0% 0/4	0% 0/16	0% 0/12	0% 0/52
Gr4.1 Pm	0% 0/1	0% 0/11	0% 0/6	0% 0/23
Gr4.2 Pl	100% 7/7	81% 96/119	86% 132/154	0% 0/67
Gr4.2 Pm	100% 3/3	77% 125/162	73% 149/204	0% 0/30
Gr4 Pl	100% 7/7	93% 112/120	90% 138/154	63% 64/102
Gr4 Pm	100% 3/3	95% 148/155	91% 164/180	60% 26/43
Gr1 Pl	77% 10/13	89% 107/120	84% 129/154	26% 26/101
Gr1 Pm	86% 12/14	95% 141/149	88% 169/193	38% 17/45
Gr2 Pl	71% 5/7	0% 0/120	84% 129/154	0% 0/101
Gr2 Pm	67% 2/3	0% 0/149	89% 171/193	0% 0/45
Gr3 Pl	67% 6/9	86% 103/120	86% 133/154	0% 0/101
Gr3 Pm	60% 3/5	87% 129/149	89% 172/193	0% 0/45
Gr5 Pl	67% 6/9	92% 110/120	84% 129/154	12% 12/101
Gr5 Pm	60% 3/5	98% 146/149	87% 168/193	9% 4/45
Gr6 Pl	56% 5/9	0% 0/120	79% 121/154	0% 0/101
Gr6 Pm	20% 1/5	0% 0/149	70% 136/193	0% 0/45
Gr7 Pl	71% 5/7	18% 22/120	17% 26/154	7% 7/101
Gr7 Pm	33% 1/3	9% 14/149	22% 42/193	11% 5/45
Gr8 Pl	67% 6/9	25% 30/120	73% 112/154	0% 0/101
Gr8 Pm	60% 3/5	17% 25/149	77% 149/193	0% 0/45

Grupa 4 jako grupa mutacyjna także wykonywała iteracje zgodne z metodyką TDD+M. W końcowej iteracji osiągnęła pokrycie na poziomie 84% pokrytego kodu oraz 90% pokrytych mutantów. W ostatniej iteracji członkowie zespołu poprawili znacznie jakość testów oraz poszerzyli część przepływów w kodzie, co wygenerowało większą liczbę mutantów. Testy tej grupy są jednymi z mocniejszych — średnie pokrycie mutacyjne z użyciem testów tej grupy we wszystkich innych grupach wyniosło 67.29%. Jest to dobry wynik, sugerujący przewagę metodyki TDD+M nad TDD. Grupa 4 to także grupa o najwyższych zdolnościach programistycznych według ankiety[9]. Najprawdopodobniej wysokie zdolności programistyczne także miały wpływ na wyniki w tej grupie.

[9]Wyniki tej grupy sugerują, że ocena własna za pomocą ankiety w tym przypadku była stosunkowo miarodajna.

Tabela 8.16: Grupa 5 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	2	3	2
Gr5 Pl	31% 5/16	100% 161/161	90% 89/99	7% 9/138
Gr5 Pm	6% 1/17	100% 181/181	92% 91/99	10% 4/42
Gr1 Pl	100% 16/16	95% 153/161	89% 88/99	40% 55/138
Gr1 Pm	76% 13/17	98% 178/181	92% 91/99	52% 22/42
Gr2 Pl	83% 5/6	0% 0/161	95% 94/99	0% 0/138
Gr2 Pm	67% 2/3	0% 0/181	95% 94/99	0% 0/42
Gr3 Pl	67% 4/6	89% 144/161	95% 94/99	3% 4/138
Gr3 Pm	33% 1/3	88% 160/181	95% 94/99	7% 3/42
Gr4 Pl	100% 5/5	93% 150/161	95% 94/99	0% 0/138
Gr4 Pm	100% 2/2	97% 176/181	85% 84/99	0% 0/42
Gr6 Pl	31% 5/16	0% 0/161	69% 68/99	0% 0/138
Gr6 Pm	6% 1/17	0% 0/181	62% 61/99	0% 0/42
Gr7 Pl	0% 0/16	17% 27/161	12% 12/99	5% 7/138
Gr7 Pm	0% 0/17	10% 18/181	11% 11/99	12% 5/42
Gr8 Pl	31% 5/16	25% 40/161	89% 88/99	0% 0/138
Gr8 Pm	6% 1/17	17% 30/181	92% 91/99	0% 0/42

Grupa 5 była grupą niemutującą, której testy wykazały niską kompatybilność z innymi testami. Wytworzyła aż 75 testów i uzyskała pokrycie mutacyjne na poziomie 82%. Używając testów z grupy 2 oraz 4 jedenaście testów nie przechodziło bez mutacji. Używając testów z grupy 3 nie przeszło 18 testów. Ponadto, w tym przypadku należało także dodać parę getterów i setterów, tak żeby testy były kompatybilne. Wygenerowało to dodatkowych 6 mutantów. Testy z grupy 1 wypadły lepiej niż w rodzimym kodzie, ponieważ grupa 5 napisała mniej kodu niż grupa 1. Grupa 5 w przeciwieństwie do 7 ma niezaimplementowane niektóre klasy oraz metody pozostawione w formie abstrakcyjnej.

Po głębszej analizie kodu grupy 5 można zauważyć, że testy generują bardzo wiele błędów wykonania, co sprawia, że mutanty są ubijane masowo. Mimo niskiej wartości metryki złożoności cyklomatycznej (co sugeruje prostotę, a więc i potencjalnie stabilność kodu), po krótkim oglądzie kodu widać, że jest w nim spora liczba nieobsłużonych wyjątków, nieuwzględnionych przypadków brzegowych, a kod jest mało elastyczny. Opisana sytuacje skutkuje tym, że większość mutacji nie tyle jest wykrywana poprzez test dający złą odpowiedź, co po prostu przez fakt, iż program rzuca nieobsłużony wyjątek[10].

[10]Z punktu widzenia systemu sam *runtime error* tez jest wykryciem mutanta.

Tabela 8.17: Grupa 6 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	2	3	2
Gr6 Pl	80% 4/5	0% 0/96	84% 81/97	0% 0/78
Gr6 Pm	50% 1/2	0% 0/137	72% 48/67	0% 0/22
Gr1 Pl	100% 9/9	85% 82/96	88% 85/97	15% 12/78
Gr1 Pm	100% 11/11	88% 120/137	94% 63/67	36% 8/22
Gr2 Pl	80% 4/5	0% 0/96	94% 91/97	0% 0/78
Gr2 Pm	50% 1/2	0% 0/137	97% 65/67	0% 0/22
Gr3 Pl	83% 5/6	83% 80/96	99% 96/97	0% 0/78
Gr3 Pm	67% 2/3	88% 120/137	100% 67/67	0% 0/22
Gr4 Pl	100% 5/5	77% 74/96	98% 95/97	36% 28/78
Gr4 Pm	100% 2/2	84% 115/137	100% 67/67	45% 10/22
Gr5 Pl	100% 5/5	90% 86/96	99% 96/97	12% 9/78
Gr5 Pm	100% 2/2	93% 127/137	100% 67/67	18% 4/22
Gr7 Pl	0% 0/5	24% 23/96	12% 12/97	9% 7/78
Gr7 Pm	0% 0/2	15% 21/137	3% 2/67	23% 5/22
Gr8 Pl	83% 5/6	39% 37/96	80% 78/97	0% 0/78
Gr8 Pm	67% 2/3	24% 33/137	75% 50/67	0% 0/22

Grupa 6 jest grupą niemutującą, stosującą zwykłe podejście TDD. Do testów z grup 3, 7 oraz 8 zostały dodane gettery i settery, które jako dodatkowy kod wygenerowały parę nowych mutantów.

Tabela 8.18: Grupa 7 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	2	5	2
Gr7 Pl	85% 11/13	83% 101/121	62% 73/118	4% 8/207
Gr7 Pm	56% 5/9	81% 96/118	58% 73/126	4% 5/114
Gr1 Pl	100% 17/17	94% 114/121	78% 96/123	13% 27/207
Gr1 Pm	100% 18/18	97% 115/118	74% 97/131	18% 20/114
Gr2 Pl	80% 16/20	0% 0/121	86% 102/118	0% 0/207
Gr2 Pm	78% 14/18	0% 0/118	87% 109/126	0% 0/114
Gr3 Pl	85% 11/13	86% 104/121	80% 99/123	0% 0/207
Gr3 Pm	78% 7/9	97% 114/118	77% 101/131	0% 0/114
Gr4 Pl	100% 13/13	83% 100/121	78% 96/123	24% 49/207
Gr4 Pm	100% 9/9	84% 99/118	74% 97/131	28% 32/114
Gr5 Pl	85% 11/13	90% 109/121	81% 100/123	5% 11/207
Gr5 Pm	78% 7/9	98% 116/118	75% 98/131	4% 4/114
Gr6 Pl	79% 11/14	0% 0/121	73% 90/123	0% 0/207
Gr6 Pm	70% 7/10	0% 0/118	69% 91/131	0% 0/114
Gr8 Pl	85% 11/13	21% 25/121	83% 102/123	0% 0/207
Gr8 Pm	78% 7/9	19% 22/118	78% 102/131	0% 0/114

Analiza testów grupy 7 wykazała że dla testów z grupy 2 w celu zagwarantowania kompatybil-

ności musiały zostać dodane settery i gettery[11], co poskutkowało zwiększoną liczbą mutantów. Można to zaobserwować także w tabeli 8.7. Dla grupy 4 koniecznym było zakomentowanie 11 testów w związku z niekompatybilną logiką wewnętrzną modułu `core.web`. W grupie 6 został dodany jeden setter a w grupie 1 – dwa settery kolekcji, co zwiększyło liczbę wygenerowanych mutantów.

Tabela 8.19: Grupa 8 — mutacje w podziale na zbiory klas

	core.Comon	core.Math.Matrix	core.Math.Shapes	core.web
Liczba klas	1	3	3	2
Gr8 Pl	86% 6/7	18% 26/147	98% 84/86	0% 0/174
Gr8 Pm	50% 1/2	11% 19/172	73% 59/81	0% 0/64
Gr1 Pl	100% 11/11	93% 137/147	95% 82/86	15% 26/174
Gr1 Pm	100% 11/11	98% 169/172	95% 77/81	28% 18/64
Gr2 Pl	86% 6/7	0% 0/147	97% 83/86	0% 0/174
Gr2 Pm	50% 1/2	0% 0/172	96% 78/81	0% 0/64
Gr3 Pl	86% 6/7	90% 132/147	71% 61/86	0% 0/174
Gr3 Pm	50% 1/2	98% 168/172	38% 31/81	0% 0/64
Gr4 Pl	86% 6/7	86% 126/147	97% 83/86	23% 40/174
Gr4 Pm	50% 1/2	94% 162/172	96% 78/81	30% 19/64
Gr5 Pl	86% 6/7	94% 138/147	97% 83/86	5% 9/174
Gr5 Pm	50% 1/2	98% 169/172	96% 78/81	6% 4/64
Gr6 Pl	71% 5/7	0% 0/147	71% 61/86	0% 0/174
Gr6 Pm	0% 0/2	0% 0/172	38% 31/81	0% 0/64
Gr7 Pl	0% 0/7	10% 14/147	27% 23/86	4% 7/174
Gr7 Pm	0% 0/2	4% 7/172	27% 22/81	8% 5/64

Grupa 8 byłą grupą niemutacyjną. Do testów z grupy 7 wymagane było dodanie jednego settera.

Tabela 8.20: Mutacje – dane zbiorcze dla grup i ich własnych testów

(a) Grupy mutacyjne

Grupa	Pl%	Pm %
1	67%	70%
2	21%	24%
3	45%	62%
4	84%	90%
Średnia	54%	62%
Odch. std.	27%	27%

(b) Grupy niemutacyjne

Grupa	Pl%	Pm %
5	64%	82%
6	31%	21%
7	42%	49%
8	28%	25%
Średnia	41%	44%
Odch. std.	16%	28%

W tabeli 8.20 znajdują się dane zbiorcze pokazujące procentowe pokrycia dla każdej z grup i ich własnych testów. *Pl* oznacza pokrycie instrukcyjne, *Pm* — pokrycie mutacyjne.

Wszystkie powyższe dane były zebrane bez ingerencji w kod oraz testy poza drobnymi zmianami w postaci dodania setterów czy getterów dla pól prywatnych. Skrypty uruchamiające mutacje także nie były modyfikowane, co w efekcie daje drobne różnice w mutowanych pakietach (niektóre

[11]Gettery oraz settery to metody ustawiające prywatne bądź chronione pola obiektów w programie.

grupy nie mutowały modułu webowego lub modułu `core.math` zawierającego abstrakcyjne klasy bazowe).

W tabeli 8.21 przedstawiono zbiorcze wyniki pokrycia instrukcyjnego i mutacyjnego dla wszystkich 64 par (kod grupy X, testy grupy Y), gdzie $X, Y \in \{1, ..., 8\}$ oraz $X \neq Y$ w przeprowadzonym eksperymencie krzyżowym. Kolumny odpowiadają kodom poszczególnych ośmiu grup, rzędy zaś – testom tych grup. W pierwszym wierszu liczby w nawiasach oznaczają liczbę mutantów wygenerowanych dla kodu danej grupy. Liczby w nawiasach w pierwszej kolumnie oznaczają liczbę testów napisanych przez daną grupę. Na przykład, grupa nr 4 napisała 76 testów, a dla jej kodu wygenerowane zostały 392 mutanty.

Z analizy wyników wyłączono sytuacje $X = Y$, czyli takie, w których kod danej grupy był testowany jej własnymi testami (są to wartości leżące na przekątnej w tabeli 8.21), pozostawiając 64-8 = 56 danych pomiarowych. Dzięki temu zwiększono obiektywność oceny, ponieważ każdy zbiór testów wykonywany był dla kodu, dla którego nie był oryginalnie projektowany. Wyniki pomiarów dla grup na ich własnych testach przedstawiono w tabeli 8.20.

Każda komórka (X, Y) tabeli pokazuje miary pokrycia instrukcyjnego i mutacyjnego dla kodu grupy nr X testowanego testami grupy nr Y. Na przykład, testy grupy nr 3 wykonane na kodzie grupy nr 6 osiągnęły 65% pokrycia instrukcyjnego oraz 83% pokrycia mutacyjnego. Dla grup 5, 6, 7, 8, które nie używały procesu testowania mutacyjnego podczas wytwarzania kodu mutacja została przeprowadzona *post factum* na ostatecznej wersji kodu.

Dwie ostatnie kolumny tabeli pokazują średnie wartości pokrycia dla testów wykonanych na wszystkich grupach TDD+M (odp. TDD). Podobnie, dwa ostatnie wiersze pokazują średnie pokrycie dla danego kodu i wszystkich testów grup TDD+M (odp. grup TDD).

Tabela 8.21: Pokrycie instrukcyjne/mutacyjne (w %) dla wszystkich kombinacji (kod grupy X, testy grupy Y)

Kod →	01M	02M	03M	04M	05	06	07	08	μM	μ
Testy ↓	(334)	(341)	(186)	(392)	(339)	(228)	(372)	(319)		
01M (54)		54/80	38/60	70/85	75/90	67/85	54/66	61/84	54/75	64/81
02M (15)	17/21		30/55	35/44	25/30	34/29	25/33	21/25	27/40	26/29
03M (32)	45/58	49/73		63/78	62/81	65/83	64/60	48/63	52/70	60/72
04M (76)	47/59	46/65	32/55		61/79	73/85	56/64	62/82	42/60	63/78
05 (75)	50/65	51/72	35/58	67/82		71/88	50/60	52/79	51/69	58/76
06 (3)	16/20	17/12	27/49	33/35	18/18		22/27	16/10	23/29	19/18
07 (25)	11/9	30/50	23/47	16/16	11/10	15/12		11/11	20/31	12/11
08 (17)	25/32	28/33	31/55	39/45	32/36	43/37	30/35		31/41	35/36
μM (44)	36/46	50/73	33/57	56/69	56/70	60/71	50/56	48/64		
μ (30)	26/32	32/42	29/52	39/45	20/21	43/46	34/41	26/33		

W tabeli 8.22 porównano średnie wartości metryk pokrycia dla eksperymentów wykorzystujących różnego typu porównania grup TDD+M i TDD. Kod typu porównania (pierwsza kolumna) XY należy rozumieć jako "testy X na kodzie Y", gdzie $X, Y \in \{M, T, A\}$. Symbol M oznacza zespoły pracujące w podejściu TDD+M, symbol T – zespoły pracujące w metodyce TDD, zaś symbol

A – wszystkie zespoły. Zatem np. kod MT oznacza wyniki obejmujące testy grup mutacyjnych TDD+M wykonanych na kodzie grup TDD.

