POLSKO-JAPOŃSKA WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Krystian Siuda
3572

Framework testowy dla języka Java

Praca magisterska
Napisana pod kierunkiem
dr inż. Mariusza Trzaski

Warszawa, luty, 2010

Streszczenie

Niniejsza praca traktuje o problemie realizacji automatycznych testów oprogramowania jako elementu procesu zapewnienia jakości przy jego produkcji. Praca zawiera omówienie rozwoju tej dziedziny na przestrzeni lat, najistotniejszych technik i rodzajów testów oraz zawiera przegląd dostępnych obecnie frameworków służących ich automatyzacji.

W drugiej części pracy opisane zostały wymagania oraz koncepcja rozwiązania, które miałoby uprościć tworzenie testów automatycznych oraz stanowić (wzbogaconą o dodatkową funkcjonalność) alternatywę dla dostępnych dziś frameworków. Przedstawiony został także przykład zastosowania prototypu takiego frameworku dla porównania z rozwiązaniami używanymi obecnie. Wymienione zostały również jego wady i propozycje dalszego rozwoju.

Podziękowania

Żonie, za wsparcie.

Spis treści

1. Wstęp

11 grudnia 1998 roku NASA wystrzeliła sondę „Mars Climate Orbiter". Miał to być kolejny, wart prawie pół miliarda dolarów, krok w kierunku zbadania sąsiadującej z Ziemią planety. Katastrofa rakiety „Ariane 5", mająca miejsce dwa lata wcześniej, wydawała się być już w tym czasie odległą przeszłością. Powody tamtejszej eksplozji były dobrze znane — rzutowanie 64-bitowej liczby zmiennoprzecinkowej na 16-bitową liczbę stałoprzecinkową. Było to kosztownym błędem, który nigdy nie powinien mieć miejsca. Specjaliści z NASA pracujący nad marsjańską sondą muszą być świadomi poważnych konsekwencji, pozornie niestety błahych błędów, które mogą pojawić się w oprogramowaniu sterującym lotem. Jednak 23 września 1999 roku kontakt z sondą będącą już bardzo blisko Marsa nie mógł być nawiązany. Sonda przepadła. W części systemu komputerowego stworzonego specjalnie dla tej misji użyto niewłaściwego układu miar. Dr Edward Weiler z NASA podsumował to wydarzenie słowami „czasami ludzie popełniają błędy" [1]. Te błędy, niestety nieraz bardzo kosztowne, zawsze nam towarzyszyły i będą nam towarzyszyć nadal. Podejmując odpowiednie działania możemy jednak minimalizować skutki ich występowania. Takim właśnie działaniem jest zarządzanie jakością przy produkcji oprogramowania oraz jego testowanie.

1.1. Cel pracy

Celem pracy jest przegląd metod testowania oprogramowania w kontekście procesu zapewnienia jakości systemów informatycznych oraz stworzenie koncepcji frameworku ułatwiającego testowanie aplikacji opartych o technologię Java, w czasie trwania całego ich cyklu życiowego. Z założenia framework ten, w odróżnieniu od większości istniejących rozwiązań tego typu, ma nie ograniczać się wyłącznie do testów jednostkowych, a dostarczać funkcjonalność wpierającą szerszy zakres testów. Ważnym jego założeniem jest możliwości rozbudowy i dostosowania do potrzeb jego użytkowników(programistów i testerów), a także możliwość jego integracji z bibliotekami logującymi, a w przyszłości także z innymi frameworkami testowymi.

1.2. Rozwiązanie przyjęte w pracy

Praca opiera się o język Java Standard Edition w wersji 6. Został on wybrany jako język obecnie bardzo popularny wśród programistów, często używany w dużych i średnich projektach informatycznych. Dodatkowym argumentem dla wyboru tego języka jest jego „przenośność" pomiędzy niemal wszystkie dostępne platformy sprzętowe oraz fakt, iż wirtualna maszyna, będącą jego częścią, zyskuje coraz większą popularność jako środowisko w którym mogą być uruchamiane aplikacje napisane również w innych językach jak Scala czy Groovy. Biorąc pod uwagę możliwość wywoływania funkcji tych języków z poziomu Javy prototyp będący rezultatem tej pracy zyskuje również zastosowanie przy testowaniu aplikacji zaimplementowanych i w tych językach.

Częścią prototypu jest graficzny interfejs użytkownika mający na celu ułatwienie użytkowania frameworku. Został on napisany przy użyciu biblioteki Swing będącej częścią Java SE. Do implementacji prototypu została też użyta dojrzała, ale i wciąż rozwijana biblioteka Apache XMLBeans, jako element odpowiedzialny za przetwarzanie plików XML opisujących scenariusze testowe.

Niniejsza praca ogranicza się wyłącznie do metod wykrywania błędów w oprogramowaniu w sposób empiryczny, w czasie jego działania. Pominięte zostały zagadnienia związane z metodami analizy kodu źródłowego aplikacji czy (statycznym) udowadnianiem jego poprawności.

Niniejsza rozprawa zawiera terminy i ich definicje zaczerpnięte ze słownika stworzonego przez International Software Testing Qualifications Board, w polskim przekładzie wykonanym przez Stowarzyszenie Jakości Systemów Informatyczncyh, dostępnym na licencji GNU Free Document Licence. Użycie tych terminów ma na celu ujednolicenie specjalistycznego słownictwa tej pracy ze słownictwem używanym w innych publikacjach tego rodzaju.

