Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

TESTOWANIE MUTACYJNE WYKORZYSTUJĄCE ASPEKTY

Jacek Pospychała

Promotor
dr inż. Bartosz Walter

Poznań, 2008 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wstęp

Testowanie jednostkowe, które swoją prostotą i użytecznością zrewolucjonizowało proces wytwarzania oprogramowania, jest obecnie jednym z podstawowych mechanizmów zapewnienia jakości pisanych programów. Poprzez proste testy małych wycinków systemu daje przybliżoną odpowiedź czy tworzony program jest zgodny z oczekiwaniami [5]. Jednak trafność tej odpowiedzi zależy od jakości testów.

Powszechnie stosowane narzędzia dokonują oceny wartości miary pokrycia kodu programu testami, a więc dostarczają jedynie informacji jak duża część systemu jest analizowana przez testy [14]. Nie ma jednak prostego sposobu oceny jakości testów jednostkowych, czyli odpowiedzi na pytanie, z jaką skutecznością wykrywają one błędy w systemie.

Rozwiązaniem tego problemu byłoby narzędzie, równie proste w obsłudze jak narzędzia do testowania jednostkowego, które w sposób automatyczny, w oparciu o jednoznaczne reguły pozwalałoby dokonać oceny jakości przygotowanych testów i wskazać wykryte wady.

Badania w dziedzinie testowania mutacyjnego dostarczają zbioru najprostszych błędów popełnianych w programach oraz sposobów ich symulowania, w postaci tzw. operatorów mutacji [1], [12], [11], [10]. Jednak tradycyjny proces wykorzystania operatorów mutacji, tj. testowanie nie tylko oryginalnego, ale i wielu dodatkowych nieznacznie zmodyfikowanych systemów (mutantów) celem jest niezwykle czasochłonny w dużych systemach z dużymi zbiorami testów jednostkowych.

B. Walter i B. Bogacki w [2] zaproponowali rezygnację z wielokrotnego uruchamiania mutantów na rzecz modyfikacji działania testów w trakcie ich uruchomienia, korzystając z programowania aspektowego. Celem niniejszej pracy jest zaimplementowanie takiego środowiska, aby nie wpływając istotnie na sposób pracy testera, zaraz po wykonaniu testów, dostarczało ono rozszerzoną informację zwrotną o jakości napisanych testów, oraz wskazywało potencjalne błędy.

Opracowane środowisko przeznaczone jest do wykorzystania z testami jednostkowymi napisanymi w języku Java, przy wykorzystaniu biblioteki JUnit. Rozszerza ono platformę Eclipse i jej interfejs użytkownika służący do testowania aplikacji w ramach modułu Java Development Tools (JDT).

Struktura pracy jest następująca. W rozdziale 2 przedstawiono przegląd literatury na temat testowania mutacyjnego, oraz programowania aspektowego. Rozdział 3 jest poświęcony środowisku testowania przygotowanemu w ramach niniejszej pracy. Prezentuje typowy scenariusz pracy testera oraz wnoszone udogodnienia. Zawiera także krótki opis możliwości dalszych badań. Rozdział 4 dokumentuje zastosowane operatory mutacyjne i ich znaczenie w kontekście jakości testów. Rozdział 5 zawiera analizę wydajności opracowanego środowiska na przykładzie kolekcji popularnych bibliotek z projektu Apache Commons. Rozdział 6 stanowi podsumowanie pracy.

Rozdział 2

Podstawy teoretyczne

2.1 Testowanie mutacyjne

2.1.1 Koncepcja testowania mutacyjnego

Testowanie jest nieodłącznym elementem procesu wytwarzania oprogramowania. Przetestowanie programu wymaga określenia kryteriów poprawności programu, przygotowania testu, wykonania go i oceny w jakim stopniu kryteria zostały spełnione. Przypadek testowy to scenariusz wykonania prostych operacji na systemie, pozwalający zweryfikować poprawność tych operacji. Najczęściej weryfikacja poprawności odbywa się przez porównanie wyników operacji zwróconych przez testowany system do wyników z góry określonych przez testera na podstawie jego wiedzy oraz specyfikacji lub odgórnie określonych wymagań. Test jest uznawany za poprawny, jeśli każda z operacji wykonywanych w teście zwróci taką odpowiedź, jaka została przewidziana przez testera. W przeciwnym razie, tj. gdy przynajmniej jedna z odpowiedzi będzie inna, wynik testu jest błędny. Innymi słowy, przypadki testowe służą porównaniu odpowiedzi systemu z oczekiwanymi, określonymi przez testera na podstawie dokumentacji, jego wiedzy, doświadczenia, lub innych czynników.

O jakości przypadków testowych świadczą ich poprawność i kompletność. Na poprawnie skonstruowany przypadek testowy składa się wiele elementów: poprawnie wyspecyfikowane odwołanie do systemu i weryfikacja poprawności otrzymanej odpowiedzi. By zbiór testów był kompletny, przypadki testowe powinny sprawdzać wszystkie oczekiwane (udokumentowane, lub opracowane przez testera) odpowiedzi pod kątem ich zgodności z odpowiedziami systemu.

Podstawowym narzędziem oceny kompletności testów jest analiza pokrycia kodu systemu przypadkami testowymi. Analiza pokrycia kodu opiera się na wyznaczeniu stopnia wykorzystania systemu przez przypadki testowe. Najpopularniejsze wskaźniki stopnia wykorzystania systemu przez przypadki testowe opierają się na porównaniu liczby linii kodu systemu odwiedzonych w trakcie wykonywania testów do wszystkich linii. Analogicznie istnieją także metryki na poziomie liczby bloków, metod, lub możliwych ścieżek wykonania kodu programu. Na podstawie oceny kompletności testów nie można jednak stawiać żadnych wniosków co do ich poprawności, a więc i jakości.

Testowanie mutacyjne to jedna z metod analizy poprawności testów jednostkowych. Pierwotnie udokumentowana w 1971 roku przez Richarda Liptona w “Fault Diagnosis of Computer Programs”. Pod koniec lat siedemdziesiątych opublikowane zostały główne prace na jej temat, autorstwa DeMillo, Liptona i Saywarda [3] i Hamleta [7].

W pracach tych sformułowano podstawową tezę dowodzącą o przydatności testowania mutacyjnego - nazwaną „efektem wiązania”:

> Dane testowe, które odróżniają wszystkie programy różniące się od poprawnego tylko o drobne błędy, są tak czułe, że pośrednio odróżniają także złożone błędy.

Innymi słowy, podstawowe błędy w oprogramowaniu, tj. zła implementacja, lub niespełnienie wymagań, czyli czynniki decydujące o tym czy program spełnia swoją rolę czy nie, są powiązane z drobnymi błędami rozsianymi w kodzie programu. Te drobne błędy często w praktyce sprowadzają się do:

1. braku obsługi określonych stanów
2. wybór złych ścieżek wykonania programu
3. wykonywanie złych akcji

Użyte zwroty „drobne" i „złożone" błędy zaczerpnięto z klasyfikacji przedstawionej w [6].

Celem testowania mutacyjnego jest poszerzenie wiedzy na temat testów. Słabe mutacje (ang. weak) sprawdzają do jakiego stopnia test symuluje zachowanie systemu. Silne mutacje (ang. strong) mierzą także jak dokładnie testy sprawdzają wyniki programu.

Dla silnych mutacji, mutanta uważa się za zabitego, gdy wynik testu jest inny od oczekiwanego. W przypadku słabych mutacji badany jest tylko stan programu bezpośrednio po zmutowanym wyrażeniu. Różnicę między silnymi a słabymi mutacjami można dostrzec gdy niepoprawny stan bezpośrednio po zmutowanym wyrażeniu nie będzie miał wpływu na zmianę wyniku testu. Tak więc silne mutacje mówią więcej o testach niż słabe, jednak słabe są szybsze do wykonania. Słabe mutacje mierzą pokrycie kodu programu. Silne mutacje mówią czy testy sprawdzają całość wyników programu, czy tylko fragmenty.

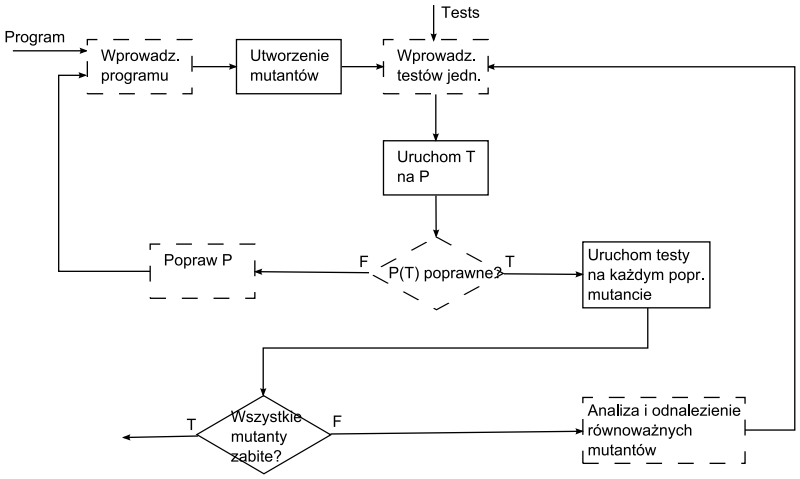
Mutacja to dowolna pojedyncza zmiana kodu, np. zmiana operatora w działaniu matematycznym, czy zmiana wartości stałej, będąca wynikiem przekształcenia testowanego systemu przy pomocy operatora mutacji. System wraz z mutacją jest testowany przy użyciu testów jednostkowych celem wykrycia wprowadzonej zmiany. Mutacja jest uznawana za zabitą, jeśli dowolny test ze zbioru testów jednostkowych zakończy się niepowodzeniem dla systemu z wprowadzoną mutacją. W przeciwnym razie mutacja jest uznawana za żywą. Ocena jakości testu jest określana na podstawie analizy żywych i martwych mutacji. Jest to iloraz liczby martwych mutantów do liczby wszystkich mutantów. Zatem celem testera jest doprowadzenie do oceny jakości równej 1, gdy wszystkie mutanty zostaną zabite.

Nadzieja pokładana w testowaniu mutacyjnym została trafnie sformułowana przez Geista [13]: „W praktyce, jeśli oprogramowanie zawiera błędy, na ogół będzie istniał zbiór mutantów, które mogą zostać zabite tylko przez przypadki testowe, który także wykrywa te błędy".