Pokrycie dla MA i TA jest uśrednione z 28 pomiarów (4 suity testowe $\times$ 7 grup, z wyłączeniem kodu grupy, która pisała testy. Pokrycie dla MT i TM jest uśrednione z 16 pomiarów (4 suity testowe z czterech grup $\times$ 4 kody z 4 grup). W przypadku MM i TT, pokrycie jest uśrednione z 12 pomiarów (wyłączając testy uruchamiane na kodzie pisanym przez ten sam zespół). Używając podejścia krzyżowego możemy porównywać różne grupy z wykorzystaniem różnych kryteriów porównania. Pozwoli nam to uzasadnić tezy 3.1, 3.2 oraz 3.3.

Teza 3.1: testy napisane w podejściu TDD+M dają lepsze pokrycie kodu niż te pisane w czystej metodyce TDD, bez udziału mutacji.

W celu uzasadnienia tezy 3.1 dokonano porównania wyników MA i TA w terminach pokrycia instrukcyjnego. Pokrycie to dla MA jest o 58.5% wyższe niż dla TA (49.3% wobec 31.1%; różnica w punktach procentowych wynosi 18.2). To pokazuje, że testy pisane przy użyciu podejścia TDD+M są silniejsze, ponieważ osiągają lepsze pokrycie kodu. Zauważmy, że różnica ta jest istotna również, gdy ograniczymy nasze pomiary tylko do kodu grup TDD. W tym przypadku różnica pomiędzy grupami MT i TT wynosi $53.3\% - 30.9\% = 22.4\%$ w terminach pokrycia instrukcyjnego.

Teza 3.2: testy napisane w podejściu TDD+M są silniejsze niż te pisane w czystej metodyce TDD.

Aby potwierdzić tezę 3.2 porównano grupy MA z TA w terminach pokrycia mutacyjnego, tzn. porównano wyniki testów obu podejść na wszystkich kodach, wyłączając kod zespołu, dla którego dane testy były oryginalnie napisane. Grupy MA osiągnęły o 60.6% wyższe pokrycie (średnio 63.3% pokrycia mutacyjnego dla MA i 39.4% dla TA, czyli o 23.9 punktów procentowych mniej). Różnica w pokryciu mutacyjnym jest istotna również, gdy ograniczymy się jedynie do kodu z grup TDD i wynosi $64.9\% - 35.2\% = 29.7\%$. To pokazuje, że analiza mutacyjna może być bardzo skutecznym narzędziem. Testy zespołów nie wykorzystujących tej techniki są zdecydowanie słabsze niż testy zespołów używających TDD+M. Prawdopodobieństwo wykrycia defektu będzie mniejsze, niż w przypadku zespołów TDD+M.

Zauważmy, że testy zaprojektowane przez grupy jednego typu nie były przeznaczone do wykonania na kodzie grup typu drugiego. Programiści pisali testy z przeznaczeniem zastosowania ich do własnego kodu – nie znali oni kodu innych grup. Zatem testy z grup TDD+M są niezależne od kodu grup TDD i *vice versa*. Taki sposób oceny testów jest bardziej miarodajny niż zwykłe sprawdzanie testów na własnym kodzie. Biorąc pod uwagę wyniki eksperymentu krzyżowego wydaje się zasadną teza, iż podejście TDD+M pozwala stworzyć zespołom silniejsze testy, niż w przypadku, gdy stosowane jest tylko czyste podejście TDD bez angażowania procesu mutacyjnego.

Tabela 8.22: Porównanie pokrycia dla grup TDD+M i TDD (w %)

		średnie pokrycie ± sd	
kryterium	opis kryterium	instrukcyjne	mutacyjne
MA	testy TDD+M na wszystkich kodach	49.3 ± 16.47	63.3 ± 20.26
TA	testy TDD na wszystkich kodach	31.1 ± 16.25	39.4 ± 23.47
MM	testy TDD+M na kodzie TDD+M	43.8 ± 13.7	61.0 ± 15.5
MT	testy TDD+M na kodzie TDD	53.3 ± 16.9	64.9 ± 22.3
TM	testy TDD na kodzie TDD+M	31.1 ± 14.3	42.5 ± 20.8
TT	testy TDD na kodzie TDD	30.9 ± 18.4	35.2 ± 25.9
AA	Średnia (z wszystkich 56 par)	40.1 ± 18.7	51.3 ± 24.9

W celu zweryfikowania statystycznej istotności różnic pomiędzy opisanymi wyżej metrykami pokryć przeprowadzono dwa testy t-Studenta (oba w wersji obustronnej i niesparowanej) dla wartości pokrycia instrukcyjnego (teza 3.1) i mutacyjnego (teza 3.2) dla grup TDD+M i TDD, zastosowane do wszystkich 8 projektów i 56 porównań (jak wspomniano wcześniej, z analizy wyłączono pomiary kodu na swoich własnych testach). Grupy, dla których wykonano porównanie, stanowią wartości z wierszy 1-4 oraz 5-8 tabeli 8.21. Otrzymaliśmy zatem dwie próby o równych rozmiarach (28) dla pokrycia kodu i identyczne próby dla pokrycia mutacyjnego.

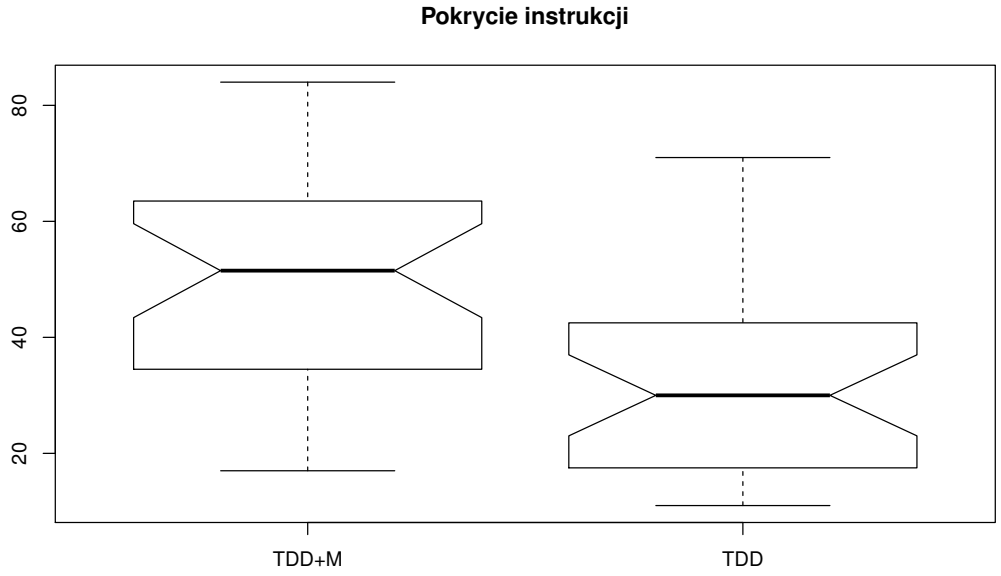
Zanim przeprowadzono testy t, należało sprawdzić, czy ich założenia są spełnione, mianowicie: 1) homogeniczność wariancji w obu populacjach, 2) rozkład normalny estymatora wartości średniej. Rozkłady wartości mierzonych parametrów we wszystkich cztery próbach są bliskie rozkładowi normalnemu (wartości p dla testu Shapiro-Wilka dla pokrycia instrukcyjnego: p=0.1825 dla grup TDD+M oraz $p = 0.02$ dla grup TDD; dla pokrycia mutacyjnego: $p = 0.011$ dla grup TDD+M oraz $p = 0.01$ dla grup TDD). Możemy zatem zastosować test F aby sprawdzić homogeniczność wariancji w próbkach. W przypadku pokrycia instrukcyjnego $p = 0.94$, a w przypadku pokrycia mutacyjnego $p = 0.45$, zatem nie możemy odrzucić hipotezy o równości wariancji zarówno dla pokrycia instrukcyjnego jak i mutacyjnego. Ponieważ próbki pochodzą z rozkładów bliskich normalnemu, estymatory wartości średniej będą również miały rozkład normalny. Moc testu t w naszym przypadku, tzn. w przypadku próby o wielkości 28, na poziomie $\alpha = 0.05$ i wielkości efektu 0.8 wynosi 0.836, co można uznać za wynik w zupełności zadowalający.

Powyższa analiza uzasadnia zastosowanie testu t Studenta do analizy istotności różnic pomiędzy średnimi wartościami pokrycia instrukcyjnego oraz mutacyjnego. Wyniki przedstawione są na rys. 8.10 oraz 8.11. Szczegółowe wyniki testów przedstawiono natomiast w tabeli 8.23.

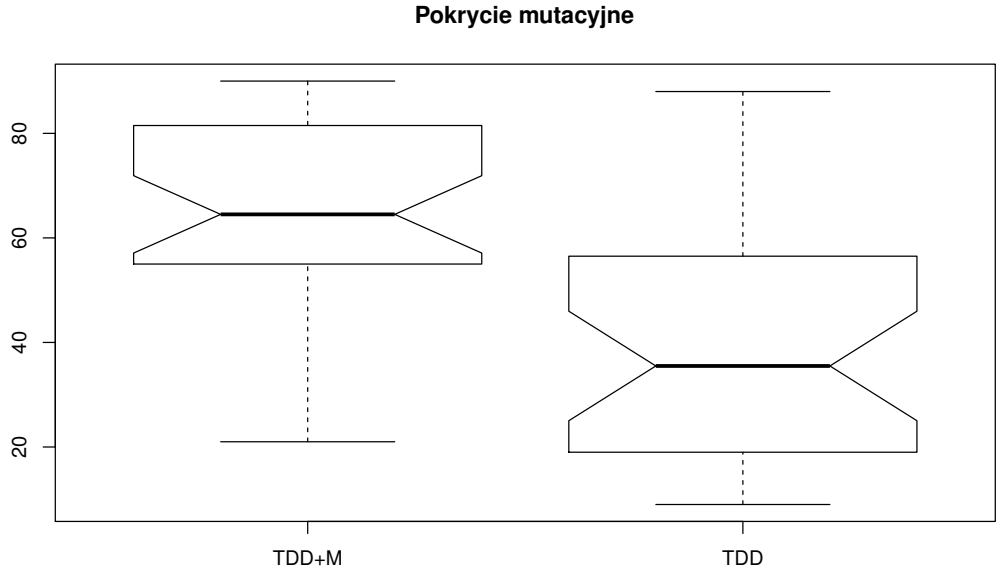
Oba testy pokazują statystyczną istotność różnic w pokryciu osiągniętym przez testy grup TDD+M oraz TDD (w obu przypadkach $p < 0.001$). Współczynnik d Cohena wynosi 1.091 dla pokrycia instrukcyjnego oraz 1.07 dla pokrycia mutacyjnego – taki efekt uznawany jest jako duży (współczynnik d Cohena zdefiniowany jest jako $d = \frac{\mu_1 - \mu_2}{s}$, gdzie $\mu_{1,2}$ to średnie w grupach, a s – odchylenie standardowe obliczane ze wzoru $s = \sqrt{\frac{n_1 s_1^2 + n_2 s_2^2}{n_1 + n_2 - 2}}$, gdzie $s_{1,2}$ to odchylenia standardowe w grupach, a $n_{1,2}$ to liczności tych grup; zwyczajowo wartości d 0.2, 0.5, 0.8 interpretuje się odpowiednio jako efekt mały, średni i duży, choć jest to kwestia czysto umowna [89, 90]).

Wyniki potwierdzają zasadność tez 3.1 i 3.2: testy pisane w podejściu TDD+M pozwalają na

uzyskanie wyższego pokrycia instrukcyjnego oraz osiągnąć wyższe pokrycie mutacyjne. Oznacza to, że podejście TDD+M pozwala programistom pisać silniejsze testy (w sensie ich możliwości do wykrywania defektów).



Rysunek 8.10: Różnice w pokryciu instrukcyjnym między grupami TDD+M i TDD



Rysunek 8.11: Różnice w pokryciu mutacyjnym między grupami TDD+M i TDD

Tabela 8.23: Wyniki testów t-Studenta dla grup TDD+M i TDD

	t-test dla różnicy pomiędzy:	
parametr	pokryciem instrukcyjnym	pokryciem mutacyjnym
N_{TDD+M}	28	28
N_{TDD}	28	28
średnia TDD+M	49.25	63.28
średnia TDD	31.07	39.39
średnia różnica	18.18	23.89
95% przedział ufności	[9.25, 27.10]	[11.93, 35.85]
sd TDD+M	16.77	20.63
sd TDD	16.54	23.90
SEM TDD+M	3.17	3.90
SEM TDD	3.13	4.52
p-value	0.0001483	0.000191
t-test	4.0823	4.0048
df	54	54
błąd std różnicy	4.45	5.97
wielkość efektu (współczynnik d Cohena)	1.091	1.07

Teza 3.3: jakość zewnętrzna oprogramowania jest wyższa, gdy zamiast TDD stosuje się podejście TDD+M

Tabela 8.24 przedstawia liczbę defektów znalezionych przez suity testowe dla każdej pary (kod, testy).

Tabela 8.24: Defekty znalezione dla każdej pary (kod, testy)

	Kod z grupy							
Testy z grupy ↓	01M	02M	03M	04M	05	06	07	08
01M	–	–	–	–	–	–	–	–
02M	–	–	–	–	11	–	–	–
03M	–	–	–	–	18	–	–	–
04M	–	–	–	–	11	–	–	–
05	–	–	5	–	–	–	–	–
06	–	–	–	–	–	–	–	–
07	–	2	–	–	–	–	–	–
08	–	–	–	–	–	–	–	–

Jak widać z tabeli, testy grup TDD+M były w stanie wykryć, średnio, 10 (= (0+11+18+11)/4) defektów w kodzie z grup TDD. Z drugiej strony, testy z grup TDD były w stanie znaleźć, średnio, tylko 1.75 defektu w kodzie z grup TDD+M. Testy z grup 1, 6 i 8 nie były w stanie znaleźć żadnych defektów w żadnym z projektów.

To uzasadnia tezę 3.3: kod napisany z użyciem podejścia TDD+M jest lepszej jakości niż kod pisany z użyciem samego TDD. Jednakże, ze względu na mały rozmiar próby (cztery zespoły TDD+M i cztery TDD) nie możemy przeprowadzić formalnego, statystycznego testu potwierdzającego tę hipotezę. Możemy jedynie dokonać interpretacji wyników z tabeli 8.24.

Z tabeli 8.7 wiemy, że całkowita złożoność cyklomatyczna dla kodu grup TDD+M (odp. TDD)

wyniosła 171 (odp. 174.5), natomiast średnia liczba linii kodu LOC – odpowiednio 750 i 792. Oznacza to, że projekty pisane w obu grupach są podobne w sensie rozmiaru i strukturalnej złożoności. Biorąc pod uwagę metrykę LOC można stwierdzić, że testy TDD+M były w stanie wykryć średnio 12.62 na 1000 linii kodu, podczas gdy testy TDD – tylko 2.33 defekty na 1000 linii kodu. Pokazuje to, że testy TDD+M wydają się być silniejsze i bardziej efektywne w znajdowaniu defektów niż testy pisane w czystym podejściu TDD.

Kod grupy 5 miał 18 wykrytych defektów. 11 defektów wykrytych przez testy grup 2 i 4 stanowią właściwy podzbiór tych 18 defektów, wykrytych przez testy grupy 3.

8.5.2 Uwaga o efektywności wykrywania defektów

Można ocenić siłę przypadków testowych i jakość kodu za pomocą jeszcze jednej miary. Ponieważ mamy cztery niezależne zestawy testów TDD+M dla tego samego zestawu czterech programów TDD i cztery niezależne zestawy testów TDD dla tego samego zestawu czterech programów TDD+M, można porównać powyższe zestawy testów napisane przy użyciu podejścia TDD oraz TDD+M pod względem ich zdolności do wykrywania defektów. W tym celu posłużymy się miarą DDE (Defect Detection Efficiency) dla obu wyżej wymienionych podejść, zakładając, że mamy tylko jedną fazę/etap rozwoju. Niech $S = \{1, 2, ...\}$ będzie zbiorem wszystkich zespołów (reprezentowanych przez indeksy) oraz niech $S = T \cup M$, $T \cap M = \emptyset$, gdzie T oznacza zespoły, które używały tylko TDD bez mutacji a M – te, które stosowały podejście TDD+M. Przez d_{ij}, $i, j \in T$ oznaczmy liczbę wykrytych defektów za pomocą i-tych testów na j-tym zestawie kodu danego zespołu, a przez D_j – całkowitą liczbę różnych defektów wykrytych w j-tym kodzie grupy. W podanym przypadku, poprzez manualną weryfikację można wywnioskować, że $D_2 = 2$, $D_3 = 5$ oraz $D_5 = 18$. W innych grupach nie znaleziono żadnych defektów, zatem bazując na powyższej analizie przyjmujemy, że kody tych grup są pozbawione usterek[12].