1.3. Rezultaty pracy

Rezultatem pracy jest koncepcja frameworku przypominającego funkcjonalnością podobne narzędzia tego typu, jednocześnie jednak wyróżniającego się ze względu na użycie opisanych w pracy wzorców projektowych oraz położenie nacisku na szybkość jego wykorzystania i wszechstronność. Jest on również wolny od ograniczenia jakim jest założenie

jego wykorzystania wyłącznie do jednego rodzaju testów — testów jednostkowych. Równocześnie dzięki jego modułowości, jest on doskonałą podstawą do modyfikacji i rozbudowy, czego nie mogą zaoferować obecne biblioteki jak JUnit czy TestNG.

1.4. Organizacja pracy

W rozdziale 1 przedstawiony został cel pracy, jej założenie, rozwiązania a także oczekiwany rezultat.

Rozdział 2 prezentuje podstawowe zagadnienia związane z dziedziną testowania oprogramowania — rodzaje testów, ich charakterystykę i zmiany w podejściu do tego zagadnienia na przestrzeni ostatnich kilkudziesięciu lat.

Rozdział 3 prezentuje dostępne obecnie rozwiązania, ich ograniczenia oraz porównuje je ze sobą.

Rozdział 4 jest prezentacją koncepcji narzędzia służącemu automatyzacji testów, wolnego od niektórych ograniczeń jakie mają dotychczasowe rozwiązania.

Rozdział 5 przedstawia szczegółową budowę prototypu tego narzędzia, omówienie jego wad i zalet oraz propozycję kierunków rozwoju w kolejnych wersjach.

2. Testowanie oprogramowania

Przy rosnącej lawinowo złożoności tworzonego oprogramowania niezbędnym elementem, dla zapewnienia jego bezawaryjności, towarzyszącym procesowi produkcji staje się zarządzanie jakością. Natomiast istotnym narzędziem w tym procesie jest testowanie, którego dotyczy ta rozprawa.

2.1. Testowanie na przestrzeni lat

Pierwotnie w procesie produkcji oprogramowania nie istniało pojęcie „testowania oprogramowania" w takim sensie, w jakim rozumiemy je obecnie. W początkowej fazie rozwoju zagadnienia zapewnienia jakości, „testowaniem" mogło być nazwane poszukiwanie błędów przy tworzeniu oprogramowania, poprzez użycie debuggera i staranne przeglądanie przez programistę kolejnych etapów wykonania kodu. Dopiero z czasem pojawiło się rozróżnienie pomiędzy debuggowaniem, mającym na celu znalezienie usterki w kodzie źródłowym aplikacji, a właściwym testowaniem, gdzie kładzie się nacisk na wykazanie zgodności oprogramowania z założeniami.

Ta dziedzina inżynierii oprogramowania przechodziła przez kolejne fazy [2]:

- zorientowania demonstracyjnego — gdzie wykazywano głównie poprawność działania oprogramowania,
- destrukcyjnego — gdzie testy miały na celu wykazanie prawidłowego obsługiwania wyjątków i błędów w czasie działania aplikacji,
- ewaluacyjnego — gdzie zintegrowano proces testowania nie tylko z fazą implementacji ale całym cyklem życia aplikacji,
- prewencyjnego — gdzie stawiano nacisk nie tylko na wykrywanie błędów ale i na unikanie ich występowania w przyszłości.

Każda z tych faz kończyła się na skutek ciągłego wzrost złożoności oprogramowania i konieczności doskonalenia oraz formalizacji procesu zapewnienia jakości. Proces ten stawał się coraz bardziej zintegrowany z metodykami zarządzania projektami. Poza tym zmienił się także znacząco sposób samej realizacji testów – upowszechniło się stosowanie testów

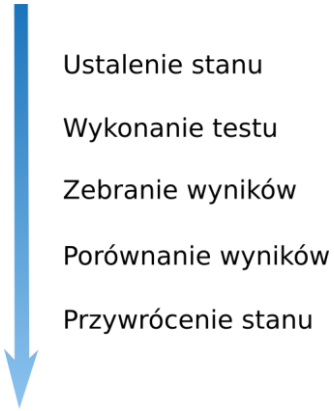
automatycznych, gdzie scenariusze testowe zostają zaimplementowane przez programistów w celu minimalizacji kosztów ich wielokrotnego wykonywania w przyszłości.

2.2. Testowanie współcześnie

Dziś testowanie oprogramowania jest już zagadnieniem znanym, przynajmniej w stopniu minimalnym, każdemu kto ma styczność z procesami produkcji, czy wdrożenia oprogramowania. Testy jednostkowe tworzone są już niemal przez każdego programistę na jego własny użytek. W większych realizacjach testy te stają się „dobrą praktyką" wymaganą przez organizację pracy. Projekty duże i złożone stawiają wiele wyzwań, którymi sprostano wprowadzając takie testy jak: integracyjne, systemowe czy pielęgnacyjne.

2.2.1. Testy jednostkowe

Zdobywającym wielkie uznanie wśród programistów sposobem testowanie oprogramowania jest tworzenie niewielkich programów sprawdzających poprawność funkcjonowania pojedynczych jednostek (w zależności od przyjętej praktyki mogą to być całe klasy lub pojedyncze funkcje) z których składa się oprogramowanie. Testy takie przeważnie składają się z pięciu kolejno występujących po sobie etapów:



Rysunek 1 Przebieg testu jednostkowego.