W tradycyjnym modelu testowania mutacyjnego (rys. 2.1), po napisaniu kodu programu P, automatycznie tworzone są mutacje programu przy pomocy operatorów mutacji. Po przygotowaniu testów jednostkowych T, każda program (P) jest testowany przy pomocy zestawu testów T. Jeżeli $P(T)$ zakończy się niepowodzeniem, należy poprawić P i ponowić testy. Jeżeli $P(T)$ zakończy się powodzeniem, dla każdej wygenerowanej mutacji programu należy uruchomić testy T. Jeżeli wszystkie mutacje są martwe, testowanie jest zakończone. Jeżeli pozostały nie zabite mutacje, należy przeanalizować wyniki.

2.1.2 Wstrzeliwanie odpowiedzi

B. Bogacki i B. Walter biorąc pod uwagę dużą złożoność tradycyjnego testowania mutacyjnego zaproponowali odmienną koncepcję testowania mutacji, wykorzystującą programowanie aspektowe



Rysunek 2.1: Tradycyjny model testowania mutacyjnego.
Linią przerywaną zaznaczone są etapy wykonywane ręcznie, a ciągłą wykonywane automatycznie.

[2]. W zobrazowaniu specyfiki tej koncepcji pomoże przykład zaczerpnięty z ich artykułu. Weźmy pod uwagę kod programu 1, oraz odpowiadający mu test 2.

Test JUnit w przykładzie 2 może się zakończyć niepowodzeniem w jednej z trzech sytuacji: (1) Metoda *Foo.bar*() wywołana z parametrem 3 zwróci wartość różną od 3000; (2) Zostanie zgłoszony nieoczekiwany w wyjątek; (3) wartości parametrów 0 i 6 nie spowodują rzucenia spodziewanego wyjątku *IllegalArgumentException*. Mutacje nie mające wpływu na wartości zwracane przez metodę *Foo.bar*() nie są możliwe do wykrycia.

Dlatego zamiast dokonywać dowolnych zmian, nie wiedząc czy przekładają się one na wyniki zwracane przez system, czy nie, wystarczy podmieniać same wyniki. Proponowane rozwiązanie, wykorzystujące programowanie aspektowe zakłada że odpowiednio skonstruowany aspekt przechwyci wywołanie metody *Foo.bar*() i jej wyniki, a w kolejnych wywołaniach testu podstawione zostaną inne od spodziewanych wartości, tak jakby miała miejsce mutacja kodu źródłowego.

```
public class Foo {
  public int bar(int a) throws IllegalArgumentException {
    if ((a > 5) || (a < 1)) {
      throw new IllegalArgumentException();
    }
    int c = a;
    for (int i = 0; i < a; i++) {
      c *= 10;
    }
    return c;
  }
}
```

Listing 1: Kod przykładowego programu.

2.1.3 Operatory mutacji

Istnieje szereg klasyfikacji operatorów mutacji. W szczególności ze względu na:

1. Paradygmat programowania.

```
public void testBar () {
  assertEquals (3000, new Foo().bar(3));
  try {
    new Foo().bar(6);
    fail ("Expected exception for value: 6");
  } catch (IllegalArgumentException e) {}
  try {
    new Foo ().bar(0);
    fail ("Expected exception for value: 0");
  } catch (IllegalArgumentException e) {}
}
```

Listing 2: Przykładowy test JUnit dla metody *Foo.bar*().

a) Strukturalny, a więc wg. [4] w przypadku gdy program jest sekwencją instrukcji, posiadających podobną strukturę jak sam program, punkt początkowy i końcowy, oraz wykonywany jest sekwencyjnie, przez wybór następnej, lub wielokrotną iterację sekwencji instrukcji. Mutacji podlegać mogą wywołania instrukcji, operatory jednoargumentowe, działań, porównań, przypisań, operacje na zmiennych i stałych.

b) Obiektowy. Programowanie obiektowe stało się odpowiedzią na rosnącą złożoność programów strukturalnych i wniosło wiele nowych zasad i ograniczeń, jakie program musi spełniać, aby mógł zostać skompilowany i uruchomiony. Testowaniu mutacyjnemu podlegają zatem wszelkie podstawowe koncepcje programowania obiektowego, tj. klasy, obiekty, metody, dziedziczenie, hermetyzacja, polimorfizm, abstrakcja. Na tej podstawie, testowaniu mutacyjnemu podlegają także ogólnie przyjęte wzorce programowania obiektowego.

2. Język programowania. Operatory mutacji uwzględniają cechy specyficzne języka, zarówno syntaktyczne jak i te związane ze stylem programowania. Jeżeli natomiast idea operatora jest niezależna od języka programowania, często sposób implementacji wymaga uwzględnienia cech specyficznych języka.

   a) C. Przegląd operatorów mutacji dla języka C zawarto w [1]. Autorzy pracy proponują dalszy podział operatorów wg. rodzaju mutowanych konstrukcji języka C na: mutacje wyrażeń, operatorów, zmiennych i stałych. Operatory w tych kategoriach zostały zaprojektowane by uwidaczniać błędy programistów w: wyborze identyfikatorów i stałych przy konstruowaniu wyrażeń, formułowaniu odwołań do funkcji, oraz formułowaniu iteracji i wyrażeń warunkowych.

   b) Fortran. Ze względu na prostotę języka Fortran, [1] proponuje jedynie 22 operatory mutacji, w porównaniu z 77 dla C. Mniejsza liczba mutacji jest wynikiem mniejszej liczby typów prostych, braku rekurencji, oraz braku części operatorów, takich jak *przecinek*, czy operatory wykorzystywane w pętlach - *break* i *continue*, występujące m.in. w C.

   c) Ada. Przegląd operatorów mutacji dla wszystkich elementów składni języka ADA dokumentuje [10]. Cechy specyficzne języka ADA mające specjalne znaczenie dla testów, a więc i operatorów mutacji to silne typowanie, przeciążanie, typowane wskaźniki, obsługa wyjątków, pakiety generyczne, zadania (tasks).

   d) C++. Bogata składnia i duża złożoność języka C++ są obiektem analizy operatorów specyficznych dla tego języka. Przykładowe operatory mutacji mogę analizować przeciążanie operatora [] - cechę C++ unikatową w skali innych języków [12].

e) Java. Obszar eksploatowany przez operatory mutacji języka Java, to elementy biblioteki standardowej języka, m.in. biblioteka kolekcji, oraz cechy paradygmatu obiektowego, oraz sposobu jego realizacji w tym języku.

3. Rodzaj mutowanych konstrukcji języka. Zbiór mutacji wg. niniejszej klasyfikacji jest najbardziej uniwersalny wśród różnych języków programowania, ze względu na wspólne dla tych języków elementy:

   a) Wyrażenia. Operatory mutujące wyrażenia mają wpływ na całość pojedynczych wyrażeń, lub główne elementy ich składni. Wybrane operatory [1], to Trap On Statement Execution (STRP), Trap on if Condition (STRI), Usunięcie wyrażenia (SSDL), czy podmiana słowa kluczowego return (SRSR). Operatory te pozwalają weryfikować pokrycie testów, poprzez mutacje wszystkich ścieżek grafu przepływu sterowania programu.

   b) Operatory. Języki programowania dostarczają bardzo szeroki zbiór operatorów, by przytoczyć tylko operatory: arytmetyczne, logiczne, bitowe, przesunięć, arytmetyczne relacji $(* =, / =)$, bitowe relacji $(\& =, | =)$, przypisania $(=)$, relacji $(<, >, >=, <=, ==, ! =)$. Dodatkowo operatory te mogą być stosowane w różnych kontekstach, jak np. matematyka liczb całkowitych, zmiennoprzecinkowa, operacje na znakach, ciągach tekstowych, obiektach, wskaźnikach, czy wyrażeniach logicznych. Proste zdefiniowanie różnych operatorów dla każdego typu operatora i kontekstu zastosowania zaowocowałoby ogromnym zbiorem operatorów mutacji, dlatego wyróżniono jedynie główne rodzaje mutacji: zamiana operatora binarnego (binary operator replacement), zamiana operatora jednoargumentowego (uniary operator replacement). W ramach operatorów dwuargumentowych: zamiana operatorów porównywalnych (comparable) i nieporównywalnych (incomparable).

   c) Zmienne. Mutacje zmiennych modelują błędy związane z odwołaniami do złych zmiennych. Klasyfikacja mutacji zmiennych wiąże się z rodzajami zmiennych, a więc istnieją operatory zmiennych dla typów skalarnych, tablicowych, struktur i wskaźników. Przedstawiciele tej grupy mutacji to Zamiana odwołania do zmiennej skalarnej (scalar variable reference replacement - VSRR), oraz analogiczne mutacje dla innych typów zmiennych.

   d) Stałe. Mutacje stałych reprezentują błędy związane z przypadkową poprawnością i tym samym są podobne do operatorów zmiennych.

4. Cel mutacji, jakim jest pomiar pokrycia kodu testami (słabe mutacje), lub ocena dokładności odwzorowania kodu przez testy (silne mutacje). Podział ten omówiono na stronie 2.

2.1.4 Narzędzia

Pierwsze narzędzie wspierające testowanie mutacyjne zostało opracowane przez Timothiego Budda w ramach pracy doktorskiej „Mutation Analysis". Inne wybrane narzędzia to [18]:

1. Mothra, [9]. Zbiór narzędzi, z których każde wspierało inny etap analizy i testowania mutacyjnego. Dzięki temu, że każde z narzędzi pakietu jest osobną komendą, łatwo je integrować i eksperymentować z nimi.

2. Proteum to pierwszy program implementujący wszystkie operatory mutacji zaprojektowane dla języka ANSI C.

3. μJava to narzędzie do testowania mutacyjnego testów jednostkowych w Javie. Generuje mutacje zarówno dla tradycyjnych operatorów, jak i na poziomie klas.

4. muClipse to wtyczka μJava do środowiska Eclipse.

5. Jumble, narzędzie mutacji dla programów napisanych w Javie, oraz testów JUnit. Mierzy jakość testów, raportuje brakujące testy, poprzez mutacje bytecode'u klas programu. Może być uruchamiany z linii poleceń, oraz jako wtyczka do środowiska Eclipse.