Zdefiniujmy teraz metryki efektywności wykrywania defektów DDE_T oraz DDE_M dla podejść TDD i TDD+M poprzez uśrednienie skuteczności wykrywania defektów dla wszystkich testów TDD i TDD+M.

$$DDE_T = \frac{1}{|\{j \in M : D_j > 0\}|} \times \sum_{j \in M : D_j > 0} \frac{1}{|T|} \sum_{i \in T} \frac{d_{ij}}{D_j},$$

$$DDE_M = \frac{1}{|\{j \in T : D_j > 0\}|} \times \sum_{j \in T : D_j > 0} \frac{1}{|M|} \sum_{i \in M} \frac{d_{ij}}{D_j}.$$

Dla danych z tabeli 8.24 uzyskujemy:

$$\begin{aligned} DDE_T &= \frac{1}{2} \times \left[\frac{1}{4} \left(0 + 0 + \frac{2}{2} + 0 \right) + \frac{1}{4} \left(\frac{5}{5} + 0 + 0 + 0 \right) \right] = 0.25, \\ DDE_M &= 1 \times \frac{1}{4} \left(0 + \frac{11}{18} + \frac{18}{18} + \frac{11}{18} \right) = 0.55. \end{aligned}$$

[12] Oczywiście nie znaczy to, że te kody są bezbłędne, a jedynie, że testy nie wykryły żadnego defektu.

Ta analiza odpowiada na tezy badawcze pierwszą oraz drugą pod względem siły testu mierzonej przez DDE. Może również pośrednio odpowiedzieć na pytanie postawione w trzeciej tezie badawczej, gdy założone zostanie, że wszystkie wykryte defekty zostały usunięte. W takim przypadku można twierdzić, że zestawy testów z wyższym DDE mają lepszy wpływ na ogólną jakość kodu niż zestawy testów z niższym DDE. W podanym przypadku mamy $\frac{DDE_M}{DDE_T} = \frac{0.55}{0.25} = 2.2$, co pokazuje, że zestawy testów TDD+M są ponad dwukrotnie bardziej efektywne w wykrywaniu defektów niż zestawy napisane w przy użyciu czystego TDD.

8.5.3 Wnioski z drugiego eksperymentu

Na podstawie powyższych danych można stwierdzić, że prowadzenie projektu w metodyce TDD+M daje ok. 60% większe pokrycie instrukcyjne i mutacyjne w porównaniu z metodyką TDD[13]. Zastosowanie mutacji w procesie TDD istotnie przyczynia się do tworzenia mocniejszych, efektywniejszych testów o większych zdolnościach pokrycia oraz wykrywania defektów.

Testy przeniesione z grup mutujących do niemutujących osiągały o 72% większe pokrycie instrukcyjne oraz o 84% większe pokrycie mutacyjne niż testy własne grup niemutujących. Z kolei testy przeniesione z grup niemutujących do mutujących osiągały tylko 71% pokrycia instrukcyjnego oraz 70% pokrycia mutacyjnego testów TDD+M na własnym kodzie[14]. Te wyniki pokazują, jak odporny na usterki może być dobrze przetestowany kod, w którym testy zmusiły do uwzględnienia wszystkich przypadków skrajnych[15]. Ponadto potwierdziły się wyniki z eksperymentu pierwszego, które jednoznacznie wskazały, że metoda TDD+M daje mocniejsze testy oraz – jako produkt uboczny – kod o małej spodziewanej liczbie usterek.

8.6 Wnioski ogólne

Wyniki obu eksperymentów opisanych w niniejszym rozdziale potwierdzają tezy nr 3.1, 3.2 i 3.3: testy pisane w podejściu TDD+M dają wyższe pokrycie kodu i osiągają lepsze pokrycie mutacyjne. Oznacza to, że podejście TDD+M daje deweloperom lepsze narzędzie do pisania silniejszych (w sensie zdolności do wykrywania defektów) testów. Wyniki badań potwierdzają tym samym tezę badawczą nr 3: wzbogacenie podejścia TDD o komponent mutacyjny w istotny sposób podnosi charakterystyki jakościowe kodu.

Iteracyjne wprowadzanie mutacji w procesie wytwórczym oprogramowania znacznie podnosi zaufanie do kodu. Metodyka TDD nie jest jedyną, która może być wzbogacona o komponent mutacyjny. Inne metodyki, oparte o modele takie jak kaskadowy, spiralny, przyrostowy itp. też mogą być skutecznie wzbogacane o element testów mutacyjnych.

Ponadto, samo testowanie mutacyjne zastosowane w tych eksperymentach pokazało nam także, że za jego pomocą można wykrywać również takie rzeczy jak poprawność implementacji obliczeń

[13] Wynika to z porównania pokrycia dla grup MA i TA opisanego w tabeli 8.22

[14] Wynika to z porównania pokrycia dla grup MT i TT oraz dla grup MM i TM opisanego w tabeli 8.22

[15] Testowanie mutacyjne sprzyja pisaniu właśnie takich testów, które analizują przypadki możliwych błędów symulowanych przez zmutowany kod.

oraz czytelność przepływu sterowania (logikę biznesową), co może skutkować upraszczaniem przepływu lub eliminacją niepotrzebnych krawędzi w grafie przepływu sterowania.

Skoro model TDD+M okazał się skuteczny w podnoszeniu jakości testów (a zarazem oprogramowania) w grupach programistycznych o poziomie odpowiadającym młodszemu programiście, to w rękach doświadczonych programistów narzędzia mutacyjne mogą stanowić potężne narzędzie i stanowić nieocenioną pomoc w skutecznym prowadzeniu projektu. Ponadto, podejście TDD+M stanowi remedium na psychologiczne zjawisko stronniczości (tzw. bias) programistów i pozwala do pewnego stopnia rozwiązać problem niskiej niezależności testowania. Programista, jako autor kodu, jest przekonany o jego poprawności. Gdy musi testować własny kod, często, nawet w sposób podświadomy, projektuje testy tak, by potwierdzały jego własne założenia o tym kodzie, które wykorzystywał podczas implementacji. Testowanie mutacyjne w sposób niezależny i obiektywny jest w stanie wskazać mu słabości jego własnego kodu, tym samym zwiększając poziom niezależności testowania.

Dane z pierwszego eksperymentu opisanego w tym rozdziale były prezentowane na konferencji TestWarez 2017 [91].

8.7 Identyfikacja zagrożeń dla poprawności badań

W obu powyżej wspomnianych eksperymentach poprawność wyników mogła być zaburzona ze względu na zdolności programistyczne członków eksperymentu. W pierwszym eksperymencie ryzyko niższej jakości kodu jest wyższe, ponieważ uczestnicy eksperymentu nie pracowali jeszcze zawodowo jako programiści. Ponadto, w eksperymencie pierwszym wzięło udział jedynie 8 studentów pogrupowanych w dwa zespoły. Podczas przeprowadzania drugiego eksperymentu praktycznie wszystkie osoby biorące w nim udział pracowały już zawodowo jako programiści, a eksperyment był wykonany na większej liczbie osób, co skutkuje większą wiarygodnością wyników.

8.7.1 Trafność zewnętrzna (external validity)

Oba eksperymenty były kontrolowane i przeprowadzone jako tzw. 'porównanie grup statycznych' (ang. static group comparison). Jest to dwugrupowy układ eksperymentalny, w którym jedna grupa jest poddana działaniu badanego czynnika (używanie podejścia mutacyjnego), a wyniki porównywane są z grupą temu czynnikowi niepoddaną (używanie klasycznej metodyki TDD). W takim układzie eksperymentalnym możliwymi czynnikami zagrażającymi poprawności badań są głównie selektywność (ang. selection) oraz śmiertelność (ang. mortality).

Śmiertelność jest oczywiście wykluczona, gdyż w obu eksperymentach skład wszystkich grup pozostawał niezmienny od początku do końca. Natomiast w przypadku selektywności może istnieć pewne zagrożenie dla poprawności badań. Ponieważ w obu eksperymentach występowała mała liczba grup, wybrane grupy mogły być rzeczywiście zróżnicowane zanim nastąpiło pisanie kodu z bądź bez użycia testów mutacyjnych. Z drugiej strony, co najmniej w przypadku dwóch czynników poddawanych samoocenie uczestników (umiejętności programistyczne i umiejętności testerskie) wariancja między grupami w eksperymencie drugim nie wydaje się być zbyt duża.

Chociaż liczba pomiarów w eksperymencie drugim była wystarczająca do uzyskania wiarygodnych statystycznie wyników, eksperyment był wykonany jedynie na jednym, niewielkim projekcie. Programistami byli studenci studiów licencjackich z informatyki, a nie zawodowi deweloperzy. Zatem nie możemy uogólniać wyników eksperymentu na *dowolny* projekt, przy którym pracują ludzie o *dowolnym* poziomie doświadczenia zawodowego. Z drugiej strony, wysoka istotność statystyczna różnic pomiędzy podejściami TDD i TDD+M może sugerować uprawnienie do ostrożnej generalizacji, przynajmniej w jakimś stopniu.

Wybrana dziedzina biznesowa problemu (obliczenia numeryczne w postaci operacji na macierzach) dobrze wpasowuje się w podejście mutacyjne. Otrzymane wyniki mogły być w części wynikiem doboru problemu do eksperymentu. Wyniki te mogą wyglądać inaczej w projektach innego typu.

8.7.2 Trafność wewnętrzna (internal validity)

Studenci z eksperymentów 1 i 2 tworzyli rozłączne zbiory uczestników, zatem efekt interakcji nie był obecny (ten czynnik może zagrozić poprawności badań, gdyż pretest może zmniejszyć bądź zwiększyć responsywność/wrażliwość osoby badanej wobec zmiennej eksperymentalnej [92] – w naszym przypadku osoba, która brałaby udział w pierwszym eksperymencie, mogłaby w wyniku tego faktu zachowywać się inaczej w eksperymencie drugim).

Jednakże, w obu eksperymentach studenci pracowali w zespołach (w parach lub większych grupach), co może wprowadzić nowy czynnik, który nie był kontrolowany podczas przeprowadzania eksperymentu. Mógł więc wpłynąć na obserwowane wyniki. To zagrożenie jest minimalizowane, jeśli rozważamy wyniki na poziomie zespołu, nie indywidualnych uczestników.

Kod mierzony był tylko za pomocą dwóch prostych metryk: pokrycia instrukcyjnego i pokrycia mutacyjnego. Choć wiadomo powszechnie, że czynniki te są silnie skorelowane z jakością kodu, należy pamiętać, że pojęcie jakości (zwłaszcza jakości zewnętrznej) jest o wiele bardziej skomplikowanym i wielowymiarowym terminem. Nie możemy zatem traktować wyników eksperymentu jako odpowiedzi na pytanie o bezpośredni wpływ podejścia TDD+M zewnętrzną jakość kodu – wyniki te co najwyżej pokazują bezpośredni wpływ TDD+M na określone parametry kodu, z którego można dopiero pośrednio wnioskować o jakości zewnętrznej, pamiętając, iż czynność ta może być obarczona pewnym błędem.

Podczas drugiego eksperymentu mierzono również liczbę wykrytych defektów. Jednak ze względu na małą liczbę zarówno zespołów jak i wykrytych defektów, nie możemy konkluzywnie stwierdzić istotnej różnicy w jakości kodu pomiędzy zespołami TDD a TDD+M. Możemy jedynie porównać te podejścia ze względu na wprost zastosowaną metrykę liczby defektów.

Wybór osób do grup był przeprowadzony losowo, co jest sposobem na uniknięcie zjawiska „wyboru obiektów” (ang. selection of subjects), którego wystąpienie może zagrozić wewnętrznej poprawności. Jednakże, ze względu na mały rozmiar grup (w eksperymencie drugim było to osiem dwu- lub trzyosobowych zespołów) randomizacja może prowadzić do znanego paradoksu

Simpsona[16].

Wyniki obu eksperymentów mogły być również zaburzone ze względu na fakt, iż uczestnicy nie byli ekspertami w zakresie wytwarzania oprogramowania. Ta uwaga odnosi się zwłaszcza do eksperymentu pierwszego, ponieważ jego uczestnicy nie mieli w ogóle uprzedniego doświadczenia zawodowego w zakresie profesjonalnego wytwarzania oprogramowania. W eksperymencie drugim prawie wszyscy studenci mieli już jakieś tego typu doświadczenie, choć siłą rzeczy nie mogło być ono zbyt duże.

Innym czynnikiem, który mógł zagrozić poprawności wewnętrznej jest tzw. dojrzewanie (ang. maturation). Jeśli eksperyment trwa długi czas, jego uczestnicy mogą w trakcie jego trwania doskonalić swoje umiejętności niezależnie od wpływu badanego czynnika. W naszym przypadku byłoby to ulepszanie umiejętności programistycznych i testerskich niezależnie od stosowania testów mutacyjnych. Studenci pracowali nad swoimi projektami przez 3 tygodnie. Można więc ostrożnie uznać, iż nie był to czas na tyle długi, aby ewentualny efekt dojrzewania mógł znacząco wpłynąć na wyniki eksperymentu.

Zaburzenie wyników mogło być również spowodowane przez to, że niektórzy uczestnicy badania nie podążali dokładnie za wyznaczonymi im instrukcjami, np. nie oprogramowali dostarczonych im szablonów interfejsów tak, jak tego oczekiwano. Z powodu zmian w niektórych szablonach, dokonanych przez samych uczestników, było koniecznym dodanie pewnych setterów lub getterów dla pewnych parametrów klas, aby umożliwić przeprowadzenie krzyżowego eksperymentu testowania obcego kodu za pomocą własnych testów. W ten sposób, poprawiony kod skutkował generacją kilku dodatkowych mutantów. Z drugiej strony, we wszystkich tego typu przypadkach mutanty te okazały się mutantami trywialnymi i zawsze były wykrywane i zabijane.

W celu weryfikacji tezy 3.3 nie można było wykorzystać formalnych narzędzi statystycznych, ze względu na mały rozmiar badanej próby (liczba grup oraz sumaryczna liczba znalezionych defektów). Wyniki mogły być zaraportowane jedynie w ich pierwotnej, surowej formie.

[16]Paradoks Simpsona polega na odwróceniu efektu dla kilku grup, gdy grupy te są połączone w jedną

Rozdział 9

System S.A.M. - Symultanic Automatic Mutation System

S.A.M. jest systemem służącym do automatyzowania i zarządzania procesem testowania mutacyjnego. Implementacja rdzenia systemu S.A.M została wykonana przez autora niniejszej rozprawy. Przez rdzeń systemu S.A.M. rozumie się wszystkie moduły mutujące, konfiguracyjne oraz dostarczające możliwość komunikacji sieciowej wraz z protokołami i architekturą sieciową. Platforma została wykonana głównie na potrzeby przeprowadzenia eksperymentów opisanych w poprzednich rozdziałach pracy. Może być jednak również używana w rzeczywistych projektach wytwarzania oprogramowania, dla których zachodzi potrzeba przeprowadzania efektywnego procesu testowania mutacyjnego.

System S.A.M. może współpracować z dowolnym językiem programowania, językiem bazodanowym czy innym językiem pod warunkiem dostępności oprogramowania realizującego testowanie mutacyjne dla danego języka oraz dostępności oprogramowania zbierające statyczne metryki kodu. Na potrzeby badań został wybrany język Java, który jest popularnym językiem wysokopoziomowym z łatwo i szeroko dostępną bazą open source'owych projektów nadających się do przeprowadzania eksperymentów związanych z zastosowaniem testowania mutacyjnego.

Implementacja systemu S.A.M. wykorzystywana w zawartych w pracy badaniach używa zmodyfikowanego jądra systemu PIT [44] jako biblioteki do mutacji kodu. Z programu PIT zostały wyekstrahowane elementy generujące mutanty oraz część elementów raportujących, a następnie zostały one przekształcone w spójną formę biblioteki. Na potrzebę badań związanych z tą pracą został również wyekstrahowany z systemu PIT rdzeń odpowiadający za mutacje kodu i uruchamianie testów. Następnie wprowadzono szereg modyfikacji poszerzających standardowe możliwości systemu PIT. Modyfikacje te dotyczyły takich kwestii jak:

- ustawienie prawdopodobieństwa losowania danego mutanta,
- włączanie i wyłączanie testów,
- dodanie modyfikacji opartych na optymalizacji Mutation Churn Model opisanej w rozdziale 6,

- modyfikację modelu losowania mutantów tak, by umożliwić losowanie z wykorzystaniem bayesowkiego modelu prawdopodobieństwa opisanego w rozdziale 7.