- ustalenie stanu — stworzenie wszystkich niezbędnych obiektów, ustalenie stanu początkowego dla testu (na przykład poprzez wprowadzenie niezbędnych modyfikacji w bazie danych);

- wykonanie — wykonanie funkcji jednostki testowanej, czyli oddanie sterowania do części testowanego systemu;
- zebranie wyników — pobranie z systemu wyniku wykonania testowanej funkcji (przy czym mogą tu być zbierane również metryki dla testów wymagań niefunkcjonalnych, takie jak czas wykonywania);
- porównywanie wyników otrzymanych po uruchomieniu testowanego kodu z oczekiwanymi rezultatami bazując na wymaganiach systemu (funkcjonalnych i niefunkcjonalnych);
- przywrócenie stanu — cofnięcie wszelkich zmian, które zostały wprowadzone w czasie działania testu (przykładem może to być wykonanie operacji „rollback" na bazie danych modyfikowanej w czasie działania funkcji systemu), tak aby był on identyczny ze stanem pierwotnym.

Zagadnieniem związanym z testami jednostkowymi jest ocena pokrycia kodu źródłowego oprogramowania. Pokrycie jest miarą mówiącą o tym w jakim stopniu testy jednostkowe w czasie swojego działania wymusiły wykonanie całego kodu testowanego oprogramowania. Przykładowo przetestowanie wszystkich funkcji danej klasy dla wszystkich możliwych jej stanów oznaczałoby pełne pokrycie testami tej funkcji.

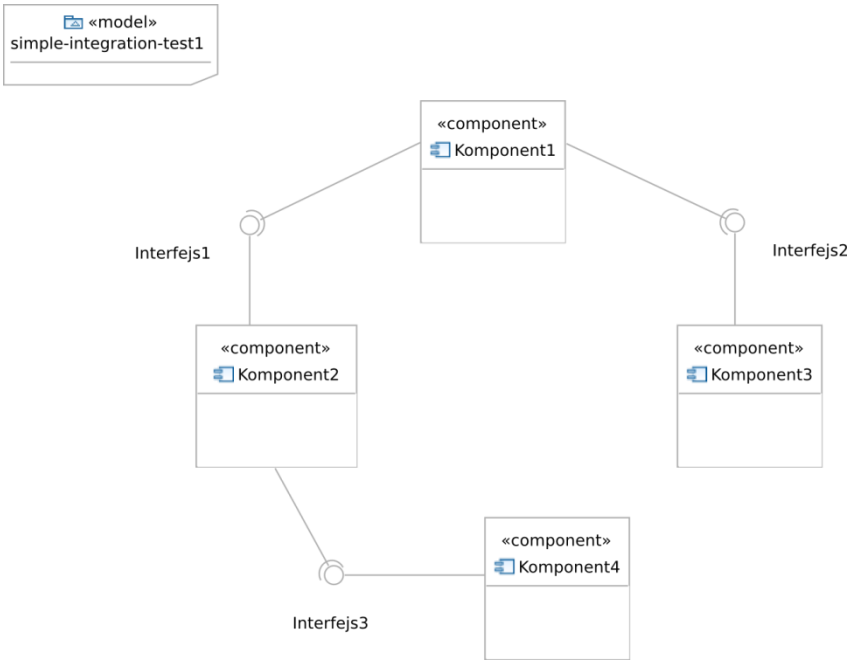
Testy jednostkowe stały się podwaliną pod metodyki *Test Driven Development*, które rozwijają się wraz z popularyzacją *Extreme Programming*. Polegają one na tworzeniu testów jednostkowych dla danego modułu przed jego implementacją, w ten sposób, aby testy te opisywały zakładaną funkcjonalność (i zarazem weryfikowały poprawność implementacji). Testy te tworzone są wówczas jeszcze przed rozpoczęciem prac programistycznych i dopiero wraz z postępem prac ich wyniki testów stają się pozytywne.

2.2.2. Testy integracyjne

Testy integracyjne to metoda koncentrująca się na współdziałaniu pojedynczych lub wybranej grupy jednostek systemu ze sobą. Przy tego rodzaju testach przyjmuje się iż każda z nich z osobna działa prawidłowo (co dowieść ma pomyślne wykonanie wcześniej testów jednostkowych). Wymagane jest natomiast sprawdzenie czy prawidłowo współpracują one z pozostałymi częściami systemu.

Przykładowymi błędami, wykrywanymi przez te testy są:

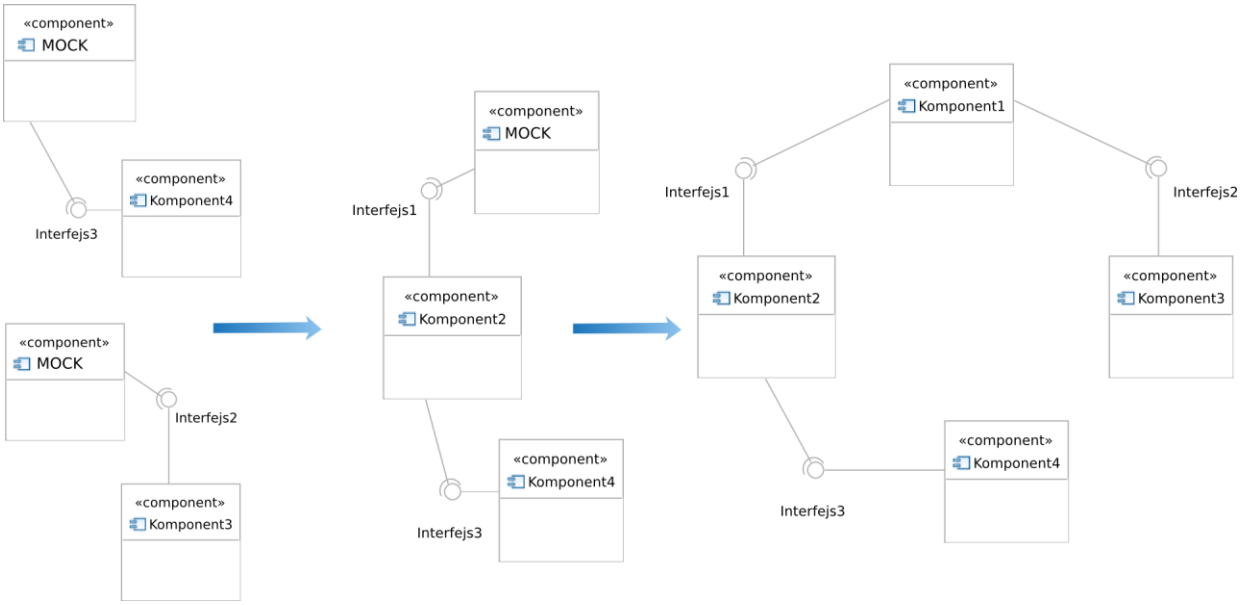
- brak komunikacji pomiędzy modułami,
- niewłaściwy rodzaj informacji przesyłanych między modułami,
- niezgodność semantyczna — przesyłane dane są w różny sposób interpretowane przez testowane moduły.



Rysunek 2 Diagram komponentów dla przykładowego testu integracyjnego.

Test integracyjny przebiega podobnie do testu jednostkowego, z tą różnicą, że w czasie inicjalizacji tworzone są powiązania pomiędzy wszystkimi komponentami mającymi brać udział w teście. Rozróżnia się przy tym trzy alternatywne podejścia do takiego testowania.

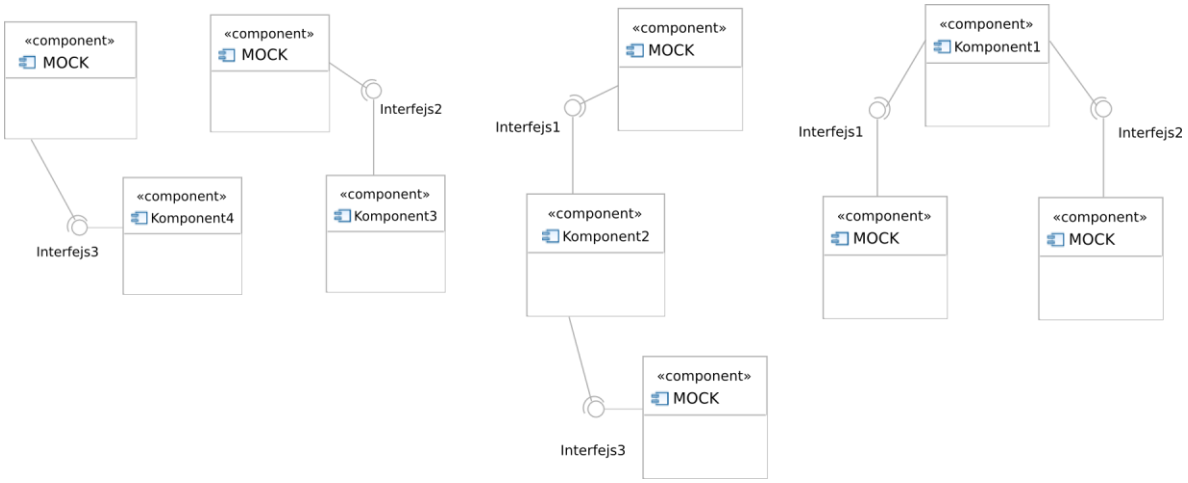
Często zależności pomiędzy komponentami wymuszają tworzenie tak zwanych obiektów typu *mock* zaślepiających dany interfejs na potrzeby uruchamianych testów. Symulują one działanie obiektów będących rzeczywistą implementując danego interfejsu, tak by testowane komponenty mogły działać jak w środowisku docelowym, lecz bez angażowania wszystkich składowych systemu. Rysunek 2 przedstawia środowisko dla testu integracyjnego gdzie wykorzystywane są trzy rzeczywiste moduły systemu oraz jeden obiekt „mock" zaślepiający interfejs „B" jednego z tych modułów.



Rysunek 3 Kolejne etapy wstępującego testu integracji.

Pierwszym podejściem do tego rodzaju testów jest łączenie ze sobą modułów od najmniej złożonych (testowanych w pierwszej kolejności) tworząc coraz to bardziej skomplikowane struktury (Rysunek 3). Testowane są one później w kolejnych iteracjach (co nazywane jest testowaniem wstępującym), aż do momentu kiedy do testów zostaną zaangażowane wszystkie moduły.