6. JesTer, darmowe narzędzie do testów JUnit w języku Java, z „silnymi" operatorami mutacyjnymi.

7. PesTer, darmowe narzędzie do testów PyUnit w języku Python, z silnymi operatorami mutacyjnymi.

8. SQLMutation, system mutacyjny do zapytań SQL. Generuje zapytania „Select...".

9. Plextest C++, komercyjne narzędzie do „silnego" testowania mutacyjnego.

10. Heckle, darmowe narzędzie dla języka Ruby.

Wszystkie wyżej wymienione narzędzia reprezentują tradycyjny model testowania mutacyjnego, różniąc się oferowanymi operatorami mutacji, algorytmami optymalizacji, językiem implementacji. Żadne z dotychczas istniejących narzędzi nie wspiera testowania mutacyjnego przez wstrzeliwanie odpowiedzi.

2.2 Programowanie aspektowe

Istnieje wiele problematycznych zagadnień w programowaniu, których rozwiązania nie ułatwia ani programowanie strukturalne, ani obiektowe. Implementacja wielu ważnych z punktu specyfikacji wymagań oczekiwań wobec systemu, musi być rozrzucona po kodzie całej aplikacji, czyniąc kod źródłowy skomplikowanym i utrudniając jego zrozumienie i utrzymywanie. Rozwiązaniem dla takich problemów, niejako przecinających wskroś cały system jest programowanie aspektowe. Definiuje ono koncepcję aspektu, a więc pewnego wspólnego dla wielu dziedzin systemu zagadnienia (np. system bezpieczeństwa, autoryzacji, logowania), oraz specyfikację powiązania tego aspektu z systemem. Wykorzystanie programowania aspektowego pozwala zwiększyć przejrzystość kodu źródłowego programów ([8]).

2.2.1 Zasady działania

Etapy projektowania systemu, oraz implementacji są ze sobą wzajemnie powiązane. Projekt rozbija system na coraz mniejsze i mniejsze komponenty. Języki programowania z kolei dostarczają narzędzi do implementacji pojedynczych, małych i większych jednostek programu. Z tej perspektywy wszystkie języki programowania wspierają koncepcje abstrakcji i kompozycji. Pozwalają tworzyć hierarchie procedur/funkcji/obiektów, z których każda ma określone funkcje.

Aspekt, jeżeli może być zawarty w jednej procedurze, jest na ogół nie tyle jednostką funkcjonalną systemu, co spełnieniem pewnej cechy, lub własności, która ma wpływ na inne jednostki. Innymi słowy, typowym zastosowaniem dla aspektów są wymagania niefunkcjonalne systemu, lub funkcjonalne, ale nie związane z logiką biznesową, np. mechanizm synchronizacji, mechanizm dostępu do pamięci, lub obsługa błędów.

Celem programowania aspektowego jest oddzielenie implementacji jednostek funkcjonalnych od aspektów w kodzie programu, poprzez dostarczenie mechanizmów, które umożliwiają abstrakcję i łączenie ze sobą tych elementów w spójny system.

Przygotowanie programu w programowaniu aspektowym wymaga znajomości języka programowania komponentów, w którym napisane zostaną elementy funkcjonalne; znajomości języków aspektowych, w których napisane zostaną aspekty, oraz narzędzia tkającego aspekty w kodzie komponentów (ang. aspect weaver). Języki aspektowe mogą być zaprojektowane tak, że tkanie aspektów odbywa się w dwóch różnych trybach: przed uruchomieniem programu - w trakcie jego kompilacji, lub dynamicznie, w trakcie uruchomienia programu. Narzędzia do tkania aspektów w trakcie kompilacji na ogół jako wejście przyjmują kod programu i kod aspektów, oraz generują kod programu z wplecionymi aspektami, gotowy do kompilacji przy użyciu standardowego kompilatora. Narzędzia działające w trakcie uruchomienia programu na ogół wymagają odpowiedniego środowiska uruchomienia, np. maszyny wirtualnej.

2.2.2 AspectJ

AspectJ to rozszerzenie do języka Java wprowadzające koncepcję programowania aspektowego, [15]. AspectJ dodaje do Javy nowy koncept - punkt złączenia (ang. join point). Natomiast składnię wzbogaca o konstrukcje takie jak: przecięcie (ang. pointcut), poradę (ang. advice), deklaracje wewnątrz typów (ang. inter-type declarations), oraz aspekty.

Join point to jednoznacznie zdefiniowany punkt w grafie przepływu sterowania programu. Przecięcie grupuje zbiór punktów złączenia oraz wartości w tych punktach. Porada to fragment kodu programu wykonywany w miejscu punktu złączenia.

Deklaracje wewnątrz typów pozwalają modyfikować elementy składowe klas i powiązania pomiędzy klasami programu.

Aspekt to jednostka w AspectJ zawierająca implementację jednej cechy, przecinającej wiele komponentów programu. Jest zbliżona do klas języka Java, jednak dodatkowo może zawierać przecięcia i deklaracje wewnątrz typów.

2.2.3 Przykład

Fragment kodu źródłowego 3 w Javie, wykorzystującego AspectJ, obrazuje składnię prostego aspektu. Zdefiniowany aspekt o nazwie `SimpleTracing` specyfikuje jedno przecięcie kodu programu (`pointcut`) o nazwie `tracedCall`. Punktem przecięcia jest wywołanie `draw(GraphicsContext)` w klasie `FigureElement`. Punkt przecięcia jest powiązany także z jedną poradą, która ma być wywoływana przed wykonaniem metody `draw`. Kod porady powoduje wypisanie na standardowe wyjście programu krótkiej informacji.

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```

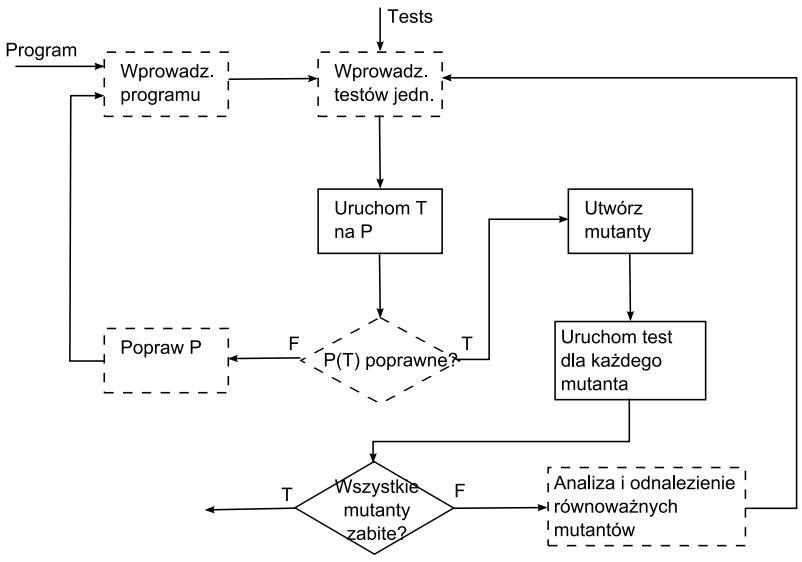
Listing 3: Implementacja przykładowego aspektu w AspectJ.

Rozdział 3

Opis opracowanego narzędzia

W ramach niniejszej pracy zaimplementowano środowisko do przeprowadzania testów mutacyjnych z wykorzystaniem programowania aspektowego, wg. koncepcji mutacji zaprezentowanej w [2]. Narzędzie to służy do oceny jakości testów jednostkowych napisanych w języku Java z wykorzystaniem biblioteki JUnit. Środowisko ma postać wtyczki do programu Eclipse, tym samym rozszerza funkcjonalność standardowego mechanizmu uruchamiania testów w tym programie.

Ogólny mechanizm działania środowiska przedstawia rys. 3.1. Różni się on od tradycyjnego, przedstawionego na rys. 2.1 kilkoma podstawowymi elementami. W trakcie uruchamiania testu, każde wywołanie z metody testu do klas systemu testowanego jest rejestrowane, a zwracana odpowiedź zapamiętywana. Jeżeli test zakończył się poprawnie, to dla każdego zarejestrowanego odwołania, na bazie oryginalnych zarejestrowanych odpowiedzi, tworzone są zmutowane odpowiedzi. Pojedynczy test jest następnie ponownie uruchamiany dla każdej zmutowanej odpowiedzi. Przy uruchomieniu testu ze zmutowaną odpowiedzią jest ona wstrzeliwana zamiast wywoływania metody z systemu. Jeżeli test zakończy się poprawnie, mimo wprowadzenia zmutowanej odpowiedzi, przyjmuje się że mutant przeżył, dlatego że nie miał wpływu na wynik testu. W przeciwnym razie mutacja kwalifikowana jest jako zabita.



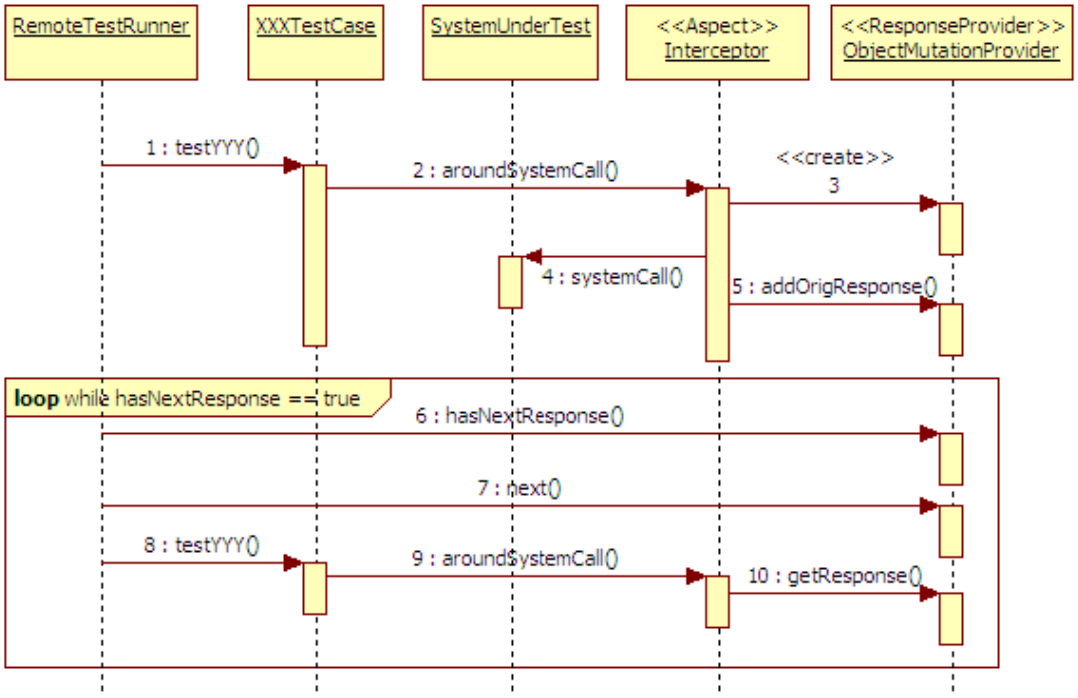
Rysunek 3.1: Schemat działania środowiska testowania mutacyjnego wykorzystującego aspekty.