Zmodyfikowano również sposób konfigurowania i komunikacji z PIT na potrzeby systemu S.A.M. i dokonano niezbędnych zmian związanych z przystosowaniem PIT do działania w środowisku rozproszonym. Dostęp do funkcjonalności systemu PIT, który do tej pory odbywał się poprzez szereg zintegrowanych pluginów środowisk budujących dla Javy został zastąpiony uniwersalną wtyczką konsolową.

Modyfikacja i dostosowanie systemu PIT na potrzeby badań wymagały bardzo dużych nakładów pracy. Proces modyfikacji trwał ponad rok.

9.1 Opis narzędzi do testowania mutacyjnego

PIT nie jest aktualnie jedynym powszechnie dostępnym oprogramowaniem do testów mutacyjnych. Inne tego typu rozwiązania to:

- MuJava [46] — platforma opracowana przez Jeffa Offutta. Jest to oprogramowanie badawcze przeznaczone do mutacji programów napisanych w Javie. Posiada opcję generowania raportów na podstawie wykonanych testów mutacyjnych. Niestety, raporty te są udostępniane w formie okienkowej, co nie daje łatwej możliwości eksportu danych do innej postaci. Na chwilę obecną oprogramowanie nie jest dalej wspierane przez twórców (od ponad 3 lat), przez co cechuje się sporą liczbą niepoprawionych defektów. W najnowszej dostępnej wersji MuJava nie daje niestety możliwości sprzęgnięcia z Mavenem[1], Antem[2] ani Gradlem[3]. Narzędzie dobrze nadaje się do prowadzenia badań na małych lub nieskomplikowanych projektach.

- Major mutation framework [40] — opracowany przez Justa, Kurtza i Ammanna. Oprogramowanie rozwijane od 2011 roku. Jest to bardzo ciekawy system, który wprowadza szereg optymalizacji związanych z eliminacją równoważnych operatorów mutacyjnych. Jest także wyposażony w możliwość przedefiniowania dokładnego sposobu mutacji poprzez skrypty mutacyjne. O optymalizacjach zawartych w systemie Major można przeczytać więcej w rozdziale 4.

- Jumble [93] — narzędzie mutacji dla programów napisanych w Javie, testowanych za pomocą JUnit. Podobnie jak w PIT mutacje wykonywane są poprzez modyfikację bajtkodu klas programu. Program może być uruchamiany z linii poleceń oraz jako wtyczka do środowiska Eclipse.

- MoMut [23] — system przeznaczony do przeprowadzania mutacyjnego testowania oprogramowania, pozwalający między innymi na mutacje procesów biznesowych zdefiniowanych

[1] Apache Maven to narzędzie automatyzujące budowę oprogramowania na platformę Java w środowisku Continuous Integration. Oprogramowanie opiera się na konfigurowalnych plikach wsadowych w formacie XML.

[2] Apache Ant to już przestarzałe, lecz dalej stosowane narzędzie budujące oprogramowanie na platformę Java. Obecnie zastąpione przez Apache Maven oraz Gradle.

[3] Gradle to jedno z najnowszych narzędzi stosowanych do budowania programów na platformę Java. Bezpośredni konkurent platformy Apache Maven. Obecnie zdobywa coraz większą popularność.

przy użyciu języka UML. Więcej o mutacji procesów biznesowych można przeczytać w rozdziale 3.

Powyżej zostały zaprezentowane najciekawsze z projektów umożliwiające przeprowadzanie testów mutacyjnych oprogramowania. Inne programy tego typu to:

- dla Javy: Javalanche, JesTer,
- dla C++: Plextest,
- dla Pythona: PesTer,
- dla .Net: VisualMutator.

Wymienione wyżej programy posiadają wiele interesujących cech, lecz na potrzeby badań związanych z tą pracą został wybrany system PIT, który okazał się najłatwiejszy do modyfikacji oraz bardzo efektywny pod względem organizacji procesu mutowania oprogramowania. Początkowo planowane było bazowanie na platformie MuJava oraz dalszy jej rozwój. Jednakże wraz z postępem prac okazało się, że system MuJava nie spełnia założeń koniecznych do dalszego prowadzenia badań przy jego użyciu. Ponadto, w pracy [19] wykazano przewagę systemu PIT nad MuJava.

9.1.1 Struktura pokrycia kodu testami w PIT

Zgodnie z dokumentacją systemu PIT [44] pokrycie testami oraz nadawanie pierwszeństwa testom (tzn. priorytetyzacja ich wykonania) opiera się na trzech czynnikach:

- pokrycie linii,
- czas wykonania testu,
- konwencja nazewnictwa testów[4].

Dla każdego pojedynczego przypadku testowego wykonywana jest analiza określająca jego zasięg. Następnie, dla każdej zmutowanej linii kodu określane jest, czy dany test ją pokrywa czy nie. Testy, które nie korzystają ze zmutowanej linii kodu są odrzucane. Pozostałe testy są porządkowane zgodnie z czasem ich wykonania, który jest głównym czynnikiem określającym kolejność wykonania testów. Kolejnym czynnikiem porządkującym jest nazewnictwo testu. Pierwszeństwo nad innymi testami będą miały te testy, których nazywa jest taka sama, jak nazwa zmutowanej klasy[5]. Dokładny opis priorytetyzacji testów jest przedstawiony w rozdziale 4.6.

[4] Faworyzowanie testów zawierających nazwę klasy

[5] Klasa zawierająca test jest uważana za test jednostkowy dla konkretnej klasy, jeśli jest zgodna ze standardową konwencją nazewnictwa JUnit dla FooTest lub TestFoo. W przeciwieństwie do wcześniejszych systemów PIT nie wymaga zgodności z tą konwencją nazewnictwa, aby testy mogła działać. Nazwy testów są używane tylko jako część heurystyki, aby zoptymalizować kolejność wykonywania.

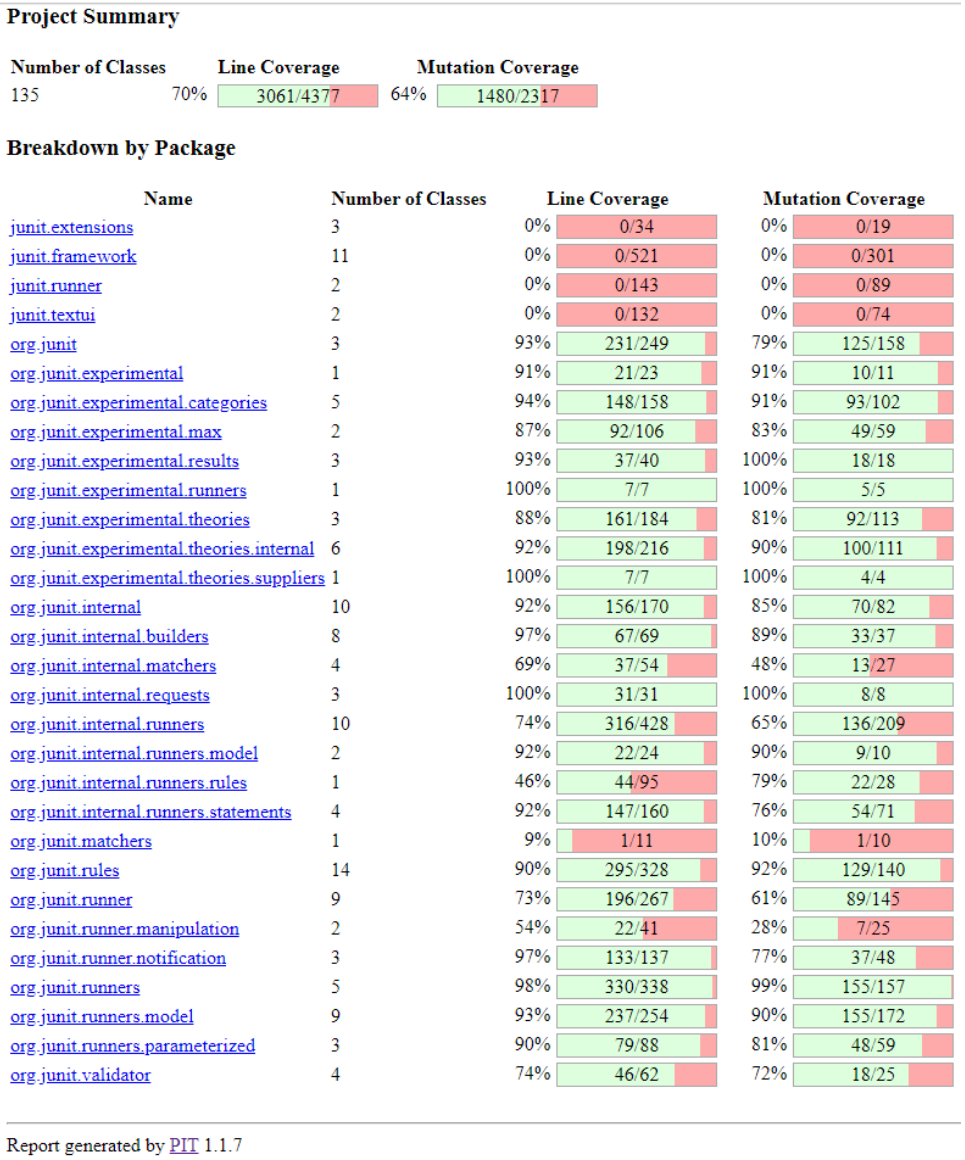
Główną klasą odpowiadającą za generację mutantów i przeprowadzanie testów jest `MutationCoverage`. Klasa znajduje się w jądrze systemu PIT[6]. Pokrycie kodu testami jest przechowywane w strukturze `coverageData`. Struktura trzymająca w sobie pokrycie testami jest postaci test—blok kodu. Wszystkie mutanty przechowywane są w specjalnej strukturze o charakterze listy generowanej na niskim poziomie z bajtkodu. Struktura zawiera dowiązanie do mutowanego miejsca w kodzie oraz listę potencjalnych mutantów.

Raporty z funkcjonowania programu PIT są zwracane w formie konsolowej zaraz po zakończeniu procesu mutacji oraz w formie plików html lub xml. Zrzut ekranu konsoli przedstawiono na rysunku 9.1. Przykładowy raport html znajduje się na rysunkach 9.2 oraz 9.3. Na tym ostatnim przedstawiony jest zrzut ekranu z fragmentu raportu szczegółowego, na którym można obserwować analizę poszczególnych bloków kodu.

```
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : 9 seconds
> build mutation tests : 1 seconds
> run mutation analysis : 6 minutes and 19 seconds
--------------------------------------------------------------------------------
> Total  : 6 minutes and 30 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Generated 2273 mutations Killed 1437 (63%)
>> Ran 33790 tests (14.87 tests per mutation)
================================================================================
- Mutators
================================================================================
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator
>> Generated 78 Killed 47 (60%)
> KILLED 47 SURVIVED 5 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 26
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 60 Killed 39 (65%)
> KILLED 39 SURVIVED 4 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 17
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.VoidMethodCallMutator
>> Generated 586 Killed 291 (50%)
> KILLED 287 SURVIVED 71 TIMED_OUT 4 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 224
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
>> Generated 808 Killed 549 (68%)
> KILLED 546 SURVIVED 34 TIMED_OUT 3 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 225
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 61 Killed 23 (38%)
> KILLED 23 SURVIVED 9 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 29
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 680 Killed 488 (72%)
> KILLED 482 SURVIVED 24 TIMED_OUT 6 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 168
--------------------------------------------------------------------------------
Sending results: PitResults [htmlResultFile=G:\Workspace2\.metadata\.plugins\org.p
Closing server
Closed
```

Rysunek 9.1: PIT raport w formie konsolowej dla programu JUnit 4.12

[6]Ta klasa, wraz z klasami z nią powiązanymi tworzą moduł mutacyjny systemu S.A.M.



Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
135	70%	3061/4377	64%	1480/2317

Breakdown by Package

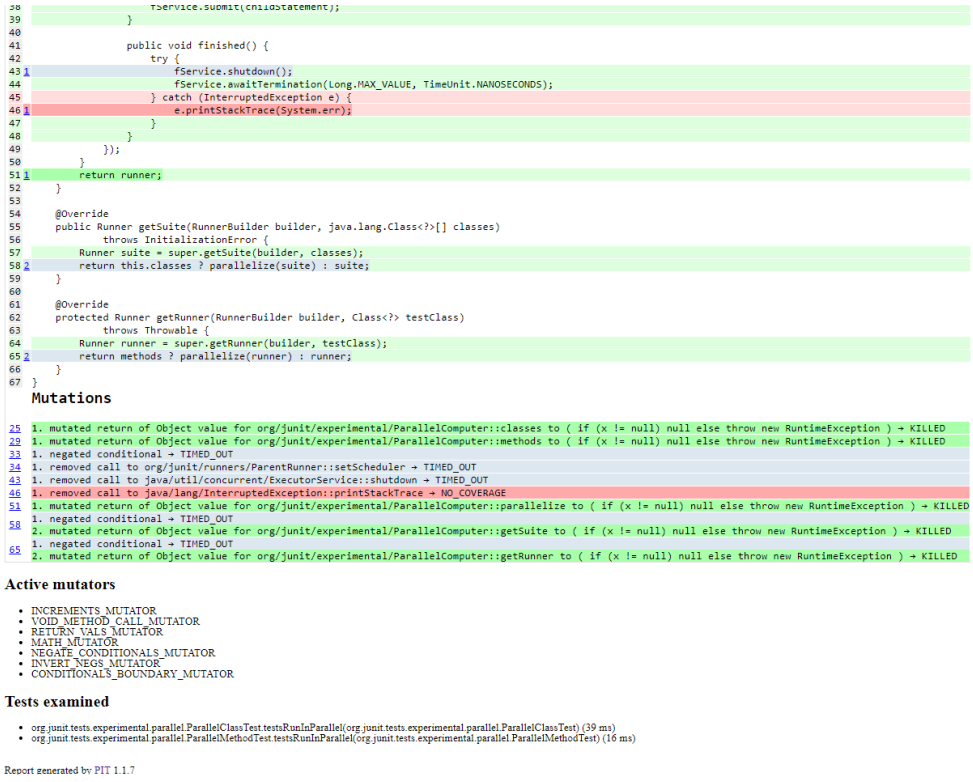
Name	Number of Classes	Line Coverage		Mutation Coverage	
junit.extensions	3	0%	0/34	0%	0/19
junit.framework	11	0%	0/521	0%	0/301
junit.runner	2	0%	0/143	0%	0/89
junit.textui	2	0%	0/132	0%	0/74
org.junit	3	93%	231/249	79%	125/158
org.junit.experimental	1	91%	21/23	91%	10/11
org.junit.experimental.categories	5	94%	148/158	91%	93/102
org.junit.experimental.max	2	87%	92/106	83%	49/59
org.junit.experimental.results	3	93%	37/40	100%	18/18
org.junit.experimental.runners	1	100%	7/7	100%	5/5
org.junit.experimental.theories	3	88%	161/184	81%	92/113
org.junit.experimental.theories.internal	6	92%	198/216	90%	100/111
org.junit.experimental.theories.suppliers	1	100%	7/7	100%	4/4
org.junit.internal	10	92%	156/170	85%	70/82
org.junit.internal.builders	8	97%	67/69	89%	33/37
org.junit.internal.matchers	4	69%	37/54	48%	13/27
org.junit.internal.requests	3	100%	31/31	100%	8/8
org.junit.internal.runners	10	74%	316/428	65%	136/209
org.junit.internal.runners.model	2	92%	22/24	90%	9/10
org.junit.internal.runners.rules	1	46%	44/95	79%	22/28
org.junit.internal.runners.statements	4	92%	147/160	76%	54/71
org.junit.matchers	1	9%	1/11	10%	1/10
org.junit.rules	14	90%	295/328	92%	129/140
org.junit.runner	9	73%	196/267	61%	89/145
org.junit.runner.manipulation	2	54%	22/41	28%	7/25
org.junit.runner.notification	3	97%	133/137	77%	37/48
org.junit.runners	5	98%	330/338	99%	155/157
org.junit.runners.model	9	93%	237/254	90%	155/172
org.junit.runners.parameterized	3	90%	79/88	81%	48/59
org.junit.validator	4	74%	46/62	72%	18/25