Przeciwieństwem testów wstępujących jest testowanie począwszy od modułu będącego zależnym od największej ilości innych modułów (będącego na szczycie hierarchii zależności, czyli widocznego na przykładzie Rysunek 2 jako jednostka „1”). Tu również stosowane jest podejście iteracyjne: testy rozpoczynają się od głównego modułu z użyciem obiektów „mock” dla wszystkich jego interfejsów, następnie pojedynczo te obiekty zastępowane są ich rzeczywistą implementacją, aż do momentu, gdy w testach nie ma już potrzeby zaślepiania interfejsów.



Rysunek 4 Pojedyncze testy integracyjne.

Alternatywnie moduły mogą być też testowane pojedynczo, za każdym razem zaślepiając wszystkie ich interfejsy od których są zależne, jak i te które są przez nie udostępniane (Rysunek 4).

2.2.3. Testy dymne, potwierdzające i wstępne

Specyficznym rodzajem testów są testy dymne, które mają na celu pokrycie najistotniejszych funkcjonalności systemu przy możliwie małej złożoności scenariuszy i krótkim czasie ich wykonania. Testy te są najczęściej podzbiorem stworzonych już przypadków testowych, uznanych za najbardziej istotne. Codzienne ich wykonywanie jest uznawane za „dobrą praktykę", podobnie jak tworzenie testów jednostkowych przez programistów.

Ściśle związane z testami dymnymi są testy potwierdzające. Ich celem jest zreprodukowanie awarii do której doszło w przeszłości, a przyczyny której (błędy w oprogramowaniu) zostały już usunięte przez programistów. Test taki potwierdza usunięcie znalezionego defektu i możliwość przystąpienia do dalszej pracy testera nad funkcjonalnością, która została w ten sposób przywrócona.

Pewnym rodzajem testów dymnych są testy wstępne. Warunkują one wykonanie właściwych testów, gdyż defekty przez nie wykryte świadczą o braku możliwości wykonania

testów właściwych (takim testem mogłoby być sprawdzenie, czy system uruchamia się poprawnie w danym środowisku).

2.2.4. Testy systemowe

Weryfikowanym przedmiotem przez testy systemowe jest system informatyczny jako całość. W tej metodzie abstrahuje się całkowicie od jego wewnętrznej budowy, a sprawdza w jakim stopniu spełnia on oczekiwania odbiorcy systemu. Testy te głównie dotyczą:

- wydajności, do czego zalicza się:
  - testowanie obciążeniowe, gdzie sprawdzana jest poprawność działania systemu przy danym (najczęściej bardzo wysokim lub zmieniającym się w czasie) obciążeniu;
  - testowanie przeciążające, badające zachowanie systemu po przekroczeniu maksymalnej jego przepustowości — tak by zbadać powstawanie możliwych wyjątków w takiej sytuacji;
- zabezpieczeń — weryfikacja przyjętych dla systemu założeń dotyczących jego bezpieczeństwa;
- międzyoperacyjności — sprawdzenie spełnienia wymogów dotyczących współpracy pomiędzy systemami (np. możliwości przenoszenia systemu między platformami sprzętowymi);
- tolerancji wyjątków — badanie zachowania po powstaniu wyjątków (po których system nadal powinien pracować stabilnie);
- skalowalności — sprawdzenie, czy istnieje zakładana możliwość rozbudowy systemu dla uzyskania większej jego przepustowości;
- niezawodności — weryfikacja stabilności pracy oprogramowania w długim czasie jego funkcjonowania;
- odtwarzalności — potwierdzenie możliwość odtworzenia stanu systemu po poważnym i nieprzewidzianym jego zatrzymaniu (np. w związku z awarią macierzy dyskowej).

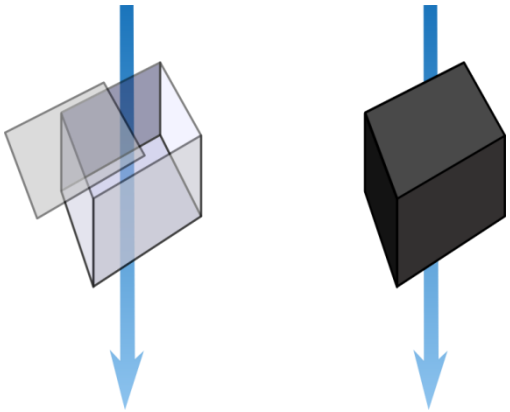
2.2.5. Testy akceptacyjne

Testy kompletnego systemu w jego docelowym środowisku nazywane są testami akceptacyjnymi. Mają one na celu udowodnienie spełnienia wszystkich założeń (wymagań funkcjonalnych i nie funkcjonalnych) stworzonego rozwiązania oraz możliwość jego produkcyjnego wykorzystania. Testy te przeprowadzane są tuż przed wdrożeniem, na wyizolowanych środowiskach testowych odpowiadających docelowym, lub zaraz po nim, na środowisku produkcyjnym. Wyniki testów przeważnie składają się z prostej kwalifikacji: spełnienia wymagań lub jego braku. Podobnie jak w testach systemowych, nie sprawdza się tu wewnętrznej implementacji systemu, ale bada spełnienie kryteriów biznesowych jakie są oczekiwane przez odbiorcę oprogramowania. Pozytywny wynik tego testu jest wystarczającym warunkiem dla rozpoczęcia eksploatacji oprogramowania.