Ograniczenie zakresu testów, które są wykonywane przy każdej mutacji, z całego zbioru tylko do tego jednego testu zawierającego mutację, opiera się na zasadzie, że testy jednostkowe powinny

być od siebie całkowicie niezależne. Tak więc mutant w jednym teście nie powinien mieć wpływu na zmiany wyników pozostałych testów. Ograniczenie to ma istotny wpływ na wydajność rozwiązania z testowaniem mutacyjnym korzystającym z aspektów, szczególnie w bardzo dużych zbiorach testów, które w oryginalnej koncepcji testowania mutacyjnego musiały być uruchamiane w całości, dla każdego mutanta.

Diagram 3.2 przedstawia mechanizm działania środowiska testowego z punktu widzenia programowania aspektowego. Osią rozwiązania jest aspekt `Interceptor`. Przy pierwszym uruchomieniu testu, dla każdego wywołania systemu w teście, aspekt ten tworzy generatory mutacji, klasy typu `ResponseProvider`. Generator mutacji jest także informowany o wartościach zwróconych przez system.

Następnie, dla każdego generatora mutacji dla testu, test ten jest uruchamiany ponownie tak długo, jak generator może wygenerować nową mutację. Takie rozwiązanie przenosi także odpowiedzialność za sprawdzanie unikalności mutacji ze środowiska testowego na generator. Powinien on dostarczać każdą unikatową mutację tylko jeden raz dla danego testu.



Rysunek 3.2: Proces uruchamiania poszczególnych mutacji dla testu.

3.1 Rodzaje mutacji

Proponowane środowisko pozwala na mutacje wszystkich wywołań metod w klasach pochodnych od typu `junit.framework.TestCase`, takich że wywołania te są ulokowane w metodach z nazwą zaczynającą się na `test` (a nie np. `setUp`, itp.), oraz są wywołaniami metod zwracających wartość (w przeciwieństwie do metod zwracających typ `void`).

Mutowane są odpowiedzi zarówno typów prostych, a więc `boolean`, `char`, `byte`, `int`, `float`, `double`, `short`, `long`, jak i obiektów. Dla każdego z tych typów możliwe jest zdefiniowanie osobnego generatora mutacji.

Tworzenie mutantów dla typów prostych jest zupełnie odmienne niż dla obiektów. Wynika to z faktu że typy proste są zwracane przez wartość i nie mają innych referencji w systemie. Zastąpienie

jednej wartości przez inną nie ma żadnych skutków ubocznych, ponieważ w momencie zastąpienia, poprzednia wartość przestaje istnieć.

W przeciwieństwie do typów prostych, obiekty zwracane są przez referencję. Mutacja może zatem przebiegać na dwa sposoby. Mutacji może ulegać referencja, tak by wskazywać na zupełnie inny obiekt zgodnego typu. W drugim przypadku mutacja może obejmować operacje na danej instancji obiektu, pierwotnie zwróconej.

W przypadku mutacji polegającej na zamianie referencji do oryginalnego obiektu na inną, nie ma gwarancji że była to jedyne wystąpienie referencji do danej instancji obiektu. Istnieje więc ryzyko skutków ubocznych w trakcie wykonywania testu, jeśli autor testów korzysta z porównań referencji.

Fragment kodu 4 ilustruje taki przykład testu. Metoda `addElements(list)` zwraca referencję do tego samego obiektu, który podano w jej argumencie. Choć w przykładowym fragmencie zwracanie tego samego obiektu, który jest podawany w argumencie wydaje się niezasadne, to istnieją przypadki kiedy takie rozwiązanie jest stosowane (np. metoda `StringBuffer.append()`, która zwraca referencję do obiektu `StringBuffer`, [16]). Wprowadzenie w miejscu wywołania metody `addElements(list)` mutacji, która zwracaną referencję zastąpi nową referencją innego obiektu spowoduje zakończenie testu błędem, nawet jeśli zwrócony obiekt jest co do wartości dokładnie taki sam jak oryginalny. Zakończenie testu błędem powinno oznaczać przeżycie mutacji, jednak w tym przypadku jest to błąd środowiska testowego, które nadużyło założeń, z jakimi została napisana metoda `addElements(list)`.

```
List addElements(List list) {
  list.add("element1");
  return list;
}

public void testSame() {
  List list = new ArrayList();
  List listWithNewElements = addElements(list);
  assertSame(list, listWithNewElements);
}
```

Listing 4: Skutki uboczne mutacji zmieniających referencje do zwracanych obiektów.

Drugi rodzaj mutacji obiektów polega na modyfikacji stanu zwracanego obiektu, w szczególności przez zmianę wartości jego pól, lub wywoływanie metod na jego rzecz. Rozwiązanie to jest wolne od problemów typowych dla mutacji obiektów zmieniających referencje, jednak wiąże się z nim inna komplikacja. Jak wspomniano, zwracany obiekt może mieć wiele referencji w systemie. W szczególności może to być obiekt współdzielony między testami, np. obiekt statyczny, niezmiennik programu – obiekt, który ze względu na duży koszt tworzenia, jest wykorzystywany miedzy testami (np. obiekt reprezentujący połączenie do bazy danych, okno aplikacji). Mutacja takiego obiektu, w szczególności uszkodzenie jego stanu, prowadzi do zakłóceń pozostałych testów.

Przykład takiego scenariusza zawarto we fragmencie kodu 5. Metoda testowana zwraca referencję do obiektu `US` typu singleton zdefiniowanego w klasie `Locale`. Ze względu na niezmienność cech lokalnych, takich jak kod kraju, czy języka lokalizacji, nie testuje się ich poprawności, ale zakłada zgodność z dokumentacją biblioteki standardowej Java [16].

Zakładając że środowisko mutacji przechwyci odpowiedź metody `getDefaultLocale()` w teście `testLocale()`, oraz zmodyfikuje wartość pola reprezentującego `displayName` tego obiektu, test zakończy się poprawnie. Jednak zmodyfikowana statyczna instancja `Locale.US` pozostanie w

pamięci maszyny wirtualnej, co doprowadzi do błędu w teście `testUnrelatedFailure()`.

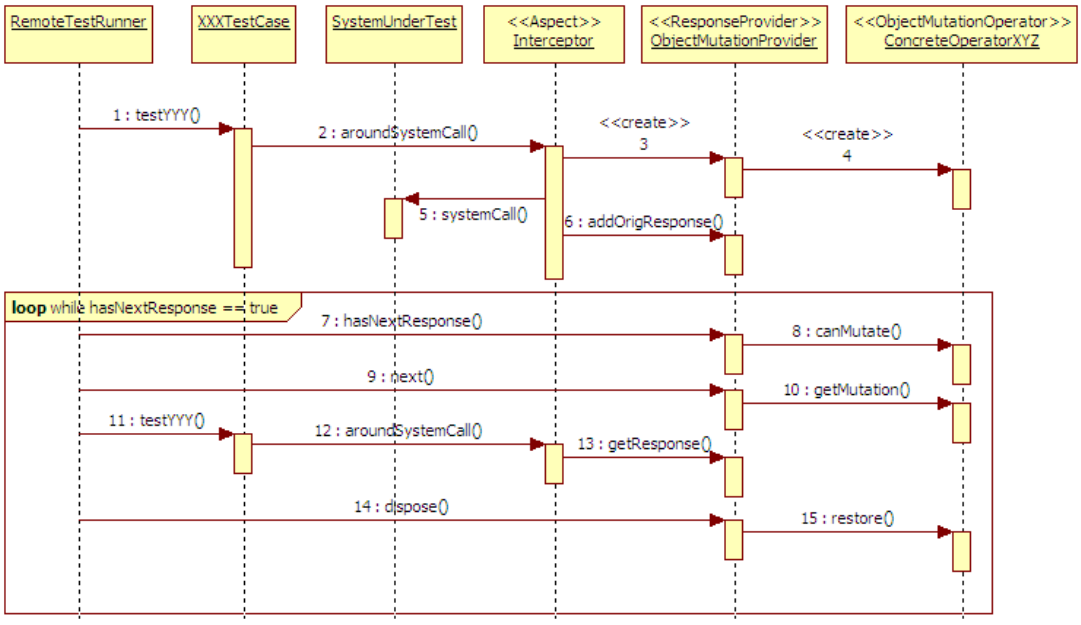
```
Locale getDefaultLocale() {
  return Locale.US;
}

public void testLocale() {
  Locale default = getDefaultLocale();
  assertSame(Locale.US, default)
}

public void testUnrelatedFailure() {
  assertEquals("US", getDefaultLocale().getDisplayName());
}
```

Listing 5: Skutki uboczne mutacji zmieniających wartość obiektów stałych systemu.

W prezentowanym środowisku zastosowano rozwiązanie gwarantujące niezmienność obiektów pomiędzy wywołaniami różnych testów. Cykl wykonania pojedynczej mutacji prezentuje rys. 3.3. W porównaniu do rys. 3.2, zauważalna zmiana to bezpośrednie zwolnienie mutacji poprzez wywołanie `dispose()` na rzecz obiektu `ResponseProvider`. Ponadto generator mutacji dla typów obiektowych sam nie dysponuje żadnymi standardowymi mutacjami. Jest jedynie pośrednikiem dla operatorów mutacji (typu `ObjectMutationOperator`). Każdy operator, musi zagwarantować że jest w stanie wygenerować mutację dla danego obiektu (`canMutate(Object)`), wygenerować mutację (`getMutation(Object)`), oraz przywrócić stan sprzed mutacji (`restore(Object)`). Generator mutacji dla metod zwracających obiekty obsługuje wiele operatorów mutacji i generuje mutacje tak długo, jak przynajmniej jeden z jego operatorów jest w stanie wygenerować mutację.



Rysunek 3.3: Szczegółowy proces uruchamiania poszczególnych mutacji dla testu.