Report generated by PIT 1.1.7

Rysunek 9.2: PIT – raport html dla programu JUnit 4.12

9.2 Mechanika funkcjonowania systemu S.A.M

Opis przypadku użycia systemu S.A.M. przedstawiono na rys. 9.4. System S.A.M. może być uruchamiany ręcznie lub za pomocą narzędzi zewnętrznych. W obu przypadkach dane wsadowe mogą być generowane bezpośrednio z linii komend lub pośrednio poprzez plik konfiguracyjny. System S.A.M. na końcu procesu zwraca wynik w postaci raportów dla użytkownika. Raport z pracy procesu testowania mutacyjnego system S.A.M zwraca w taki sam sposób jak system PIT, czyli w postaci pliku html lub XML. Opis konfigurowania systemu oraz modułów znajduje się w dodatku C.



```
                    fService.submit(childStatement);
                }

                public void finished() {
                    try {
1                       fService.shutdown();
                        fService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
                    } catch (InterruptedException e) {
1                       e.printStackTrace(System.err);
                    }
                }
            });
        }
1       return runner;
    }

    @Override
    public Runner getSuite(RunnerBuilder builder, java.lang.Class<?>[] classes)
            throws InitializationError {
        Runner suite = super.getSuite(builder, classes);
2       return this.classes ? parallelize(suite) : suite;
    }

    @Override
    protected Runner getRunner(RunnerBuilder builder, Class<?> testClass)
            throws Throwable {
        Runner runner = super.getRunner(builder, testClass);
2       return methods ? parallelize(runner) : runner;
    }
}
```

Mutations

```
25  1. mutated return of Object value for org/junit/experimental/ParallelComputer::classes to ( if (x != null) null else throw new RuntimeException ) → KILLED
29  1. mutated return of Object value for org/junit/experimental/ParallelComputer::methods to ( if (x != null) null else throw new RuntimeException ) → KILLED
33  1. negated conditional → TIMED_OUT
34  1. removed call to org/junit/runners/ParentRunner::setScheduler → TIMED_OUT
43  1. removed call to java/util/concurrent/ExecutorService::shutdown → TIMED_OUT
46  1. removed call to java/lang/InterruptedException::printStackTrace → NO_COVERAGE
51  1. mutated return of Object value for org/junit/experimental/ParallelComputer::parallelize to ( if (x != null) null else throw new RuntimeException ) → KILLED
58  1. negated conditional → TIMED_OUT
    2. mutated return of Object value for org/junit/experimental/ParallelComputer::getSuite to ( if (x != null) null else throw new RuntimeException ) → KILLED
65  1. negated conditional → TIMED_OUT
    2. mutated return of Object value for org/junit/experimental/ParallelComputer::getRunner to ( if (x != null) null else throw new RuntimeException ) → KILLED
```

Active mutators

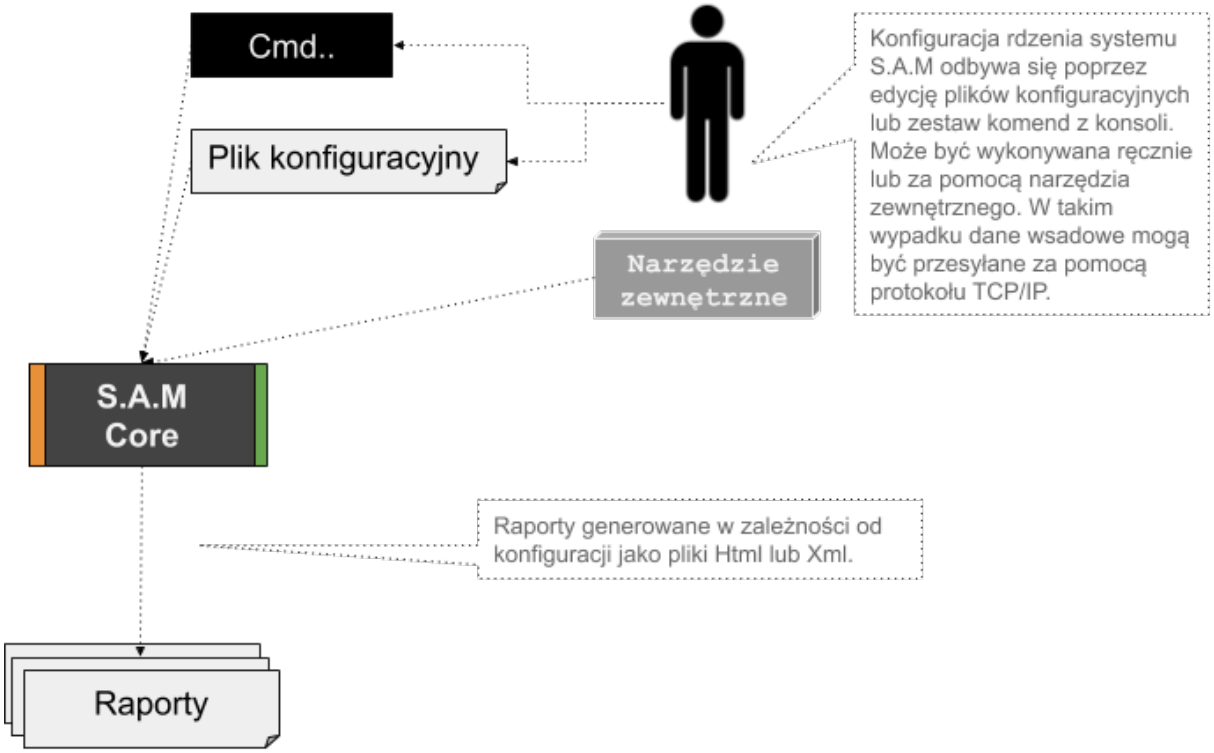
- INCREMENTS_MUTATOR
- VOID_METHOD_CALL_MUTATOR
- RETURN_VALS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- INVERT_NEGS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR

Tests examined

- org.junit.tests.experimental.parallel.ParallelClassTest.testsRunInParallel(org.junit.tests.experimental.parallel.ParallelClassTest) (39 ms)
- org.junit.tests.experimental.parallel.ParallelMethodTest.testsRunInParallel(org.junit.tests.experimental.parallel.ParallelMethodTest) (16 ms)

Report generated by PIT 1.1.7

Rysunek 9.3: PIT – szczegółowy raport html dla programu JUnit 4.12



Rysunek 9.4: Ogólny opis działania systemu S.A.M.

9.3 Architektura systemu S.A.M.

System S.A.M. jest oprogramowaniem wielomodułowym. Rdzeń systemu jest napisany w języku Java oraz został oparty o mechanizmy mutacji kodu zawarte w systemie PIT. Poniżej znajduje się krótki opis funkcjonalności poszczególnych modułów.

9.3.1 Moduły konfiguracyjne oraz rozdzielające zadania

Są to trzy moduły odpowiedzialne kolejno za:

- definiowanie roli poszczególnych instancji programu oraz przekierowywanie do nich danych konfiguracyjnych za pomocą plików wsadowych, bezpośrednio przez linię komend lub za pomocą komunikatów wysyłanych protokołem TCP/IP;

- estymację czasu wykonania poszczególnych zadań, a następnie optymalne rozdzielenie ich pomiędzy węzłami obliczeniowymi systemu;

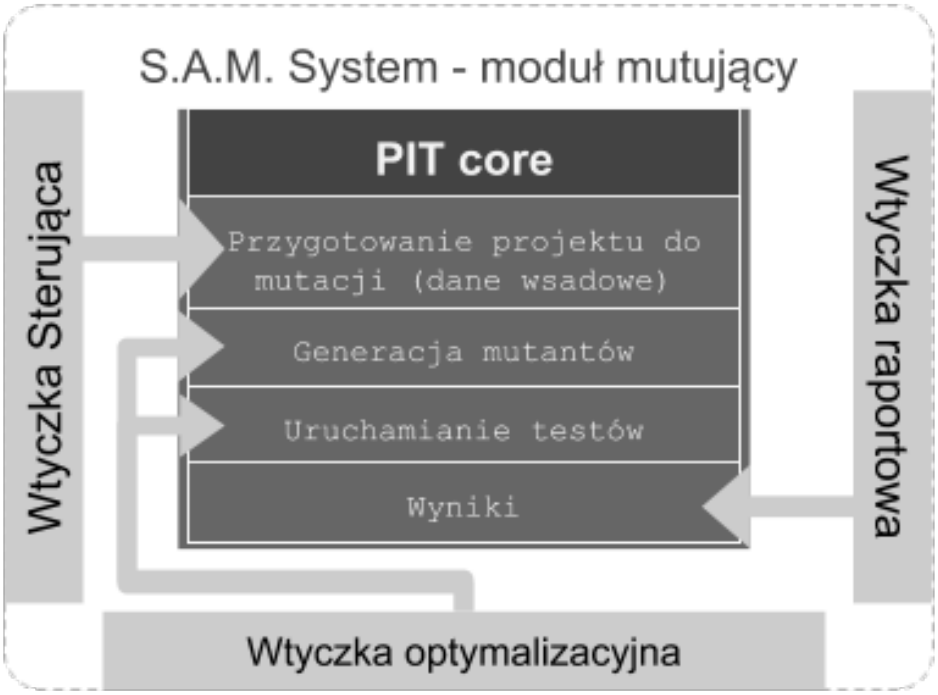
- agregację oraz zebranie raportów z wszystkich węzłów systemu – do tych celów używane są narzędzia zewnętrzne lub pluginy dodane do rdzenia systemu S.A.M.

9.3.2 Moduł testująco-mutujący

Głównym zadaniem modułu jest wykonanie testowania mutacyjnego określonego zadania na podstawie zadanej konfiguracji. Sam moduł rozszerzony jest również o funkcjonalności związane z kolejkowaniem zadań i generowaniem odpowiednich wyników. W fizycznej implementacji systemu istnieć będzie zazwyczaj wiele kopii tego modułu, po jednym dla każdego węzła wykonawczego.

Moduł jest oparty na jądrze systemu PIT, do którego zostały dodane wtyczki pozwalające na modyfikację sposobu, liczby oraz prawdopodobieństwa doboru mutantów. Ze względu na sztywną architekturę użytego jądra sterowanie wewnętrznym stanem mutacji i testowania jest oparte na wzorcu programistycznym `Singleton`[7]. Wzorzec ten został zastosowany poprzez wstrzyknięcie w wewnętrzne struktury systemu PIT obiektu, który na podstawie zadanej konfiguracji modyfikuje dane oraz przepływ pracy w systemie PIT. Diagram na rys. 9.5 pokazuje sposób, w jaki następuje manipulacja danymi oraz stanami jądra systemu PIT. Dodatkowo jądro PIT zostało zmodyfikowane w taki sposób, by przeprowadzać testowanie mutacyjne w oparciu o bayesowski model prawdopodobieństwa opisany dokładniej w rozdziale 9.4.

[7]Singleton to kreacyjny wzorzec projektowy, którego celem jest wymuszenie tworzenia tylko jednej instancji danej klasy.



Rysunek 9.5: Architektura komunikacji z jądrem systemu PIT

9.4 Funkcjonalność dla optymalizacji testowania

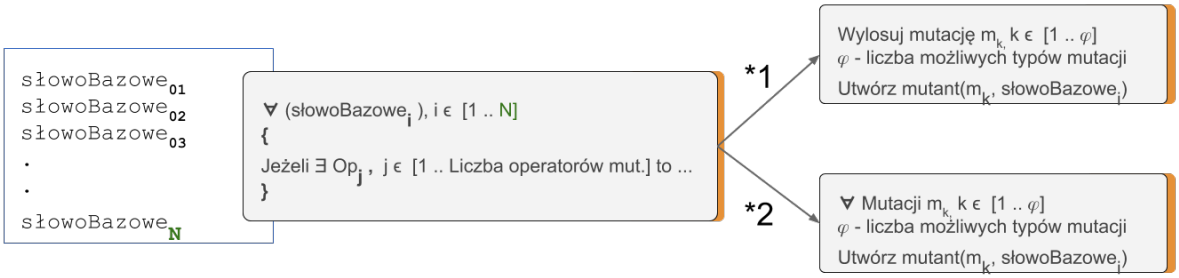
System S.A.M. zawiera zestaw funkcjonalności pozwalających na optymalizację procesu generowania mutantów oraz uruchamiania testów. Propaguje swoją logikę biznesową do modułu testująco-mutującego za pomocą mechanizmów opisanych w podrozdz. 9.3.2. Optymalizacje zaimplementowane w tym fragmencie systemu to:

- Dobór i generacja mutantów zgodnie z prawdopodobieństwem bayesowskim definiowanym dla każdego operatora mutacyjnego z osobna. Opis algorytmu losowania znajduje się w rozdziale 7.
- Dobór i generacja mutantów oparte o prawdopodobieństwo zadane z góry. W tym wypadku dla każdego operatora mutacyjnego możemy ustawić prawdopodobieństwo, z jakim będziemy decydować drogą losowania, czy dany mutant ma wejść do puli, dla której zostaną uruchomione testy czy też nie. Prawdopodobieństwa zadawane są poprzez plik konfiguracyjny. Mechanizm opisujący różnicę w generacji mutantów w standardowy sposób oraz zgodnie z prawdopodobieństwem opisuje diagram na rys. 9.6.
- Włączanie i wyłączanie operatorów mutacyjnych używanych w systemie. Ta optymalizacja pozwala na usunięcie z puli operatorów mutacyjnych zgodnie z listą zadaną w pliku konfiguracyjnym.
- Wykrywanie, czy klasy się zmieniły i mutowanie tylko klas, które zostały zmodyfikowane. Ta optymalizacja została zaproponowana po wykonaniu eksperymentów opisanych w rozdziale 6. Mechanizm, który został zastosowany do tej implementacji, opiera się na porównywaniu

liczby bitów kodu danej klasy z wersją z poprzedniej iteracji mutacji. Proces pomija białe znaki oraz komentarze. Optymalizacja nie zostanie zastosowana do klas, w których istnieją żywe mutanty lub gdy pojawi się nowy test. Ten moduł jest jeszcze w fazie wczesnego prototypu — w przyszłości planowane są badania z jego wykorzystaniem.

- Wymuszone wyłączenie pewnej puli testów. Proces ten odbywa się poprzez uzupełnienie listy z nazwami testów w odpowiednim pliku konfiguracyjnym. Zasada działania tej funkcjonalności jest wzorowana na systemie PIT.

Konfiguracja plików wsadowych do modułu optymalizującego jest opisana w dodatku C.



*1 opisuje proces generowanie mutantów na podstawie zadanego prawdopodobieństwa.
*2 Opisuje proces mutacji bez użycia losowania mutantów.

Rysunek 9.6: Różnice między generacją mutantów losową i zwykłą

9.5 Architektura sieciowa i przypadki użycia

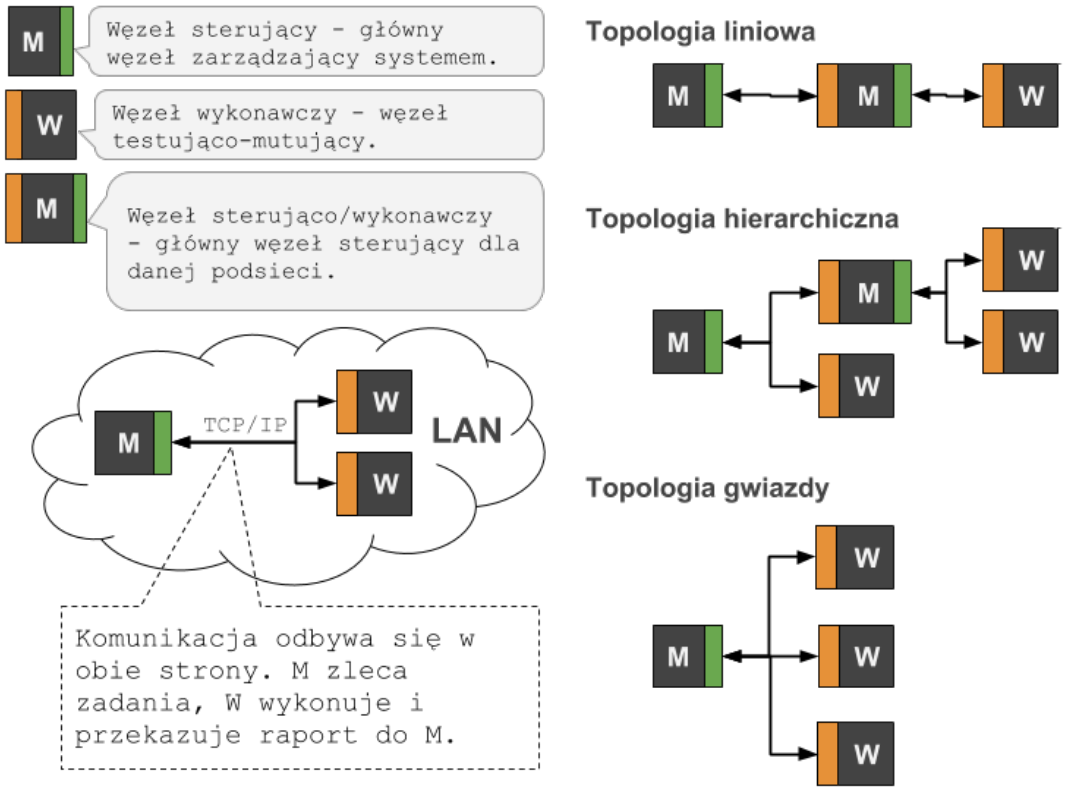
Proces tworzenia klastrów obliczeniowych opartych na systemie S.A.M. powinien być rozpatrywany pod kątem architektury sieci oraz przypadków użycia programu.