Ten rodzaj testów jest wyjątkowo istotny, gdy do realizacji oprogramowania używane są metodyki zwinne (zwłaszcza *Extreme Programming*). Jest on tam ostatecznym etapem potwierdzenia spełnienia wszystkich wymogów systemu i jeśli zakończy się pomyślnie, zwieńcza implementacje systemu. Takie testy są wówczas realizowane tuż przed podpisaniem protokołu odbioru gotowego systemu. Ich negatywny wynik prowadzi do wpisania brakującej funkcjonalności na *listę usterek* (z ang. *punch list*.) i warunkowy odbiór oprogramowania lub jego odrzucenie.

2.2.6. Testy biało i czarno skrzynkowe

Testy można podzielić na dwie główne grupy [4]:



Rysunek 5 Test biało skrzynkowy i czarno skrzynkowy.

- biało-skrzynkowe — gdzie do tworzenia przypadków testowych wykorzystywana jest znajomość wewnętrznej struktury testowanego modułu; ten rodzaj testów to większość testów jednostkowych i integracyjnych;
- czarno-skrzynkowe — gdzie przy tworzeniu scenariuszy testowych używana jest wyłącznie wiedza dotycząca wymagań funkcjonalnych i niefunkcjonalnych systemu; jest to więc najczęściej wykorzystywany rodzaj testów przy testach systemowych i akceptacyjnych, gdzie kładzie się nacisk na spełnienie biznesowych założeń tworzonego systemu abstrahując od jego implementacji, technologii etc.

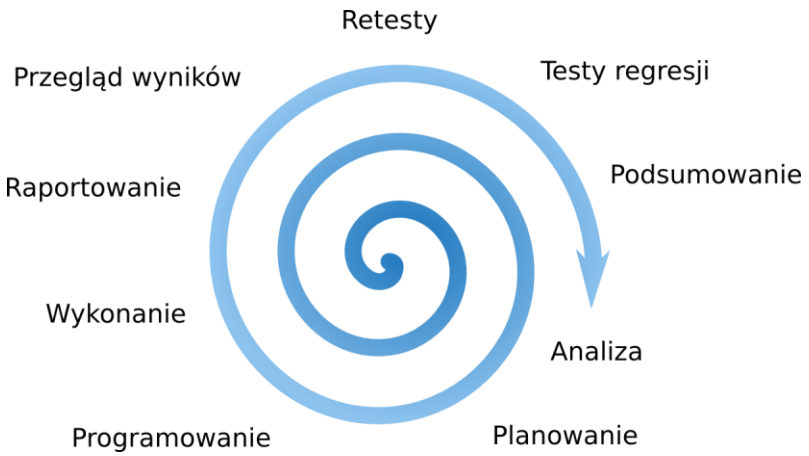
2.2.7. Testy w fazie utrzymania

Z testowaniem oprogramowania mamy do czynienia również w czasie utrzymania wdrożonego już oprogramowania. Systemy po wdrożeniu często poddawane są dalszym modyfikacjom, czy naprawom, odkrytych w czasie ich użytkowania usterek.

Każdorazowo po wprowadzeniu zmian w systemie powinny wykonywane być testy nowej funkcjonalności, jeśli taka jest wprowadzana, lub retesty dla funkcjonalności, która została zmodyfikowana, gdy poprawiono błędy z nią związane.

Biorąc jednak pod uwagę brak możliwości pełnego spojrzenia na zależności w budowie złożonego systemu, prowadzi się również testy regresji. Jest to ponowne przetestowanie uprzednio już testowanego oprogramowania, w którym wprowadzono zmiany, w celu upewnienia się, że w wyniku zmian nie powstały nowe, lub nie ujawniły się wcześniej nie wykryte defekty w niezmienionej części oprogramowania. Testy takie prowadzi się również po zmianie środowiska w którym działa wdrożony system.

2.2.8. Cykl życia testów



Rysunek 6 Cykl życia procesu testowania.

Cykl życia procesu testowania można podzielić na następujące fazy (Rysunek 6):

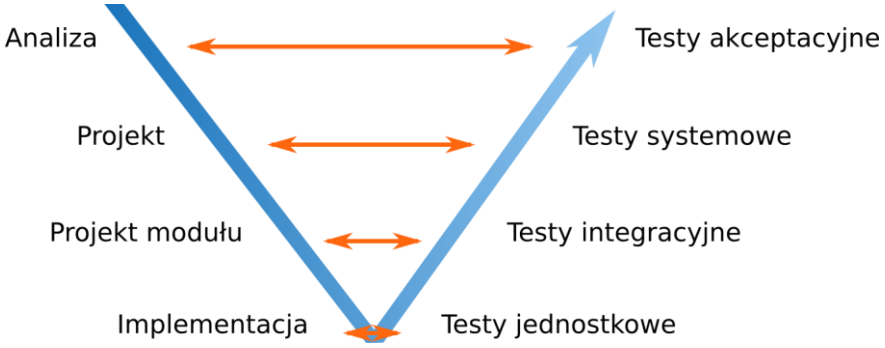
- analiza — gdzie zbierane są wymagania odnośnie procesu testowania, formułowany jest cel tego procesu i uzgadniane są dalsze jego etapy;
- planowanie — w czasie którego uzgadniana jest metodologia używana w czasie testów, określany jest zakres testów, wyliczane są zasoby niezbędne do ich przeprowadzania, estymowany jest czas ich prowadzenia, tworzone są scenariusze testów;
- programowanie — etap gdzie testerzy automatyzują (o ile jest to możliwe) testy tworząc oprogramowanie które będzie je wykonywać na podstawie scenariuszy;
- wykonanie — uruchamianie automatów testujących lub wykonywanie ręcznych scenariuszy testów;
- raportowanie — zbieranie wyników wykonania testów w formie raportów zgodnych z przyjętymi wcześniej metodykami;
- przegląd wyników — formalne zebranie zespołu prowadzącego projekt;
- retesty — powtórne wykonywanie testów dla składowych systemu, które zostały zmienione po wykazaniu w nich defektów w fazie wykonania;

- testy regresji — wykonanie testów mających na celu znalezienie nowopowstałych wad po wprowadzonych do systemu zmianach;
- podsumowanie —formalne przedstawienie raportu z całego procesu.