3.2 Scenariusz zastosowań

Środowisko do testowania mutacyjnego z wykorzystaniem aspektów jest przeznaczone do użytku w trakcie pisania testów jednostkowych, ze szczególnym naciskiem na zwinne praktyki programowania, takie jak Test-Driven Development.

Jest to główny powód integracji mechanizmu oceny jakości testów, środowiska mutacyjnego, z jednym z najczęściej wykorzystywanych środowisk programistycznych dla języka Java - Eclipse. Etap implementacji systemu i testów, oraz ich bieżącego uruchamiania jest kluczowym momentem, kiedy programiście najłatwiej zmodyfikować i poprawić testy. Dlatego środowisko mutacyjne jest w pełni zintegrowane ze standardowym widokiem uruchamiania testów JUnit w Eclipse, a korzystanie z dodatkowych informacji opartych na bazie testów mutacyjnych jedynie nieznacznie różni się od standardowego uruchamiania testów jednostkowych. Typowy przypadek użycia jest przedstawiony w poniższym scenariuszu:

1. Programista wprowadza nowy fragment kodu źródłowego systemu, oraz testów jednostkowych.
2. Programista definiuje zbiór testów do uruchomienia.
3. Środowisko testowe wykonuje testy i wyświetla wyniki.
4. Środowisko mutacyjne wyświetla uzupełniające wyniki nt. jakości.
5. Programista, w razie potrzeby poprawia testy i wraca do pkt. 3.

Scenariusz różni się od typowego scenariusza uruchomienia testów jedynie dodatkowym krokiem 4. Krok ten jest wykonywany przez środowisko, zatem nie wymaga od programisty dodatkowej pracy, oczywiście z wyjątkiem analizy wyników.

Ponieważ natura procesu testowania, realizowanego przez człowieka, podlega subiektywnej ocenie, scenariusz zastosowań środowiska wymaga weryfikacji eksperymentalnej z prawdziwymi użytkownikami, piszącymi rzeczywiste przypadki testowe w trakcie implementacji aplikacji. Jednak badania eksperymentalne są silnie uzależnione od skuteczności środowiska w wykrywaniu błędów w testach, co z kolei jest zadaniem dobrze dobranych operatorów mutacji. Z tej perspektywy niniejsza praca jest jedynie wstępem do dalszych badań, dostarczającym narzędzi i środowiska do doboru i optymalizacji operatorów mutacji. W dalszej części pracy omówiono wybrane zaimplementowane operatory mutacji oraz ich skuteczność w wykrywaniu błędów w testach.

3.3 Możliwości dalszych badań

Środowisko umożliwia dalsze badania nad testami mutacyjnymi, API do pisania operatorów mutacyjnych. Integracja wyników wszystkich testów i raporty nie na poziomie testów jednostkowych, ale na poziomie odwołań do systemu.

Rozdział 4

Operatory mutacyjne

4.1 Charakterystyka operatorów

Zgodnie z cyklem życia operatora mutacyjnego, przedstawionym na rysunku 3.3, wszystkie operatory mutacji opracowane dla środowiska mutacyjnego wykorzystującego aspekty muszą implementować interfejs na listingu 6.

```
public interface ObjectMutationOperator {
  boolean canMutate(Object o);
  Object getMutation(Object o);
  void restore(Object o);
  String getName();
}
```

Listing 6: Interfejs operatorów mutacji dla obiektów.

Łatwość tworzenia nowych operatorów w prezentowanym środowisku najlepiej demonstruje przykład operatora RemoveListElement, na listingu 7. Operator ten mutuje tylko niepuste listy, poprzez usunięcie pierwszego elementu na liście. Jest to operator modyfikujący wskazywaną instancję obiektu, nie zwracający referencji do nowej listy, zatem musi także przywracać zmutowane wartości, by zapewnić niezależność pomiędzy testami.

Dla umożliwienia szerszej gamy implementacji operatorów mutacyjnych, badano różne sposoby ich implementacji, w szczególności następujące podejścia: bezpośrednią manipulację na obiektach, manipulacja poprzez refleksję, proxy.

Bezpośrednia manipulacja na obiektach to najprostszy sposób na zmianę działania systemu. Jest jednak ograniczony do tylko tych typów, które są znane w trakcie definiowania mutacji, jak np. standardowa biblioteka Java (np. kolekcje). Już na etapie definiowania operatora muszą być w nim zawarte odwołania do konkretnych klas, które będą miały podlegać mutacji. Natomiast sama mutacja jest ograniczona do możliwości jakie daje interfejs mutowanego typu, tj. w ramach mutacji możliwe jest tylko wywoływanie metod publicznych, lub modyfikacja pól publicznych. Przy tym rodzaju operatorów nie jest natomiast możliwa mutacja obiektów specyficznych dla testowanych systemów. Korzystając z tej metody zaimplementowano wszystkie operatory mutacji kolekcji, oraz `ObjectInsteadOfNull` (OION).

Manipulacja poprzez refleksję opiera się na mechanizmie refleksji języka Java. Mechanizm refleksji pozwala operować na obiektach o typach, które nie były znane w trakcie kompilacji kodu. API mechanizmu refleksji pozwala m.in. przeglądanie udostępnianego interfejsu typu, wywoływanie metod na rzecz obiektów, czy manipulacje na polach obiektów, wskazując na metody

```
public class RemoveListElement implements ObjectMutationOperator {

  private Object removedElement;

  public boolean canMutate(Object o) {
    return (o != null) && (o instanceof List)
           && (!((List) o).isEmpty());
  }

  public Object getMutation(Object o) {
    removedElement = ((List) o).remove(0);
    return o;
  }

  public String getName() {
    return "RLE";
  }

  public void restore(Object o) {
    ((List) o).add(0, removedElement);
  }
}
```

Listing 7: Przykład implementacji operatora mutacji.

lub pola przez ich nazwy. Operatory korzystające z mechanizmu refleksji mogą modyfikować stan obiektów, których typ nie jest znany w trakcie implementacji operatora, w szczególności w obiekty typów specyficznych dla testowanych systemów. Podobnie, jak w przypadku bezpośredniej mutacji na obiektach, także mutacja przez refleksję modyfikuje instancje obiektów, zatem musi także przywracać ich stan sprzed mutacji. Korzystając z refleksji zaimplementowano operatory `NullAllFields` (NAF) oraz `ZeroAllPrimitiveFields` (ZAPF).

Proxy to również mechanizm bazujący na refleksji. Pozwala na zdefiniowanie klas pośredniczących w wykonywaniu metod na rzecz wybranych interfejsów. Z punktu widzenia operatorów mutacyjnych umożliwia zatem przesłonienie implementacji metod interfejsów zmutowanymi implementacjami. Do poprawnego działania, referencja do obiektu pośrednika (proxy) musi zostać zwrócona w miejscu referencji do oryginalnego obiektu. Zatem w przeciwieństwie do dwóch poprzednich podejść, proxy podmienia referencje do obiektu, a nie modyfikuje instancji z testowanego systemu. Korzystając z mechanizmu proxy zaimplementowano operator `NullAllMethodsProxy` (NAMP).

W dalszej części rozdziału omówiono konkretne operatory mutacji. Dla każdego operatora przedstawiono rodzaj typowych błędów w kodzie testowanego systemu, jakie ma on wykrywać, przykład, oraz algorytm działania operatora.

4.2 Obsługa wartości null

Definiując operatory mutacyjne skupiono się na dwóch rodzajach typowych problemów popełnianych przez programistów. Pierwsza grupa, to obsługa wartości `null`. Klasycznym objawem błędów w obsłudze wartości `null` są wyjątki typu `NullPointerException`, a więc w sytuacji gdy nastąpiło odwołanie do pola lub metody klasy na referencji, która nie wskazuje na żaden obiekt.

4.2.1 NullAllFields (NAF)

Brak obsługi wartości `null` w polach obiektu. Testy niedostatecznie chronią przed potencjalnym wyjątkiem `NullPointerException` podczas odwołania do pustego pola obiektu.

Przykładowy test jednostkowy w listingu 8 jest podatny na mutację NAF. Nie zakłada bowiem, że `product` może mieć niezainicjalizowane pole `kind`. Poprawny test jednostkowy powinien zawierać sprawdzenie `assertNotNull(product.kind);`.

```
public void testFactory() {
  Product product = Factory.createProduct("pickup");
  assertEquals("pickup", product.kind.getName());
}
```

Listing 8: Przykład zastosowania operatora NullAllFields.

Operator mutacji, korzystając z refleksji podstawia wartość `null` we wszystkich polach zwracanego obiektu, dziedziczących po typie `java.lang.Object`.

4.2.2 NullAllMethodsProxy (NAMP)

Brak obsługi wartości `null` zwracanych przez metody obiektu. Testy niedostatecznie chronią przed potencjalnym wyjątkiem `NullPointerException` podczas wywołania metody obiektu.

Przykładowy test jednostkowy w listingu 9 jest podatny na mutację NAMP. Nie zakłada bowiem, że metoda `Product.getKind()` może zwracać `null`. Poprawny test jednostkowy powinien zawierać sprawdzenie `assertNotNull(product.getKind());`.

```
public void testFactory() {
  Product product = Factory.createProduct("pickup");
  assertEquals("pickup", product.getKind().getName());
}
```

Listing 9: Przykład zastosowania operatora NullAllFieldsProxy.

Operator mutacji, podmienia oryginalny obiekt zwracany przez `createProduct("pickup")` pośrednikiem, który dla wywołań dowolnych metod interfejsu `Product` zwraca wartość `null`.

4.2.3 ObjectInsteadOfNull (OION)

Niedostateczna weryfikacja wartości `null` zwracanej przez wywołanie metody. Wartość `null` jest na ogół zwracana w sytuacjach skrajnych. Test mający gwarantować że wartość `null` zostanie zwrócona jest niekompletny.

Przykładowy test jednostkowy w listingu 10 jest podatny na mutację OION. Zakładając że metoda `Map.get(String)` zwraca `null`. Zwrócenie wartości `null` nie jest bezpośrednio testowane, a potencjalnie pusty wynik jest wykorzystywany w dalszej części testu. Poprawny test jednostkowy powinien zawierać sprawdzenie `assertNull(value);`.

Operator generuje mutacje tylko dla tych wywołań systemu, które zwracają `null`. W miejsce wartości pustej wstrzeliwuje obiekt o typie zgodnym z zadeklarowanym typem zwracanym przez metodę.

```
public void testNoUser() {
  Map usersMap = new HashMap();
  Value value =  usersMap.get("NonExistentKey");
  assertNull(Processor.process(value));
}
```

Listing 10: Przykład zastosowania operatora ObjectInsteadOfNull.