9.5.1 Architektura sieciowa

Implementacja mechanizmów sieciowych w systemie S.A.M. pozwala na replikację poszczególnych instancji programu oraz połączenia ich w sieć. Każdy instancja programu posiada funkcjonalności serwerowe i klienckie. Instancje porozumiewają się ze sobą za pomocą protokołu TCP/IP przesyłając miedzy sobą informacje sterujące, nowe zadania oraz raporty z wykonanej pracy. Transmisja danych w sieci odbywa się w architekturze rozgłoszeniowej typu *broadcast*[8]. Dla systemu S.A.M. mogą zostać zastosowane następujące topologie sieciowe[9]: topologia liniowa, topologia gwiazdy, topologia gwiazdy rozszerzonej, topologia hierarchiczna oraz wszystkie inne topologie oparte na grafach. Każda instancja S.A.M. może być łączona z innymi instancjami przy pomocy odpowiednich wtyczek, co daje w pełni modułową, rozproszoną, skalowalną oraz samo-adaptacyjną platformę mutującą. Dokładne informacje opisujące uruchamianie i konfigurację S.A.M. znajdują się w dodatku C. Przykładowe topologie systemu S.A.M. opisuje diagram na rys.

[8]Broadcast to rozsiewczy (rozgłoszeniowy) tryb transmisji danych polegający na wysyłaniu przez port (kanał informacyjny) informacji do wszystkich elementów sieci połączonych z węzłem rozgłoszeniowym.
[9]Topologia sieci to fizyczny opis architektury sieci. Topologia opisuje kształt sieci w terminach połączeń miedzy jej elementami oraz rodzaju tych połączeń.

9.7. Konfiguracja każdego węzła należącego do sieci odbywa się poprzez moduł konfiguracyjno-sterujący, który został opisany w punkcie 9.3.1.

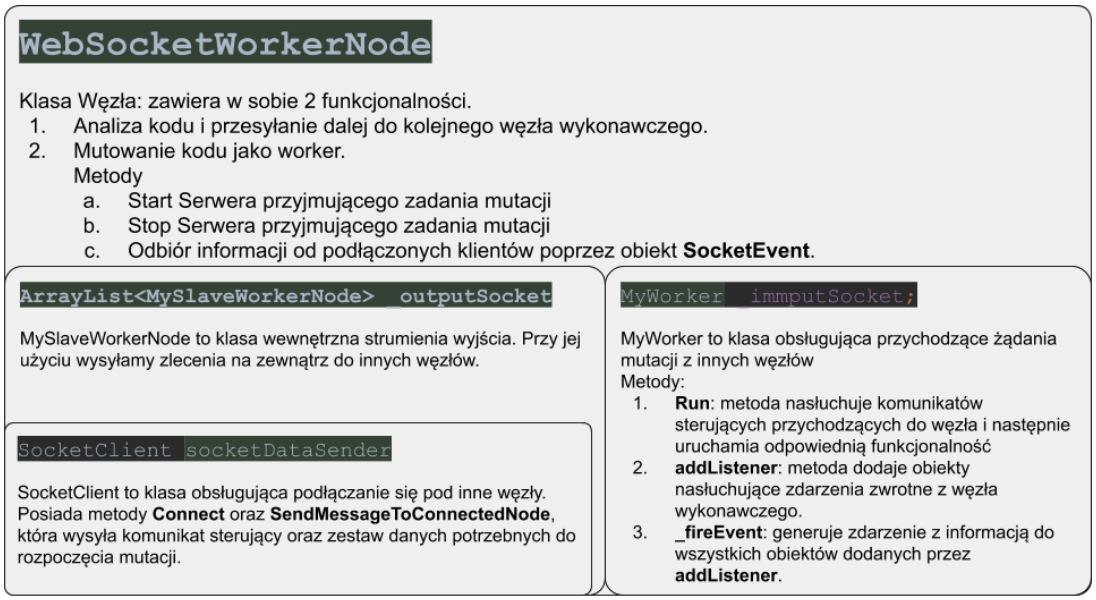


Rysunek 9.7: Diagram topologii obrazujący przykładowe topologie sieci, jakie można stosować dla systemu S.A.M.

W ten sposób zaprezentowana architektura sieci wyróżnia dwa typy węzłów, z czego jeden może posiadać dwie role (będzie to zależeć od zadanej topologii).

- **Węzeł sterujący**. Jest głównym wejściowym węzłem sieci, bądź węzłem rozdzielającym, definiującym podsieć. Głównymi zadaniami tego węzła są: rozdzielanie zadań do wykonania przez inne elementy sieci, zbieranie statystyk do predykcji oraz optymalizacja całościowego procesu mutacji i testowania dzięki zastosowaniu modułu optymalizującego. Zadania tego typu węzłów polegają na ogólnym przypadku użycia danego węzła, który jest opisany w punkcie 9.5.2.
- **Węzeł wykonawczy**. Fizycznie przeprowadza kompilację, generuje mutanty, uruchamia testy oraz zbiera dane statystyczne, a następnie odsyła je do węzła nadzorującego, propagując raporty do wyższych warstw sieci. Każdy węzeł wykonawczy może przyjmować i wykonywać innego rodzaju zadanie. Wszytko zależy od sposobu, w jaki zostanie on skonfigurowany. Proces konfiguracji został opisany w punkcie 9.5.2.

Mechanika komunikacji między węzłami systemu S.A.M. została opisana na diagramie przedstawionym na rys. 9.8.



Rysunek 9.8: Komunikacja między węzłami

9.5.2 Sposoby użycia systemu S.A.M.

Poprzez swoją w pełni konfigurowalną, rozbudowaną architekturę S.A.M. może dostosować się do różnych typów zadań związanych z jednostkowym testowaniem oprogramowania. W zależności od konfiguracji, w jakiej zostanie ustawiony, może spełniać następujące zadania.

1. Lekka, szybka i prosta aplikacja do wykonywania testów mutacyjnych oraz pokrycia kodu testami jednostkowymi. Jest to najprostsza konfiguracja systemu. Posiada ona tylko jedną instancję, która wykonuje testy dla projektu przekazanego poprzez wsad z linii komend, pliku bądź dowolnego innego narzędzia kompatybilnego z protokołem wsadowym. Możemy tu wydzielić dwie podkonfiguracje: mutującą oraz niemutującą.

- Konfiguracja mutująca analizuje projekt pod kątem pokrycia testami jednostkowymi, a następnie generuje mutanty we wszystkich miejscach, które są osiągane przez testy jednostkowe. Następną fazą jest testowanie mutantów. Dla każdego mutanta są uruchamiane testy z nim związane, a następnie generowany jest raport zawierający informacje o pokryciu kodu oraz o przeżywalności mutantów dla każdego z testów. W tej konfiguracji S.A.M. nie różni się niczym od PIT.
- Druga konfiguracja sprowadza S.A.M. do roli narzędzia badającego pokrycie kodu testami jednostkowymi. W tym wypadku kod nie jest poddawany mutacji i żadne mutanty nie są generowane. Taka konfiguracja może być przydatna dla deweloperów w implementacyjnych fazach procesu wytwarzania oprogramowania, pozwalając na szybką analizę pokrycia aktualnego kodu, a następnie wykrycie oraz uzupełnienie braków w testach jednostkowych dla istotnych elementów projektu. Na małą skalę jest to operacja szybka, jak i wydajna obliczeniowo, a z racji przejrzystości generowanych raportów pozwala zaoszczędzić czas na szukanie i analizowanie niepokrytych fragmentów kodu. S.A.M. dla tej konfiguracji może także zostać uruchomiony w trybie wielowątkowym.

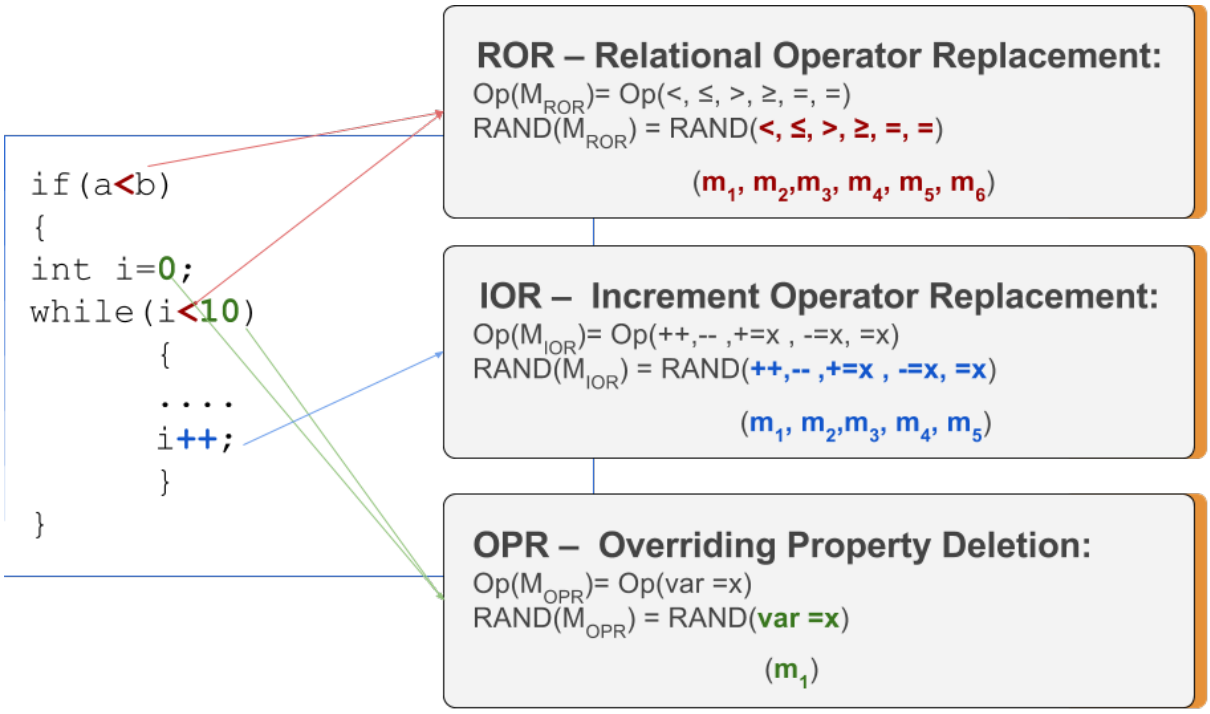
- Jako osobny typ konfiguracji wyróżniamy także usprawnienie mutujące polegające na możliwości wprowadzania poprzez plik konfiguracyjny opcji doboru konkretnych operatorów mutacyjnych oraz definiowania za pomocą skryptów mutacyjnych, jakie dokładnie mutanty chcemy, ażeby powstawały.

2. Jednoinstancyjna aplikacja do badań statystycznych związanych z mutacją kodu. W tym przypadku konfiguracja oprogramowania jest podobna do tej opisanej w punkcie 1., z tym że w tym wypadku zostaje uruchomiony także moduł statystyczny, który sprawia, że losowanie mutantów wykonywane jest zgodnie z pewnym rozkładem prawdopodobieństwa, wyznaczonym globalnie bądź dla każdego operatora mutacyjnego indywidualnie. Generowanie mutantów w ten losowy sposób niesie ze sobą korzyści obliczeniowe związane z mniejszą liczbą mutantów, dla których należy uruchomić testy. Jednak zgodnie z tym, co zostało napisane w rozdziale 7, zmniejszenie liczby mutantów poprzez losowanie tylko ich określonego podzbioru niesie ze sobą ryzyko związane z utratą istotnych danych o potencjalnych wadach projektu. Wobec tego stosowanie tej optymalizacji musi być wykonywane rozważnie. Dla tego sposobu użycia S.A.M. także możemy wydzielić dwie podkonfiguracje.

- Konfiguracja ze sztywno ustalonym prawdopodobieństwem, która każdego mutanta wylosuje zgodnie z prawdopodobieństwem globalnym bądź prawdopodobieństwem dla typu operatora mutacyjnego, do którego ten mutant należy.

- Prawdopodobieństwo wyznaczane w sposób uczący się, przy zastosowaniu podejścia bayesowskiego opisanego w rozdziale 7. Podobnie jak we wcześniejszym przypadku deklaruje się początkowe wartości prawdopodobieństw generacji mutantów. Następnie, dla każdej klasy bądź zestawu klas zostaje przeprowadzony proces generacji mutantów zgodnie z zadanym prawdopodobieństwem oraz uruchamianie testów. Na podstawie wyników testów oraz wykrywalności mutantów odbywa się wnioskowanie statystyczne. Proces ten odbywa się właśnie poprzez bayesowski mechanizm uczący, który w zależności od liczby wykrytych żywych mutantów dla danego operatora mutacyjnego podniesie bądź obniży prawdopodobieństwo losowania tego mutanta w najbliższej iteracji. Cały proces jest szerzej opisany w rozdziale 7. Dokładną pulę mutantów przeznaczonych do losowania można zdefiniować poprzez plik konfiguracyjny.

Obie konfiguracje generują takie same raporty jak te w punkcie 1. Użycie mechanizmów uczących generuje dodatkowy raport. Więcej informacji o raportach znajduje się w dodatku C wraz z instrukcją obsługi systemu. Na rys. 9.9 przedstawiony został w postaci diagramu opis algorytmu mutującego oraz mechanizm losowania mutantów. W celu lepszego zrozumienia diagramu pomocne będą definicje z podrozdziału 3.3. Podczas wykonywania procedury losowania mutant nie zawsze musi być wylosowany, co jest zgodne z przytoczonym modelem optymalizacyjnym.



Rysunek 9.9: Przykładowe użycie algorytmu losowania

3. Rozproszony wielowęzłowy samoadaptacyjny system mutacyjny. W tej konfiguracji S.A.M. jest zestawem porozumiewających się ze sobą instancji samego siebie, tworzących sieć węzłów mutująco-raportujących. Komunikacja między węzłami odbywa się za pośrednictwem protokołu TCP/IP. Każdy węzeł w sieci implementuje funkcjonalności serwerowe, jak i klienckie. Główny węzeł wejściowy za pomocą narzędzi analitycznych oraz wtyczki predykcyjnej opisanej w pracy [2][10] rozdziela prace na węzły jemu bezpośrednio podległe, zgodnie z deklarowaną przez nie mocą obliczeniową. Wobec powyższego, węzeł może spełniać dwie role: mutującą bądź nadzorującą.

- Rola mutująca (worker/robotnik). W tej roli węzeł deklaruje węzłowi nadrzędnemu swoją moc obliczeniową oraz oczekuje na zadania od niego. Po wykonaniu prac informuje nadzorcę o tym, że jest wolny oraz odsyła raport z wykonanego zadania. Każdy robotnik jest instancją systemu S.A.M. z uruchomioną wtyczką pozwalającą na odbieranie danych po sieci oraz raportowanie wyników pracy do węzła, który mu te dane wysłał.

- Rola nadzorująca (master/rozdzielacz). Węzeł nadzorujący wraz z węzłami robotnikami tworzy podsieć stanowiącą pewnego rodzaju klaster, który deklaruje swoją moc obliczeniową węzłowi nadrzędnemu, a następnie przyjmuje zestaw klas przeznaczonych do mutacji. W kolejnym kroku rozdziela prace po robotnikach wewnątrz swojej podsieci. Raport z testów przekazywany jest od robotników z powrotem do rozdzielacza, a następnie do wyższych warstw sieci bądź bezpośrednio do użytkownika. Każda podsieć zawiera również wtyczkę

[10]Prace dotyczące predykcji czasu wykonania mutacji nie wchodzą w zakres niniejszej rozprawy – za tę część badań odpowiedzialny był współautor systemu, Piotr Wawrzyniak.

predykcyjną pozwalającą na optymalne rozdzielenie pracy na węzły robotnicze. Rozdzielacz może przyjmować swoje zadania bezpośrednio od użytkownika — wtedy jest serwerem.

Topologie sieci węzłów możliwe do wygenerowania są przedstawione na diagramie z rys. 9.7.