Cykl ten (jak i każda z jego faz) może być wielokrotnie powtarzany, prowadząc do ciągłego wzrostu jakości oprogramowania, aż do osiągnięcia satysfakcjonujących wyników.

2.2.9. Model V

Obecnie najczęściej zapewnienie jakości wytwarzanego oprogramowania spostrzegane jest jako czynność równoległa do właściwej jego produkcji, przy czym silnie z nią związana i ciągle wzajemnie oddziaływująca [3].



Rysunek 7 Model V.

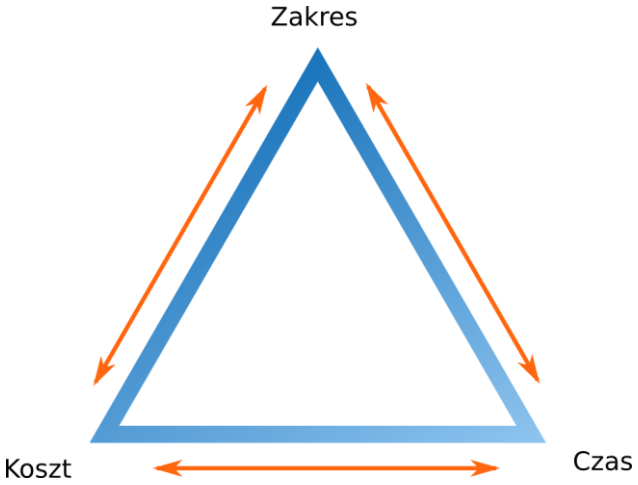
Popularnym sposobem przedstawiania tego oddziaływania jest diagram w kształcie litery V (Rysunek 7). Na jednej z krawędzi tego diagramu wymienione są czynności pojawiające się w modelu kaskadowym. Tworzenie oprogramowania w tym modelu rozpoczyna się od analizy, która poprzez stopniowe uszczegółowienie przeradza się w dokładny projekt, aby ostatecznie projekt ten mógł zostać zaimplementowany. Każdy stopień uszczegółowienia niesie za sobą tworzenie odpowiednich założeń i scenariuszy testów które będą później wykonywane w kolejności odwrotnej (tj. od najbardziej szczegółowego do najbardziej ogólnego) i tworzy to drugą krawędź diagramu.

Implementacja znajduje swoje odzwierciedlenie w testach jednostkowych, gdzie weryfikowana jest jej poprawność. Projekt systemu i modułu weryfikowane są przez odpowiednie testy systemowe i integracyjne. Natomiast sprawdzenie spełnienia przez system fundamentalnych wymagań zebranych w czasie analizy, realizowane jest poprzez testy akceptacyjne.

Model taki wymusza współdziałanie pomiędzy testowaniem, a tworzeniem oprogramowania, od zarysowania koncepcji, aż do odbioru gotowego produktu. Jest przy tym też zachowana duża dowolność co do granulacji kolejnych etapów produkcji oprogramowania i łatwo komponuje się on z innymi modelami (przykładem może być model spiralny gdzie każdy obrót spirali oznaczałby przejście wszystkich stopni modelu V).

2.3. Wyzwania związane z testowaniem

Z pewnością dziedzina *testowania oprogramowania*, jak i szerzej rozumiane pojęcie *zapewnienia jakości systemów informatycznych*, będą intensywnie rozwijane w nadchodzących latach. Jako powód tego może być brana pod uwagę, między innymi, rosnąca ilość złożonych i wciąż rozwijanych systemów informatycznych, ale i potrzeba migracji starych rozwiązań na ich nowsze odpowiedniki. Pożądana jest minimalizacji kosztów ich wdrażania i utrzymywania, przy jednoczesnym podnoszeniu jakości. Sytuacja ekonomiczna, w tym światowe kryzysy gospodarcze, przyczyniają się do bardziej ostrożnego podejścia klientów — odbiorców końcowych systemów — do inwestycji w nowe, z ich punktu widzenia mniej pewne oprogramowanie. Wszystko to sprawia, że pojawia się potrzeba poszukiwania coraz skuteczniejszych metod zarządzania jakością, a jednocześnie minimalizacji kosztów związanych z podejmowaniem takich działań.



Rysunek 8 Trójkąt zarządzania projektem.