4.2.4 ZeroAllPrimitiveFields (ZAPF)

Brak obsługi wartości 0 w polach obiektu. Testy nie dostatecznie weryfikują wartości pól obiektu.

Przykładowy test jednostkowy w listingu 11 jest podatny na mutację ZAPF. Test nie sprawdza poprawności zwróconego obiektu. Powinien zawierać asercję `assertEquals(4, truck.wheels)`

```
class Car {
  int wheels;
}

public void testTruck() {
  Car truck = Factory.createCar("truck");
  assertNotNull(truck);
}
```

Listing 11: Przykład zastosowania operatora ZeroAllPrimitiveFields.

Operator mutacji, korzystając z refleksji podstawia wartość 0, lub `false` we wszystkich polach typów podstawowych zwracanego obiektu.

4.3 Błędy obsługi kolekcji

Druga grupa operatorów mutacji bazuje na wybranych operatorach z [11], dotyczących operacji na kolekcjach. Biblioteka kolekcji języka Java jest jednym z najczęściej wykorzystywanych fragmentów biblioteki standardowej, a stosowanie jej wiąże się z grupą standardowych problemów, do których wykrywania przygotowano zbiór operatorów.

4.3.1 EmptyCollection (EC)

Mutacja imituje błędy związane ze wstawieniem elementów do złej kolekcji, lub przypadkowym usunięciem elementów z kolekcji.

Przykładowy fragment kodu źródłowego 12 ilustruje błąd programisty symulowany przez mutację. W konstruktorze klasy `Calendar` miesiące błędnie zostały dodane do listy `seasons`, zamiast do `months`. Niedokładny test nie sprawdza zawartości listy zwracanej przez `Calendar.getMonths()`, co powoduje że błąd w konstruktorze nie zostanie wykryty w porę. Rozwiązanie problemu to dodanie w teście `assertFalse(months.isEmpty())`.

4.3.2 AddElementToList (AETL)

Mutacja imituje przypadkowe wstawienie nadmiarowego elementu listy.

Przykładowy fragment kodu źródłowego 13 ilustruje błąd programisty symulowany przez mutację. W konstruktorze klasy `Calendar` do listy `seasons` błędnie dodano nadmiarowy element. Niedokładny test nie sprawdza zawartości listy zwracanej przez `Calendar.getSeasons()`, co powoduje

```
class Calendar {
  private list months = new ArrayList();
  private list seasons = new ArrayList();
  public Calendar() {
    seasons.add("spring");
    seasons.add("summer");
    seasons.add("autumn");
    seasons.add("winter");

    seasons.add("January");
    seasons.add("February");
    seasons.add("March");
  }
  public List getMonths() {
    return months;
  }
}
public void testMonths() {
  List months = calendar.getMonths();
  assertNotNull(months);
}
```

Listing 12: Przykład zastosowania operatora EmptyCollection.

że błąd w konstruktorze nie zostanie wykryty w porę. Jednym z możliwych rozwiązanie jest dodanie w teście `assertEquals(4,seasons.getSize())`

```
class Calendar {
  private list seasons = new ArrayList();
  public Calendar() {
    seasons.add("spring");
    seasons.add("summer");
    seasons.add("autumn");
    seasons.add("winter");
    seasons.add("January");
  }
  public List getSeasons() {
    return seasons;
  }
}
public void testMonths() {
  List seasons = calendar.getSeasons();
  assertNotNull(seasons);
}
```

Listing 13: Przykład zastosowania operatora AddElementToList.

4.3.3 RemoveListElement (RLE)

Mutacja imituje przypadkowe usunięcie elementu listy.

Przykładowy fragment kodu źródłowego 14 ilustruje błąd programisty symulowany przez mutację. W konstruktorze klasy `Calendar` do listy `seasons` błędnie dodano tylko trzy z czterech pór roku. Niedokładny test nie sprawdza zawartości listy zwracanej przez `Calendar.getSeasons()`, co powoduje że błąd w konstruktorze nie zostanie wykryty w porę. Rozwiązanie to dodanie w

teście `assertEquals(4,seasons.getSize())`

```
class Calendar {
  private list seasons = new ArrayList();
  public Calendar() {
    seasons.add("spring"); seasons.add("summer"); seasons.add("autumn");
  }
  public List getSeasons() {
    return seasons;
  }
}

public void testMonths() {
  List seasons = calendar.getSeasons();
  assertNotNull(seasons);
}
```

Listing 14: Przykład zastosowania operatora RemoveListElement.

4.3.4 IteratorSkip (IS)

Mutacja imituje nadmiarowe odwołanie do metody `Iterator.next()`.

Przykładowy fragment kodu źródłowego 15 ilustruje błąd programisty symulowany przez mutację. Mutacja IS spowoduje, że iterator zwrócony w teście przez `calendar.getSeasons()` będzie wskazywał nie na pierwszy, a od razu na drugi element reprezentowanej kolekcji, imituje więc brak pierwszego elementu kolekcji. Rozwiązanie polega na poprawieniu testu `testSeasons` w taki sposób, aby dokładniej weryfikował otrzymany iterator.

```
class Calendar {
  private list seasons = new ArrayList();
  public Calendar() {
    // seasons.add("spring");
    seasons.add("summer"); seasons.add("autumn"); seasons.add("winter");
  }
  public Iterator getSeasons() {
    return seasons.iterator();
  }
}
public void testSeasons() {
  Iterator seasons = calendar.getSeasons();
  assertNotNull(seasons);
}
```

Listing 15: Przykład zastosowania operatora IteratorSkip.

Rozdział 5

Analiza działania

Opracowane środowisko testów mutacyjnych zostało przetestowane na wybranych bibliotekach zbioru narzędzi Apache Commons: math v.1.2, dbutils v.1.1, oraz collections v.3.2.1. Biblioteki te zostały wybrane ze względu na stabilność i duży zbiór testów jednostkowych. Pokrycie testami jednostkowymi w tych bibliotekach prezentuje tabela 5.1. Pomiar pokrycia został wyliczony przy pomocy programu EclEmma ([17]), korzystając z opcji "line counters".

Projekt	Pokrycie [%]	Pokrytych linii	Razem linii
math	91, 2	8519	9337
dbutils	57, 7	370	641
collections	81, 7	11296	13830

Tablica 5.1: Pokrycie testami jednostkowymi bibliotek wybranych do analizy.

Testy miały na celu wykazać dodatkowe narzuty czasowe związane z przeprowadzaniem testów mutacyjnych, w porównaniu do standardowego środowiska testowego. Drugim celem była analiza skuteczności operatorów na testach wybranych bibliotek.

W tabeli 5.2 zawarto listę testowanych pakietów, oraz wyniki testów. Bibliotekę collections testowano nie w całości, a wybrane pakiety, dlatego że jest nieproporcjonalnie duża w porównaniu do dwóch pozostałych bibliotek, co utrudniało porównanie.

Projekt	L.testów	L.błędów (error)	L.defektów (failure)
math	1172	0	0
dbutils	1394	0	0
collections-collections	2518	0	0
collections-bidimap	3444	0	0
collections-iterators	382	0	0

Tablica 5.2: Zbiory testów jednostkowych, które testowano, wraz z wynikami działania.

Jako środowisko testowe wykorzystano Eclipse v. 3.4M7 i AspectJ 1.6.0, oraz wersję 1.0.0 proponowanego środowiska.

5.1 Analiza złożoności czasowej

W tym punkcie przedstawiono porównanie czasu potrzebnego na wykonanie testów jednostkowych na testach wymienionych w tabeli 5.2. Do porównania czasu wykorzystano standardowy mechanizm mierzenia czasu na poziomie całego zbioru testów, pakietów, oraz poszczególnych przypadków testowych - dostępny standardowo w środowisku Eclipse. Wartości te nie uwzględniają

zatem czasu kompilacji testów jednostkowych i testowanego programu, ani czasu potrzebnego na ich instrumentację w przypadku testów mutacyjnych. Jednak czas instrumentacji można pominąć z uwagi na fakt że jej koszt zależy tylko od liczby klas w systemie. Z punktu widzenia złożoności obliczeniowej, jest to więc jedynie stała dodana do potrzebnego czasu kompilacji.

Wyniki pomiarów prezentuje tabela 5.3. Korzystając ze zmierzonych wartości, oraz liczby mutacji i liczby testów w poszczególnych zbiorach, oszacowano średni czas wykonania pojedynczego testu i mutacji (tabela 5.4).

Projekt	JUnit [sec]	Mutacyje [sec]	Δ [sec]	L.mutacji
math	11, 10	15, 20	4, 10	4408
dbutils	0, 43	2, 71	2, 28	6919
collections-collections	1, 82	8, 01	6, 19	22930
collections-bidimap	1, 76	3, 85	2, 09	21403
collections-iterators	0, 10	0, 42	0, 32	1624

TABLICA 5.3: Czas wykonania testów dla wybranych zbiorów. JUnit - czas wykonania w standardowym środowisku JUnit Eclipse; Mutacje - czas wykonania w opracowywanym środowisku mutacyjnym; Δ - różnica czasu $Muacje - JUnit$

Projekt	śr.test [msec]	śr.mutacja [msec]	mutacja [% testu]
math	9, 47	0, 93	9, 82
dbutils	0, 30	0, 32	106, 82
collections-collections	0, 72	0, 26	37, 34
collections-bidimap	0, 51	0, 09	19, 13
collections-iterators	0, 26	0, 19	75, 27

TABLICA 5.4: Oszacowanie średniego czasu wykonania testu i mutacji oraz porównanie.

Z porównania wynika że wykonanie pojedynczej mutacji może zajmować nawet do 9% czasu wykonania testu. W najgorszym przypadku, wśród badanych zbiorów testów, wykonanie mutacji zajmuje w przybliżeniu czas potrzebny na wykonanie jednego testu.

Należy zwrócić uwagę, że średni czas wykonania testów w przypadku biblioteki math jest znacznie dłuższy niż w pozostałych bibliotekach. Wynika to z faktu że testy jednostkowe biblioteki math zawierają także testy wydajnościowe. Testy takie zajmują zdecydowanie więcej czasu:

Test	czas [sec]
HypergeometricDistributionTest.testLargeValues()	1, 03
HypergeometricDistributionTest.testMoreLargeValues()	5, 50

Widać zatem że testowanie mutacyjne z uwzględnieniem aspektów nie zwiększa w istotny sposób czasu poterzebnego na przeprowadzenie testów, co jest zupełnie nową cechą w porównaniu do tradycyjnych testów mutacyjnych. Duży rozrzut czasu potrzbnego czasu wynika z wielu czynników, jak np. rodzaj użytych operatorów mutacji, czy liczba i złożoność testów.