9.6 Podsumowanie

System S.A.M. obecnie jest systemem prototypowym, zastosowanym głównie do wykonywania eksperymentów na potrzeby badań opisanych w niniejszej pracy. Dzięki swojej elastyczności wynikającej z wielomodułowej architektury może być z powodzeniem dalej rozwijany do postaci w pełni funkcjonalnego narzędzia. Także dodawanie kolejnych funkcjonalności do systemu nie stanowi żadnego problemu. Platforma S.A.M może po udoskonaleniach i modyfikacjach znaleźć swoje zastosowanie w przemyśle związanym z wytwarzaniem oprogramowania. Dokumentacja techniczna systemu S.A.M. znajduje się w dodatku C

Rozdział 10

Wnioski, podsumowanie

Celem niniejszej pracy było przeanalizowanie i zbadanie procesu oraz roli testowania mutacyjnego w procesach wytwórczych oprogramowania, a także zaproponowanie i wprowadzenie możliwych ulepszeń optymalizujących proces testowania mutacyjnego. W pracy staraliśmy się wykazać, że testowanie mutacyjne można optymalizować pod kątem samego procesu generacji mutantów oraz poprzez zrównoleglenie i rozproszenie procesu testowania. W podrozdziale 1.2 opisaliśmy pięć tez badawczych, które w kolejnych rozdziałach staraliśmy się uzasadnić.

Teza 1. Efektywność testowania mutacyjnego nie zmniejsza się znacząco, jeśli w nowej wersji oprogramowania mutanty generowane są wyłącznie dla kodu nowego bądź zmodyfikowanego.

Teza 2. Zmiana rozkładu prawdopodobieństwa losowania operatorów mutacyjnych, dokonana przy użyciu podejścia bayesowskiego, może być wykorzystana do istotnego zmniejszenia liczby mutantów przy dopuszczalnej niewielkiej utracie efektywności procesu.

Teza 3. Wzbogacenie podejścia Test-Driven Development o testowanie mutacyjne znacząco podnosi charakterystyki jakościowe kodu i zwiększa niezależność testowania poprzez obniżenie zjawiska stronniczości (ang. bias) autora testowanego kodu.

Teza 4. Podejście polegające na wprowadzeniu wielu mutacji do jednej kompilacji ma swoje ograniczenia i nie każdy operator mutacyjny może współistnieć z wszystkimi innymi.

Teza 5. Podejście polegające na wprowadzeniu wielu mutacji do jednej kompilacji dla języków, w których modyfikacja kodu pośredniego jest nie wykonalna może znacząco przyspieszyć proces testowanie mutacyjnego.

Poniżej przedstawiamy podsumowanie wyników naszych badań w kontekście weryfikacji tych tez.

W rozdziale 6 opisaliśmy tzw. Mutation Churn Model. Jego założeniem jest przeprowadzanie testów mutacyjnych tylko na zmienionym kodzie. Eksperymenty wykazały dużą skuteczność takiego działania. Liczba mutantów, których stan mógłby się zmienić z wykrytego na niewykryty przez testy w niezmienionym kodzie pod wpływem zmian w innych fragmentach oprogramowania

na podstawie badań okazała się nieznaczna. Oznacza to, że testowanie mutacyjne w procesie wytwarzania oprogramowania dla kolejnej wersji kodu można przeprowadzać tylko na małych wycinkach programu, które zostały w bieżącej wersji zmodyfikowane. Skuteczność tej optymalizacji sprawia, że testowanie mutacyjne może odbywać się praktycznie w sposób ciągły po każdej zmianie w kodzie zaraz po wykonaniu testów jednostkowych. Optymalizacja oparta o Mutation Churn Model może okazać się szczególnie skuteczna podczas stosowania metodyki TDD+M. Teza 1 została zatem uzasadniona.

Zaproponowany model optymalizacji generowania mutantów oparty na bayesowskim wnioskowaniu statystycznym okazał się skutecznym narzędziem. Badania wykazały, że przy odpowiednim dostosowaniu parametrów modelu do projektu można zmniejszyć liczbę generowanych mutantów o około 50%, przy jednoczesnej utracie niewielu tylko istotnych informacji w postaci niewygenerowania niewykrywanego przez testy mutanta. Dalsze prace nad tą optymalizacją z większym zespołem badawczym przy użyciu większej liczby projektów uczących mogą doprowadzić do wygenerowania jeszcze wydajniejszego modelu optymalizacyjnego. Podejście bayesowskie uzasadnia samoadaptacyjność systemu S.A.M. Teza 2 została zatem – przynajmniej w dużej części – wykazana.

Rola, jaką odgrywa testowanie mutacyjne w procesie wytwarzania oprogramowania szczególnie uwydatniła się podczas eksperymentu związanego z wprowadzaniem metodyki TDD+M. Wprowadzenie dodatkowej iteracji testów mutacyjnych do metodyki TDD poskutkowało znaczącym wzrostem jakości testów oraz wytwarzanego kodu. Koszty stosowania metodyki TDD+M okazały się na tyle nieznaczne, że wprowadzanie jej w początkowych fazach wytwórczych praktycznie nie wygenerowało jakichkolwiek dodatkowych kosztów związanych z czasem poświęconym przez programistę na wdrożenie w ten model i wykorzystanie go w praktyce. Teza 3 została zatem w pełni wykazana.

Ostatnią metodą optymalizacji odnoszącą się do generacji mutantów opisaną w niniejszej pracy była propozycja wprowadzenia modelu uwzględniającego mechanikę generowania wielu mutantów w jednej kompilacji. Teoretyczne podstawy tego modelu nadają się do implementacji i dalszych badań. Model może okazać się szczególnie przydatny podczas mutacji kodu opartego na języku, w którym nie ma możliwości modyfikowania skompilowanej wersji oprogramowania (np. nie ma możliwości bezpośredniej pracy na bajtkodzie). Na podstawie prototypu oprogramowania opartego na modelu generowania wielu mutacji w jednej kompilacji możemy stwierdzić, że proces ten podnosi wydajność procesu mutacyjnego testowania oprogramowania poprzez zmniejszenie czasu potrzebnego na kompilację poszczególnych wersji zmutowanego kodu. Wynik ten jest szczególnie istotny w kontekście zastosowania tej techniki dla języków nie mogących przeprowadzać testowania mutacyjnego bez każdorazowej kompilacji kodu dla każdego mutanta. Zatem tezy 4 oraz 5 zostały wykazane.

Opisany w niniejszej rozprawie proces testowania mutacyjnego (z wyłączeniem badań związanych z modelem TDD+M) był badany za pomocą systemu S.A.M., czyli skalowalnego, rozproszonego, samoadaptacyjnego systemu zarządzającego procesem testowania mutacyjnego. System okazał się wyjątkowo skuteczny podczas przeprowadzania badań nad optymalizacją opartą na bayesowskim

wnioskowaniu statystycznym. Dzięki temu, że praktycznie każdą funkcjonalność systemu można konfigurować i dopasowywać do określonych zadań, system S.A.M. stał się potężnym narzędziem badawczym mogącym testować różnorodne hipotezy badawcze dotyczące efektywności testowania mutacyjnego. W tym celu z reguły należy wprowadzić do systemu niewielki dodatkowy moduł, który pokieruje procesem mutacji w sposób, jaki jest wymagany do przeprowadzenia badań. Pisanie i dodawanie takiego modułu do systemu nie jest trudne, bowiem programista posiada dostęp do każdego etapu testowania mutacyjnego przeprowadzanego w systemie.

Na chwilę obecną proces testowania mutacyjnego jest dostarczany za pomocą zmodyfikowanej wersji programu PIT, która została sprowadzona do poziomu biblioteki dostarczającej mechanizmy mutacji. Uzależnienie mutacji kodu od programu PIT nie jest jednak konieczne – system S.A.M. może zostać dostosowany do dowolnego innego narzędzia mutującego.

Wszystkie tezy niniejszej pracy zostały potwierdzone, jednak ich tematyka nie została do końca wyczerpana. Praktycznie każdy obszar opisanych w niniejszej dysertacji prac może być podstawą do dalszych badań w danym zakresie. System S.A.M. także może być z powodzeniem dalej rozwijany. Na podstawie wiedzy zdobytej podczas jego wytwarzania i używania w celach badawczych można również stworzyć jego nowszą wersję, która będzie jeszcze bardziej wszechstronna.

Bibliografia

[1] M. Pančur and M. Ciglarič. Impact of test-driven development on productivity, code and tests: A controlled experiment. *Information and Software Technology*, 53:557–573, 2011.

[2] Piotr Wawrzyniak. *Rozprawa doktorska (w przygotowaniu).*

[3] Jiantao Pan. Software Testing. *Carnegie Mellon University*, 1999.

[4] ISTQB Certyfikowany tester - sylabus poziomu podstawowego (2011).

[5] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[6] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley and Sons, Inc., USA, 2004.

[7] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[8] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[9] Glenford J. Myers. *The art of software testing.* Wiley, 1979.

[10] L. Luo. Software testing techniques. technology maturation and research strategy. *Carnegie Mellon University, Class Report for 17-939A.*

[11] D. Gelperin and B. Hetzel. The growth of software testing. *CACM*, 31(6):687–695, 1988.

[12] Boris Beizer. *Software Testing Techniques.* Itp – Media, 1990.

[13] wired. https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.

[14] P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, Cambridge, UK, 2008.

[15] P. Jorgensen. *Software Testing. A Craftsman's Approach.* CRC Press, 2014.

[16] R.A. De Millo, R.J. Lipton, and F.G. Sayward. *Hints on test data selection: help for the practicing programmer*, chapter 4, pages 34–40. IEEE Computer 11, 1978.

[17] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11:207–225, 2001.

[18] Y.-S Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. *Proceedings of the 13th International Symposium on Software Reliability Engineering, IEEE*, pages 352–363, 2002.

[19] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, Aug 2018.

[20] Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[21] Orit Hazzan and Yael Dubinsky. *Agile Anywhere: Essays on Agile Projects and Beyond*, pages 45–46. Springer Publishing Company, Incorporated, 2014.

[22] Phra Pridsadi Tadeesom and Taratip Suwannasart. Mutation Operators in BPMN Model. In *Proceedings of the 2017 International Conference on Industrial Design Engineering*, ICIDE 2017, pages 45–48, New York, NY, USA, 2017. ACM.

[23] Matt Kirk. MoMuT. https://www.momut.org.

[24] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobst, and H. Brandl. MoMut::UML Model-Based Mutation Testing for UML. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, April 2015.

[25] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korošec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, pages 1–19, Cham, 2014. Springer International Publishing.

[26] Donald W. McCormick II. Towards a sufficient set of mutation operators for structured query language (sql). *Virginia Polytechnic Institute and State University Masters Theses [18201]*, 18201, 2010.

[27] Jose Suárez-Cabal and Javier Tuya. Structural Coverage Criteria for Testing SQL Queries. *Journal of Universal Computer Science*, 15(3), 2009.

[28] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, April 2010.

[29] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*, pages 300–309, Nov 2010.

[30] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014.

[31] Macario Polo, Mario Piattini, and Ignacio García Rodríguez de Guzmán. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19:111–131, 2009.

[32] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25:605–628, 2015.

[33] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. *SSTA 2013 – Proceedings of the International Symposium on Software Testing and Analysis*, pages 235–245, 2013.

[34] M.P. Usaola and P.R. Mateo. Mutation testing cost reduction techniques. *A Survey, IEEE Software*, 27:80–86, 2010.

[35] Eric Wong and Aditya Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[36] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996.

[37] R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, and K.N. King. An extended overview of the Mothra software testing environment. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, July 1988.

[38] Ilona Bluemke and Karol Kulesza. Reduction of computational cost in mutation testing by sampling mutants. In Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk, editors, *New Results in Dependability and Computer Systems*, pages 41–51, Heidelberg, 2013. Springer International Publishing.

[39] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, November 9–11 2011.

[40] René Just. The Major mutation framework. http://mutation-testing.org.

[41] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, November 28–30 2012.

[42] Gary Kaminski, Paul Ammann, and Jeff Offutt. Better Predicate Testing. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 57–63, New York, NY, USA, 2011. ACM.

[43] René Just and Franz Schweiggert. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, 25(5-7):490–507, 2014.

[44] Henry Coles. PIT Mutation Testing. http://pitest.org.

[45] ASM. https://asm.ow2.io/.

[46] Mujava. https://cs.gmu.edu/ ∼ offutt/mujava/.

[47] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava : An Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15:97–133, 2005.

[48] BCEL. https://commons.apache.org/proper/commons-bcel/.

[49] Francisco Palomo-Lozano, Antonia Estero-Botaro, Inmaculada Medina-Bulo, and Manuel Núñez. Test Suite Minimization for Mutation Testing of WS-BPEL Compositions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 1427–1434, New York, NY, USA, 2018. ACM.

[50] Rajvir Singh and Mamta Santosh. Test case minimization techniques: A review. 2:1048–1056, 12 2013.

[51] Marcin Data. Efektywność testowania mutacyjnego. *Uniwersytet Jagielloński, Instytut Informatyki i Matematyki Komputerowej, praca magisterska*, 2016.

[52] Quang Vu Nguyen and Lech Madeyski. On the relationship between the order of mutation testing and the properties of generated higher order mutants. In Ngoc Thanh Nguyen, Bogdan Trawiński, Hamido Fujita, and Tzung-Pei Hong, editors, *Intelligent Information and Database Systems*, pages 245–254, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[53] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. *SIGSOFT Softw. Eng. Notes*, 18(3), 1993.

[54] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 50–56, May 23–24 2011.

[55] Markus Kusano and Chao Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 722–725, Piscataway, NJ, USA, 2013. IEEE Press.

[56] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2, 1976.

[57] Michał Mnich, Adam Roman, and Piotr Wawrzyniak. Mutation churn model. *Schedae Informaticae*, 2016(25):227–236, 2017.

[58] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *M.I.T. Sloan School of Management*, E53-315, December 1993.

[59] Victor Laing and Charles Coleman. Principal components of orthogonal object-oriented metrics. *White Paper Analyzing Results of NASA Object-Oriented Data*, 323-08-14, October 2001.

[60] James O. Berger. *Statistical Decision Theory and Bayesian Analysis*, chapter 1, pages 1–45. Springer, New York, NY, 1985.

[61] Claude A Pruneau. *Data analysis techniques for physical scientists.* Cambridge University Press, Cambridge, Oct 2017.

[62] Roman Nowak. *Statystyka dla fizyków*, chapter 9, pages 534–543. PWN, 2002.

[63] Adam Roman and Michał Mnich. Test-driven development with mutation testing – an experimental study. *Software Quality Journal*, 2020.

[64] Anna Derezinska and Piotr Trzpil. Mutation testing process combined with test-driven development in .net environment. In *Theory and Engineering of Complex Systems and Dependability*, pages 131–140, Cham, 2015. Springer International Publishing.

[65] Matt Kirk. PIT Mutation Testing TDD. http://pitest.org/sky_experience.

[66] I. Ahmed, C. Jensen, A. Groce, and McKenney P.E. Applying mutation analysis on kernel test suites: An experience report. *IEEE Int. Conf. on Software Testing Verification and Validation Workshop, ICSTW*, pages 110–115, 2017.

[67] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: a practical mutation testing tool for Java. *ACM International Symposium on Software Testing and Analysis, ISSTA*, pages 449–452, 2016.

[68] A. Groce, I. Ahmed, C. Jensen, and P.E. McKenney. How verified is my code? falsification-driven verification. *IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 737–748, 2015.

[69] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature).* Addison-Wesley Longman, Amsterdam, 2002.

[70] David S. Janzen. Software Architecture Improvement Through Test-driven Development. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 222–223, New York, NY, USA, 2005. ACM.

[71] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 356–363, New York, NY, USA, 2006. ACM.

[72] L. Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2010.

[73] B. George and L. Williams. A structured experiment on test-driven development. *Information and Software Technology*, 46:337–342, 2004.

[74] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering*, pages 356–363, New York, NY, USA, 2006. ACM.

[75] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

[76] T. Flohr and T. Schneider. Lessons learned from an XP experiment with students: test-first needs more teachings. In M. Vierimaa J. Münch, editor, *Lecture Notes in Computer Science*, volume 4034, pages 305–318, 2006.

[77] A. Gupta and P. Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *ESEM'07: International Symposium on Empirical Software Engineering and Measurement*, pages 285–294. IEEE Computer Society, 2007.

[78] L. Huang and M. Holcombe. Empirical investigation towards the effectiveness of test first programming. *Information and Software Technology*, 51:182–194, 2009.

[79] D.S. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, 2008.

[80] D. Janzen and H. Saiedian. On the influence of test-driven development on software design. In *CSEET, 19th Conference on Software Engineering Education and Training (CSEET'06)*, pages 141–148, 2006.