5.2 Podsumowanie działania operatorów

W tabeli 5.5 zawarto liczbę wygenerowanych mutacji wg. rodzajów operatorów, uwzględniając liczbę żywych oraz wszystkich mutacji.

Na pierwszy rzut oka zauważalny jest fakt, że dla biblioteki math nie wygenerowano żadnej mutacji typu EC, RLE, IS ani AETL, tj. mutacji dla kolekcji. Wynika to z faktu że w testach math nie korzystano z kolekcji. Jest to typowy wniosek który można wyciągnąć na podstawie tzw. słabych testów mutacyjnych, czyli dotyczących pokrycia.

Projekt	NAMP	SIRP	ZAPF	OION	NAF
math	86/350	50/2022	150/1098	0/11	191/927
dbutils	0/425	0/1479	34/2108	34/272	255/2091
collections-collections	683/2214	3715/7725	883/2620	151/620	1703/4273
collections-bidimap	1077/2387	4852/7920	1148/2633	114/303	1529/3596
collections-iterators	97/159	183/522	158/280	7/22	209/397

Projekt	EC	RLE	IS	AETL
math	0/0	0/0	0/0	0/0
dbutils	0/119	0/187	0/51	0/187
collections-collections	800/1749	848/1681	154/359	854/1693
collections-bidimap	541/1648	338/1196	213/524	338/1196
collections-iterators	1/20	0/17	99/188	0/17

Tablica 5.5: Liczba mutacji wg. rodzajów operatorów. Liczba mutacji w formie żywych/wszystkich. Górna tabela zawiera operatory null, oraz SIRP, a dolna operatory kolekcji.

Biorąc pod uwagę łączny stosunek liczby żywych do wszystkich mutacji (Tab.5.6) najmniej (bo jedynie 4%) żywych mutacji pozostało w projekcie dbutils. Łączna ocena mutacji jest także interesująca w kontekście informacji nt. pokrycia (Tab. 5.1). Projekt dbutils - mimo najniższego stopnia pokrycia ma najmniej żywych mutacji, natomiast projekty zbioru testów collections mają stosunkowo dużą liczbę żywych mutacji (ok. 45%, mimo bardzo wysokiego poziomu pokrycia (81, 7%). Fakt ten zależy oczywiście w głównej mierze od jakości generowanych mutacji. Dokładniejsza analiza byłaby potrzebna aby oszacować liczbę fałszywych przypadków żywych mutacji - tj. takich że mutacja przeżyła, mimo że nie wiąże się z błędem w teście. Typowy przykład fałszywych przypadków żywych mutacji, to mutacje tożsame z oryginalnym wynikiem.

Projekt	Żywych	L.Mutacji	Żywych/Razem [%]
math	477	4408	10
dbutils	323	6919	4
collections-collections	9791	22930	42
collections-bidimap	10150	21403	47
collections-iterators	754	1624	46

Tablica 5.6: Podsumowanie wyników testów mutacyjnych.

5.3 Przegląd przykładów

W tym punkcie zawarto przegląd niektórych przypadków testowych wykrytych przez środowisko mutacyjne i omówiono wykryty błąd. Dla każdego operatora mutacji przedstawiono także przykład zabitego mutanta.

5.3.1 NullAllFields

Listing 16 przedstawia przykład testu z żywym mutantem NullAllFields z testów biblioteki `commons-math`. Mutacji uległa wartość zwracana przez `getUnivariateStatistic()`, jak przedstawiono na rys. 5.1. Na podstawie wyniku testu można zatem twierdzić, że pole `variance` typu `StandardDeviation` nie jest wykorzystywane w zamieszczonym teście.

Listing 17 przedstawia przykład testu z zabitym mutantem NullAllFields z testów biblioteki `commons-math`. Mutacji uległo wywołanie metody `ComplexFormat.getRealFormat`. Asercja, która wykryła błąd (`assertSame`) zwróciła komunikat „expected not same”.

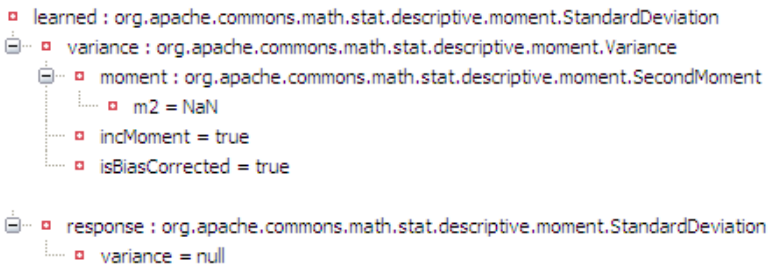
```
public void testMomentSmallSamples() {
  UnivariateStatistic stat = getUnivariateStatistic();
  if (stat instanceof SecondMoment) {
    SecondMoment moment = (SecondMoment) getUnivariateStatistic();
    assertTrue(Double.isNaN(moment.getResult()));
    moment.increment(1d);
    assertEquals(0d, moment.getResult(), 0);
  }
}
```

Listing 16: Przykład zastosowania operatora NullAllFields. Żywy mutant

```
public void testGetRealFormat() {
  NumberFormat nf = NumberFormat.getInstance();
  ComplexFormat cf = new ComplexFormat();

  assertNotSame(nf, cf.getRealFormat());
  cf.setRealFormat(nf);
  assertSame(nf, cf.getRealFormat());
}
```

Listing 17: Przykład zastosowania operatora NullAllFields. Zabity mutant



Rysunek 5.1: Obiekt oryginalnie zwracany przez metodę przedstawia gałąź „learned". Obiekt zmutowany znajduje się poniżej tej gałęzi.

5.3.2 NullAllMethodsProxy

Listing 18 przedstawia przykład testu z żywym mutantem NullAllMethodsProxy z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez metodę `nullFactory()` klasy `FactoryUtils`. Przykład świadczy o fałszywym żywym mutancie, dlatego że podstawiona zmutowana klasa factory dla wywołania metody `create()` zwraca null, czyli dokładnie taką odpowiedź, jaka była oczekiwana w teście. Niestety bez wcześniejszego wywołania oryginalnej metody nie jest możliwe sprawdzenie jaką wartość ona zwróci, zatem i mutacja nie jest w stanie uniknąć generowania takiego fałszywego mutanta.

Listing 19 przedstawia przykład testu z zabitym mutantem NullAllMethodsProxy z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `BagUtils.synchronizedBag(Bag)`. Asercja, która wykryła błąd (`assertSame`) zwróciła komunikat „Returned object should be a SynchronizedBag.".

Dla tego przykładu nie są prezentowane struktury obiektów, dlatego że operator nie zmienia pól obiektów, a podstawia własny obiekt (Proxy) bez żadnych pól.

```
public void testNullFactory() {
  Factory factory = FactoryUtils.nullFactory();
  assertNotNull(factory);
  Object created = factory.create();
  assertNull(created);
}
```

Listing 18: Przykład zastosowania operatora NullAllMethodsProxy. Żywy mutant

```
public void testSynchronizedBag() {
  Bag bag = BagUtils.synchronizedBag(new HashBag());
  assertTrue("Returned object should be a SynchronizedBag.",
  bag instanceof SynchronizedBag);
  try {
    bag = BagUtils.synchronizedBag(null);
    fail("Expecting IllegalArgumentException for null bag.");
  } catch (IllegalArgumentException ex) {
    // expected
  }
}
```

Listing 19: Przykład zastosowania operatora NullAllMethodsProxy. Zabity mutant

5.3.3 ObjectInsteadOfNull

Listing 20 przedstawia przykład testu z żywym mutantem ObjectInsteadOfNull z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `Map.Entry.getKey()`, w tym przypadku testowym zwracającej null.

Listing 21 przedstawia przykład testu z zabitym mutantem ObjectInsteadOfNull z testów biblioteki `commons-dbutils`. Mutacji uległo wywołanie metody `ResultSetHandler.handle(ResultSet)`. Błąd został wykryty dzięki asercji `assertNull(results)`.

Dla tego przykładu nie są prezentowane struktury obiektów, dlatego że operator nie zmienia pól obiektów, a podstawia wartość `new Object()`.

```
public void testEntrySetRemove1() {
  if (!isRemoveSupported()) return;
  resetFull();
  int size = map.size();
  Set entrySet = map.entrySet();
  Map.Entry entry = (Map.Entry) entrySet.iterator().next();
  Object key = entry.getKey();

  assertEquals(true, entrySet.remove(entry));
  assertEquals(false, map.containsKey(key));
  assertEquals(size - 1, map.size());
}
```

Listing 20: Przykład zastosowania operatora ObjectInsteadOfNull. Żywy mutant

5.3.4 ZeroAllPrimitiveFields

Listing 22 przedstawia przykład testu z żywym mutantem ZeroAllPrimitiveFields z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `PredicateUtils.allPredicate()`,

```
public void testEmptyResultSetHandle() throws SQLException {
  ResultSetHandler h = new ScalarHandler();
  Object results = h.handle(this.emptyResultSet);
  assertNull(results);
}
```

Listing 21: Przykład zastosowania operatora ObjectInsteadOfNull. Zabity mutant

jednak nie jest ona w żaden sposób weryfikowana, dlatego mutant przeżył.

Listing 23 przedstawia przykład testu z zabitym mutantem ZeroAllPrimitiveFields z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `getList().listIterator()`. Błąd został wykryty dzięki asercji w `backwardTest()`, z komunikatem „Iterator should have next".

```
public void testAllPredicateEx5() {
  PredicateUtils.allPredicate(Collections.EMPTY_LIST);
}
```

Listing 22: Przykład zastosowania operatora ObjectInsteadOfNull. Żywy mutant

```
public void testListListIterator() {
  resetFull();
  forwardTest(getList().listIterator(), 0);
  backwardTest(getList().listIterator(), 0);
}
```

Listing 23: Przykład zastosowania operatora ObjectInsteadOfNull. Zabity mutant

Rysunek 5.2: Obiekt oryginalny przedstawia gałąź „learned". Obiekt zmutowany przedstawia gałąź „response". Przykłady dotyczą listingu 23.

5.3.5 EmptyCollection

Listing 24 przedstawia przykład testu z żywym mutantem EmptyCollection z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `makeFullList()`. Metoda ta zwraca listę z wieloma elementami, jednak nie wpływają one w żaden sposób na wynik testu. Można zatem uznać że metoda ta niepotrzebnie komplikuje test i mogłaby być zastąpiona prostym i czytelnym `new ArrayList()`.

Listing 25 przedstawia przykład testu z zabitym mutantem EmptyCollection z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `CollectionUtils.union`. Błąd został wykryty.

```
public void testListAddByIndexBoundsChecking2() {
  if (!isAddSupported()) {
    return;
  }
  List list;
  Object element = getOtherElements()[0];
  try {
    list = makeFullList();
    list.add(Integer.MIN_VALUE, element);
    fail("List.add should throw " +
         "IndexOutOfBoundsException [Integer.MIN_VALUE]");
  } catch (IndexOutOfBoundsException e) {
    // expected
  }
...
```

Listing 24: Przykład zastosowania operatora EmptyCollection. Żywy mutant

```
public void testUnion() {
        Collection col = CollectionUtils.union(collectionA,collectionB);
        Map freq = CollectionUtils.getCardinalityMap(col);
        assertEquals(new Integer(1),freq.get("a"));
...
}
```

Listing 25: Przykład zastosowania operatora EmptyCollection. Zabity mutant

5.3.6 AddElementToList

Listing 26 przedstawia przykład testu z żywym mutantem AddElementToList z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `CollectionUtils.union()`. Test weryfikuje jedynie pierwsze pięć elementów kolekcji, swobodnie zakładając że są to jedyne elementy w kolekcji.

Listing 27 przedstawia przykład testu z zabitym mutantem AddElementToList z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `CollectionUtils.collect()`. Mutant został zabity w asercji porównującej rozmiary obu kolekcji, w trzeciej linii testu.

```
public void testUnion() {
  Collection col = CollectionUtils.union(collectionA,collectionB);
  Map freq = CollectionUtils.getCardinalityMap(col);
  assertEquals(new Integer(1),freq.get("a"));
  assertEquals(new Integer(4),freq.get("b"));
  assertEquals(new Integer(3),freq.get("c"));
  assertEquals(new Integer(4),freq.get("d"));
  assertEquals(new Integer(1),freq.get("e"));
...
```

Listing 26: Przykład zastosowania operatora AddElementToList. Żywy mutant

5.3.7 RemoveListElement

Listing 28 przedstawia przykład testu z żywym mutantem RemoveListElement z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `makeFullList()`. Test

```
public void testCollect() {
  Transformer transformer = TransformerUtils.constantTransformer("z");
  Collection collection = CollectionUtils.collect(collectionA,
    transformer);
  assertTrue(collection.size() == collectionA.size());
  assertTrue(collectionA.contains("a") && ! collectionA.contains("z"));
  assertTrue(collection.contains("z") && !collection.contains("a"));
...
```

Listing 27: Przykład zastosowania operatora AddElementToList. Zabity mutant

weryfikuje aspekty listy związane z metodą `get()`, jednak brak jednego elementu na liście pozostaje bez wpływu na wynik testu.

Listing 29 przedstawia przykład testu z zabitym mutantem RemoveListElement z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `getFullElements()`. Mutant został zabity w trakcie porównywania elementów listy i elementów oczekiwanych.

```
public void testListGetByIndexBoundsChecking2() {
  List list = makeFullList();
 try {
    list.get(Integer.MIN_VALUE);
    fail("List.get should " +
         "throw IndexOutOfBoundsException [Integer.MIN_VALUE]");
  } catch (IndexOutOfBoundsException e) {
    // expected
  }
...
```

Listing 28: Przykład zastosowania operatora RemoveListElement. Żywy mutant

```
public void testListGetByIndex() {
  resetFull();
  List list = getList();
  Object[] elements = getFullElements();
  for (int i = 0; i < elements.length; i++) {
    assertEquals("List should contain correct elements", elements[i],
      list.get(i));
    verify();
  }
}
```

Listing 29: Przykład zastosowania operatora RemoveListElement. Zabity mutant

5.3.8 IteratorSkip

Listing 30 przedstawia przykład testu z żywym mutantem IteratorSkip z testów biblioteki `commons-collections`. Mutacji uległa wartość zwracana przez `makeFullListIterator()`. Podobnie jak w przypadku żywego mutanta RemoveListElement, tak i ten przeżył, dlatego że test weryfikował inny scenariusz (dodawanie i usuwanie elementu), niż przewidziany w mutacji. Wynik taki można rozpatrywać dwojako. Jako fałszywe trafienie operatora mutacji - operator wskazujący błędy tam gdzie w rzeczywistości kod jest poprawny jest zły, ponieważ obniża zaufanie do po-

zostałych wyników. Z drugiej strony operator dostarcza informacji o pokryciu, tj. wskazuje testy które weryfikują poprawność funkcji Iteratora związanych z wywołaniem metody `next()`.

Listing 31 przedstawia przykład testu z zabitym mutantem IteratorSkip z testów biblioteki `commons-collections`. Mutacji uległo wywołanie metody `makeFullIterator()`. Mutant został zabity w trakcie kolejnych wywołań metody `next()`.

```
public void testAddThenRemove() {
  ListIterator it = makeFullListIterator();
  // add then remove
  if (supportsAdd() && supportsRemove()) {
    it.next();
    it.add(addSetValue());
    try {
      it.remove();
      fail("IllegalStateException must be thrown from remove after add");
    } catch (IllegalStateException e) {
    }
  }
}
```

Listing 30: Przykład zastosowania operatora RemoveListElement. Żywy mutant

```
public void testRemove() {
  Iterator it = makeFullIterator();
  if (supportsRemove() == false) {
    try { it.remove();
    } catch (UnsupportedOperationException ex) {}
    return; }
  try { // should throw IllegalStateException before next() called
    it.remove(); fail(); } catch (IllegalStateException ex) {}
  verify();
  // remove after next should be fine
  it.next(); it.remove();
  try { // should throw IllegalStateException for second remove()
  it.remove(); fail(); } catch (IllegalStateException ex) {}
}
```

Listing 31: Przykład zastosowania operatora RemoveListElement. Zabity mutant

Rozdział 6

Zakończenie

Praktyczne zastosowanie aspektów w testowaniu mutacyjnym w taki sposób jak zaproponowano w [2] jest podejściem nowym. Niniejsza praca miała na celu stworzenie funkcjonalnego i prostego w obsłudze środowiska, które zaoferuje testerowi rozszerzoną informację zwrotną o jakości napisanych testów już w trakcie ich pisania i uruchamiania.

W ramach pracy zintegrowano istniejące narzędzia, takie jak Eclipse, JUnit, kompilator AspectJ, do postaci jednolitego środowiska testowego, które w oparciu o łatwo rozszerzalną bazę operatorów mutacji, dostarcza czytelnej informacji na temat potencjalnych wad testów. Niniejsze opracowanie zawiera także dokumentację zastosowanych operatorów, dzięki której informacje ze środowiska testowego można w prosty sposób zinterpretować i wykorzystać do poprawienia testu.

Duża wydajność proponowanego rozwiązania przełamuje stereotyp testowania mutacyjnego i pozwala skoncentrować się na nowych obszarach badań, takich jak doskonalsze operatory mutacji, które przy mniejszej liczbie fałszywych trafień będą dostarczać bardziej precyzyjnej informacji na temat błędów; oraz szerszej analizy dużej ilości danych, jakie dostarczają wyniki mutacji. Szczególnie w ujęciu z perspektywy testowanych pakietów i klas systemu, a nie tylko testów jednostkowych, jak zaproponowano w pracy.

Literatura

[1] Agrawal H., DeMillo R. A., Hathaway B., et al, Design of Mutant Operators for the C programming language, Technical Report SERC-TR-41-PL Software Engineering Research Center, Purdue University, 1989

[2] Bogacki B., Walter B., Evaluation of test code quality with aspect-oriented mutations w Lecture Notes in Computer Science, 2006, s. 202-204

[3] DeMillo R. A., Lipton R. J., Sayward F. G., Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34 (41, April 1978)

[4] Dijkstra E., Notes on Structured Programming, 1969

[5] Gamma E, Beck K., Test infected: programmers love writing tests. The Java Report, 3(7):37-50, 1998.

[6] Goodenough John B., Gerhart Susan L., Toward a Theory of Test Data Selection" Proc. International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, June 1975, s. 493-510.

[7] Hamlet R. G., Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering, 3(4), 1977.

[8] Kiczales G., Lamping J., Mendhekar A., et al., Aspect-Oriented Programming, Finland, Springer-Verlag LNCS 1241, 1997.

[9] Offutt A. J. , A Practical System for Mutation Testing: Help for the Common Programmer, George Mason University, na International Test Conference, 1994. Proceedings., s. 824-830

[10] Offutt A. J., Voas J., Payne J., Mutation Operators for Ada. Technical Report ISSE-TR-96-09, George Mason University, 1996.

[11] Roger T. A., James M. B., Sudipto G., Bixia J., Mutation of Java Objects., na 13th International Symposium o Software Reliability Engineering, 2002. ISSRE 2002. Proceedings., s. 341-351

[12] Sarala S., Valli S., An Automatic defect detection for C++ programs, The 2004 Asian International Workshop on Advanced Reliability, 2004r., s. 419-426

[13] Vincenzi A. M. R., Maldonado J. C., Barbosa E. F., Delamaro M. E., Unit and integration testing strategies for C programs using mutation-based criteria. Symposium on Mutation Testing, strony 56-57, San Jose, CA, 2000

[14] Williams T., Mercer M., Mucha J., Kapur R., Code coverage, what does it mean in terms of quality?, Reliability and Maintainability Symposium, 2001. Proceedings

[15] AspectJ Programming Guide, http://www.eclipse.org/aspectj/doc/released/progguide/index.html, odwiedzana w czerwcu 2008

[16] Dokumentacja Języka Java, http://java.sun.com/j2se/1.4.2/docs/

[17] EclEmma, http://www.eclemma.org/, odwiedzana w czerwcu 2008

[18] MutationTest Online http://www.mutationtest.net/, odwiedzana w czerwcu 2008

Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Jacek Pospychała",
    title = "{Testowanie mutacyjne wykorzystujące aspekty}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2008",
}
```