[81] L. Crispin. Driving software quality: how test-driven development impacts software quality. *IEEE Software*, 23(6):70–71, 2006.

[82] A. Geras, M.R. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *IEEE METRICS'2004: Proceedings of the 10th IEEE International Software Metrics Symposium*, pages 405–416. IEEE Computer Society, 2004.

[83] M. Pančur, M. Ciglarič, M. Trampuš, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *EUROCON, Proceedings of the International Conference on Computer as a Tool*, pages 83–86, 2003.

[84] M. Siniaalto and P. Abrahamsson. A comparative case study on the impact of test driven development on program design and test coverage. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 275–284. IEEE Computer Society, 2007.

[85] M.M. Mueller and O. Hagner. Experiment about test-first programming. *IEE Proceedings – Software*, 149(5):131–136, 2002.

[86] L. Madeyski. Preliminary analysis of the effects of pair programming and test driven development on the external code quality. In *Software Engineering: Evolution and Emerging Technologies*, volume 130, pages 113–123. IOS Press, 2005.

[87] M. Aniche and M.A. Gerosa. Does test-driven development improve class design? a qualitative study on developers' perceptions. *Journal of the Brazilian Computer Society*, 21:15:1–11, 2015.

[88] A. Caueviv, S. Punnekkat, and D. Sundmark. Tddhq: Achieving higher quality testing in test driven development. *IEEE 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 33–36, 2013.

[89] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Routledge, 1988.

[90] S. Savilowsky. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods*, 8:467–474, 2009.

[91] Testowanie mutacyjne w praktycznych procesach deweloperskich, http://2017.testwarez.pl/testowanie-mutacyjne-w-praktycznych-procesach-deweloperskich/.

[92] V.L. Willson and R.R. Putnam. A meta-analysis of pretest sensitization effects in experimental design. *American Educational Research Journal*, 19(2):249–258, 1982.

[93] Jumble. http://jumble.sourceforge.net/.

[94] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń.* Wydawnictwo Naukowe PWN, 2012.

[95] IEEE Standard for Software Test Documentation. *IEEE Std 829-1998*, pages 1–64, 1998.

[96] R. Likert. *A Technique for the Measurement of Attitudes.* Archives of Psychology, 1932.

[97] J.C.F. deWinter. Using the student's t-test with extremely small sample size. *Practical Assesment, Research and Evaluation*, 18(10):1–12, 2013.

Dodatek A

Legenda do tabel 7.4.1 oraz 7.5

Poniższa tabela pokazuje nazwy klas wykorzystywanych w eksperymentach w rozdziale 6, gdzie — ze względu na ilość miejsca — kodowane są symbolami c01, c02 itd.

Tabela A.1: Legenda do tabel 7.4.1 oraz 7.5

c01	org.JUnit.Assert
c02	org.JUnit.runners.BlockJUnit4classRunner
c03	org.JUnit.runner.JUnitCommandLineParseResult
c04	org.JUnit.runner.Description
c05	org.JUnit.rules.TemporaryFolder
c06	org.JUnit.internal.ComparisonCriteria
c07	org.JUnit.runners.Parameterized
c08	org.JUnit.runner.JUnitCommandLineParseResult
c09	org.JUnit.runner.Description
c10	org.JUnit.Assert
c11	org.JUnit.runner.JUnitCommandLineParseResult
c12	org.JUnit.internal.TextListener
c13	org.JUnit.Assume
c14	org.JUnit.Assert
c15	org.JUnit.validator.AnnotationValidatorFactory
c16	org.JUnit.runners.BlockJUnit4classRunner
c17	org.JUnit.runners.Parameterized
c18	org.JUnit.runners.ParentRunner
c19	org.JUnit.runner.JUnitCommandLineParseResult
c20	org.JUnit.runner.Description
c21	org.JUnit.rules.ExpectedExceptionMatcherBuilder
c22	org.JUnit.rules.Stopwatch
c23	org.JUnit.rules.TemporaryFolder
c24	org.JUnit.internal.ComparisonCriteria
c25	org.JUnit.internal.ArrayComparisonFailure

c26	org.JUnit.internal.TextListener
c27	org.JUnit.ComparisonFailure
c28	org.JUnit.Assert
c29	org.JUnit.validator.AnnotationValidatorFactory
c30	org.JUnit.runners.BlockJUnit4classRunner
c31	org.JUnit.runners.Parameterized
c32	org.JUnit.runners.ParentRunner
c33	org.JUnit.runners.Suite
c34	org.JUnit.runner.FilterFactories
c35	org.JUnit.runner.JUnitCommandLineParseResult
c36	org.JUnit.runner.JUnitCore
c37	org.JUnit.runner.Description
c38	org.JUnit.runner.Request
c39	org.JUnit.runner.Result
c40	org.JUnit.rules.DisableOnDebug
c41	org.JUnit.rules.ErrorCollector
c42	org.JUnit.rules.ExpectedException
c43	org.JUnit.rules.ExternalResource
c44	org.JUnit.rules.RuleChain
c45	org.JUnit.rules.RunRules
c46	org.JUnit.rules.TemporaryFolder
c47	org.JUnit.rules.TestWatcher
c48	org.JUnit.rules.Timeout
c49	org.JUnit.internal.ComparisonCriteria
c50	org.JUnit.internal.AssumptionViolatedException
c51	org.JUnit.internal.MethodSorter
c52	org.JUnit.internal.TextListener

Dodatek B

Podręcznik systemu generowania wielu mutantów w jednej kompilacji oraz dane dostępowe do repozytorium

B.1 Podręcznik

Niniejszy podręcznik systemu do testowania mutacyjnego oprogramowania korzystającego z technologii generowania wielu mutantów w jednej kompilacji składa się z dwóch sekcji: opisu komend oraz przykładowych uruchomień.

B.1.1 Komendy

- Program uruchamia się za pomocą komendy `MMIOC.exe nazwaPlikuCpp`. Przekazywany w parametrze plik „cpp" będzie plikiem, na którym odbędzie się proces mutacji.
- `help` – wyświetla listę dostępnych komend.
- `mut` – generuje zmutowane pliki „cpp". Z parametrem `-a` wygenerowany zostanie jeden plik zawierający zagregowane wszystkie mutacje w kodzie. Z parametrem `-s` zostanie wygenerowany zbiór plików, przy czym każdy z nich będzie zawierał pojedynczą mutację.
- `comp` – generuje skompilowany plik „exe". Z parametrem `-a` kompilowany jest plik zawierający zagregowane mutanty. Z parametrem `-s` kompilowane są wszystkie pliki zawierające pojedynczą mutację.
- `run` – uruchamia skompilowany plik „exe". Z parametrem `-a` uruchamiany jest w pętli plik zawierający zagregowane mutanty. Każde przejście pętli uruchamia plik z kolejnym numerem parametru sterującego od 1 do liczby wszystkich mutantów. Z parametrem `-s` uruchamiane są wszystkie pliki zawierające pojedynczą mutację.

- `stat` zwraca statystyki w formie loga oraz CSV.

B.1.2 Przykładowe uruchomienia

Pniżej podana jest pełna sekwencja komend dla pliku `synt01.cpp`. Plik znajduje się w repozytorium pod adresem:

https://github.com/michaelmnich/-MMIOC/tree/master/MMIOC/CodeSamples/Synthetic

- Uruchomienie programu: .\MMIOC.exe synt01. Uwaga w katalogu .\code\musi znajdować się plik synt01.cpp.
- `mut -a` – wygenerowany zostanie w katalogu .\code\mutants\oneComp\zmutowany plik synt01.cpp.
- `mut -s` – wygenerowany zostanie w katalogu .\code\mutants\manyComp\zestaw katalogów, gdzie każdy z nich będzie zawierał jeden zmutowany plik synt01.cpp. W tym wypadku mamy jeden plik dla jednej mutacji.
- `comp -a` – wygenerowany zostanie w katalogu .\code\mutants\oneComp\zmutowany plik synt01.exe.
- `comp -s` – wygenerowany zostanie w katalogu .\code\mutants\manyComp\KatalogMutacji plik synt01.exe.
- `run -a` – w pętli zostanie uruchomiony plik synt01.exe x, gdzie x to numer od 1 do liczby wszystkich mutantów.
- `run -s` – w każdym katalogu zawierającym zmutowany kod z jedną mutacją zostanie uruchomiony plik synt01.exe.

B.2 repozytorium

Repozytorium projektu znajduje się pod adresem: https://github.com/michaelmnich/-MMIOC

Dodatek C

S.A.M. — dokumentacja użytkowa oraz dane dostępowe do repozytorium

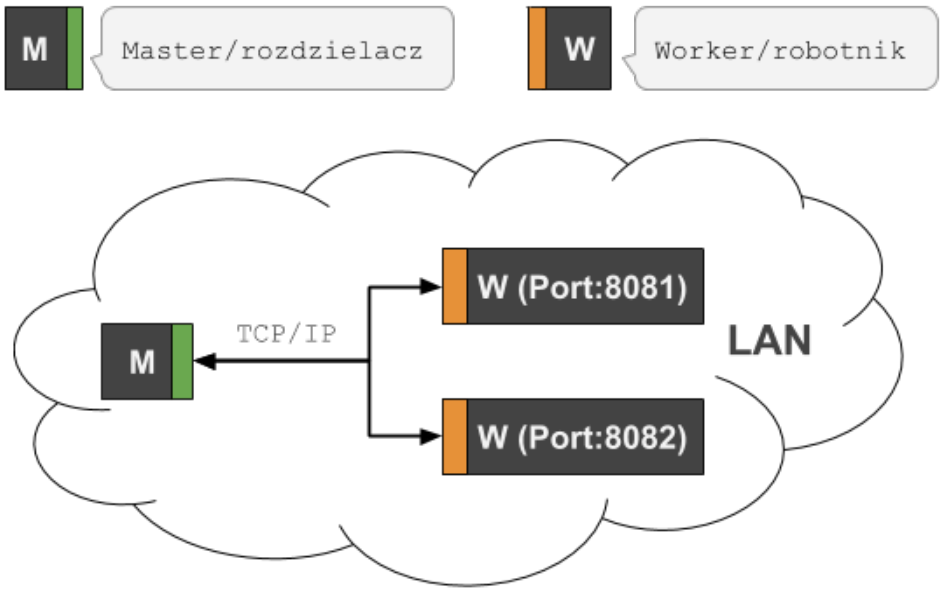
S.A.M. jest system zarządzanym poprzez linię komend bądź zewnętrzne narzędzia. Ten dodatek opisuje instrukcje obsługi programu. Program uruchamiany jest ze środowiska programistycznego lub z wiersza poleceń. Główną klasą projektu jest `Java MainWorker`.

Po uruchomieniu system wyświetla konsolę z komunikatem powitalnym. Po wpisaniu komendy `help` konsola wypisze dostępne komendy, czyli:

- `test` – wewnętrzny test systemu.
- `connect` – system pyta o adres IP i port, a następnie po podaniu prawidłowych danych połączy z nim. Po połączeniu można wysłać żądanie rozpoczęcia testów mutacyjnych.
- `start` – uruchamia serwer mutacyjny. Należy podać port, na którym serwer będzie działał.
- `run mutation -i` – wysyła wszystkim połączonym maszynom komunikat o rozpoczęcia mutacji. Po parametrze `-i` należy podać zestaw klas i testów, które mają być przekazane do danego węzła.
- `run mutation` – uruchamia proces mutacji zgodnie ze zdefiniowanym plikiem konfiguracyjnym.
- `run mutation - pc` – uruchamia proces mutacji zgodnie ze zdefiniowanym plikiem konfiguracyjnym (mutacja odbywa się per plik).
- `run mutation - bayes` – uruchamia proces mutacji zgodnie ze zdefiniowanym plikiem konfiguracyjnym, przy użyciu optymalizacji opartej o model bayesowski.
- `run mutation - bayes -pc` – uruchamia proces mutacji zgodnie ze zdefiniowanym plikiem konfiguracyjnym, przy użyciu optymalizacji opartej na modelu bayesowskim (mutacja odbywa się per plik).

C.1 Przykładowa konfiguracja

Konfigurując system, posłużymy się topologią przedstawioną na rys. C.1.



Rysunek C.1: Przykładowa topologia systemu S.A.M.

W celu stworzenia klastra obliczeniowego opartego o topologię opisaną diagramem z rys. C.1 należy wykonać szereg następujących kroków.

- Krok 1. Postawienie serwera nasłuchującego SLAVE 1. Uruchamiamy instancję S.A.M., następnie wpisujemy komendę `start` i wybieramy port. Serwer zostanie uruchomiony.

  Przykład:
  ```
  start
  Set server working port Port:
  8081
  Serwer waiting for request on port:8081
  ```

- Krok 2. Postawienie serwera nasłuchującego SLAVE 2. Powtarzamy procedurę z kroku 1.

  przykład:
  ```
  start
  Set server working port Port:
  8081
  Serwer waiting for request on port:8082
  ```

- Krok 3. Postawienie węzła typu Master. Uruchamiamy trzecią instancję S.A.M. i w konsoli wpisujemy:

```
connect
Server address:
localhost
Server Port:
8081
FROM SERVER: You are connected to SAM-SYSTEM Node
```

a następnie:

```
connect
Server address:
localhost
Server Port:
8082
FROM SERVER: You are connected to SAM-SYSTEM Node
```

- Krok 4. W węźle Master uruchamiamy komendę `run mutation`, co przy odpowiednio skonfigurowanym pliku mutacyjnym uruchomi proces mutacji w każdym z węzłów typu robotnik (SLAVE). Dane do rozesłania mogą być zdefiniowane w pliku konfiguracyjnym lub przesłane za pomocą linii komend. W tym celu należy uruchomić komendę z dodatkowym parametrem `-i`, a następnie po nim przesłać zdefiniowany zestaw klas i testów.

  Przykład:

```
run mutation -i -classPath
D:\\Doktorat\\PitPlayground\\IOgr602-master\\
target\\test-classes\\
,D:\\Doktorat\\PitPlayground\\IOgr602-master\\target\\classes\\
-reportDir D:\\trash\\
-targetclasses
matrixlibrary.*
-targetTests
matrixlibrary.*
-sourceDirs
D:\\Doktorat\\PitPlayground\\IOgr602-master\\
```

C.2 Główny plik konfiguracyjny

Kolejną istotną sprawą jest stworzenie głównego pliku konfiguracyjnego `mainconfig.cfg`. Powinien on znajdować się w głównym katalogu projektu.

Przykład:

```
<HereConfigStarts>
PIT-JAR-DIR;D:\\Doktorat\\pitcmd\\PIT-1.1.11-SNAPSHOT.jar
PIT-CMD-JAR-DIR;D:\\Doktorat\\pitcmd\\
PIT-command-line-1.1.11-SNAPSHOT.jar
HAMCREST-DIR;C:\\JUnit\\hamcrest-core-1.3.jar
JUnit-DIR;C:\\JUnit\\JUnit-4.12.jar
PROJECT-class-DIR;D:\\Doktorat\\PitPlayground\\IOgr602-master
\\target\\test-classes\\
PROJECT-TEST-class-DIR;D:\\Doktorat\\PitPlayground\\IOgr602-master
\\target\\classes\\
```

Szczegóły związane ze sposobem użycia głównego pliku konfiguracyjnego znajdują się w kodzie oprogramowania w klasach `ProjectConfig` oraz `PitRunner`.

C.3 Konfiguracja mutacji

Konfiguracja procesu mutacji znajduje się w pliku o nazwie `config.ini`. Format pliku konfiguracyjnego odpowiada formatowi CSV. Plik wypełnia się zgodnie z poniższym schematem.

```
  Operatory mutacyjne:------------------------
NAZWA_OPERATORA;Prawdopodobieństwo;SkalaPrawdopodobieństwa
```

W powyższym schemacie został opisany sposób, w jaki należy ustawiać początkowy stan pliku definiującego prawdopodobieństwo losowania mutanta.

Przykład:

```
INVERT-NEGS;0.5;100
Zmienne środowiskowe-----------------------
MUTATION-MODE;1 lub 0.
1 oznacza że losowanie mutantów włączone
0 oznacza że wyłączone i bierze wszystkie operatory
<HereConfigStarts> //wszytko powyżej jest uznawane za komentarz
INVERT-NEGS;0.5;100
//od tego miejsce zmienne środowiskowe dla pita
MUTATION-MODE;1 //Definicja trybu mutacji
```

C.4 repozytorium

Repozytoria systemu S.A.M. opisanego w dziale 9 znajdują się w serwisie Github.

- S.A.M. Mutation-Core: https://github.com/michaelmnich/S.A.M-Mutation-Core.git
- S.A.M. opti-module: https://github.com/wawrzyniak/SAM-optimalization-module.git