Trójkąt ograniczeń, nazywany też trójkątem zarządzania projektem (Rysunek 8) obrazuje relacje pomiędzy: czasem, kosztem oraz zakresem w realizowanych przedsięwzięciach. Jest to znana w dziedzinie zarządzania zależność pomiędzy tymi trzema czynnikami, sprawiająca iż chcąc kłaść większy nacisk na skrócenie czasu realizacji, trzeba liczyć się ze wzrostem kosztów projektu, bądź zmniejszeniem jego zakresu. Analogicznie: kierując się ku zmniejszeniu kosztów, niezbędne jest zmniejszenie zakresu, bądź wydłużenie czasu. Również zwiększając zakres zwiększą się koszty, jak i wydłuża czas realizacji. [5]

Zależność ta tyczy się również testowania, będącego częścią projektu. Dbając o najwyższą jakość końcowego produktu — systemu informatycznego — liczyć się trzeba z drastycznie rosnącymi kosztami lub czasem jego produkcji. Z kolei uzyskiwanie oszczędności, poprzez redukcję zakresu testów, zwiększając ryzyko pojawienia się błędów w oprogramowaniu, co może mieć daleko idące negatywne konsekwencje. Widoczny jest więc związek między zarządzaniem ryzykiem, a zapewnieniem jakości. Wszystkie te zależności mają olbrzymi wpływ na proces zapewnienia jakości, jak i samą czynność testowania, gdzie czynnik ekonomiczny odgrywa istotną rolę.

Mając ograniczony budżet oraz określony czas na wdrożenie systemu informatycznego, nie ma możliwości kompletnego przetestowania aplikacji, tak aby wykluczyć wszystkie możliwe w niej błędy [6].

Obecnie poszukuje się metod integracji procesów zapewnienia jakości ze wszystkimi pozostałymi, tak by defekty mogły być wykrywane jak najwcześniej jak to jest możliwe.

W najlepszym przypadku testowanie rozpoczynałoby się jeszcze przed implementacją, w czasie trwania fazy analizy oraz projektowania, przybierając formę audytów (wewnętrznych oraz zewnętrznych) i list kontrolnych.

W procesie testowania wskazane jest wyraźne rozróżnienie pomiędzy osobami pełniącymi w zespole role testerów i programistów (wyjątkiem jest tworzenie testów jednostkowych, które są też pomocą dla samych programistów). Ma to związek ze specyfiką pracy [7], w czasie pełnienia w nim funkcji testera, a więc osoby wykrywającej błędy popełnione przez innych członków zespołu. Stwarza to też wyzwanie przed osobami zarządzającymi tak podzielonym zespołem. Nieodzowna jest bowiem jak najbliższa współpraca pomiędzy programistami i testerami przez cały czas trwania projektu.

Szybki rozwój i popularyzacja zwinnych metod wytwarzania oprogramowania prowadzi też do konieczności adaptacji procesu testowania do tychże metod [8] które, najprawdopodobniej, w przyszłych latach wciąż będą zyskiwać na popularności oraz będą intensywnie rozwijane. W przypadku wyboru takich metodyk konieczna jest możliwość szybkiego zmieniania procedury prowadzenia testów. Niezbędne jest możliwie szybkie wykonywanie testów, dokonywanie w nich poprawek oraz informowanie osób kierujących pracami o ich wynikach poprzez raportowanie.

Pochodną uwarunkowań ekonomicznych jest też chęć zwiększania możliwości ponownego użycia przygotowanych już scenariuszy testowych, czy stworzonego kodu, na potrzeby testów automatycznych. Dzięki temu, bez ponoszenia dużych kosztów, czy angażowania dodatkowych zasobów, możliwe jest poszerzanie zakresu testów jak i szybka zmiana mechaniki ich działania. Przykładem może być wykorzystywanie wcześniej zautomatyzowanych przypadków testowych po wdrożeniu aplikacji, w czasie jej utrzymywania, do retestów i testów regresji.

Również sama automatyzacja testów jest zagadnieniem podlegającym ciągłemu rozwojowi. Ma to na celu zmniejszanie nakładu pracy niezbędnego do tworzenia takich testów oraz zwiększanie możliwości późniejszego, ponownego wykorzystania testów automatycznych.

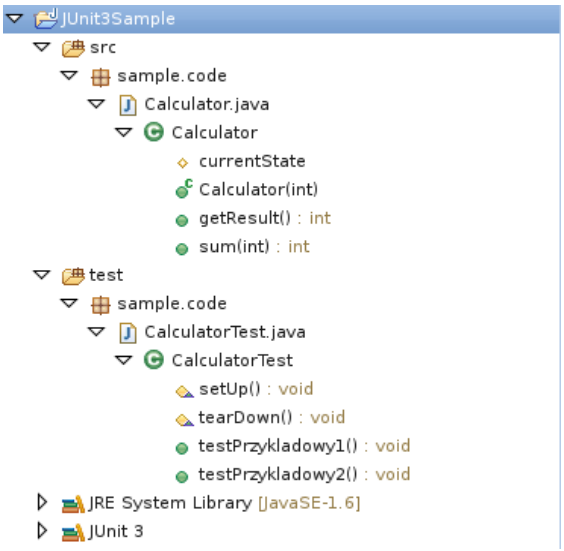
3. Przegląd dostępnych rozwiązań

Obecnie najpopularniejszymi narzędziami wykorzystywanymi do automatyzacji testów w języku Java są biblioteki JUnit i TestNG, omówione poniżej. Cechują się one gotowymi, wygodnymi szablonami, które używane są do implementacji zestawów testów. Jednak rosnące oczekiwania wobec rozwiązań tego typu wymuszają ciągłe poszukiwanie ich nowszych, wzbogaconych o nowe możliwości alternatyw.

3.1. JUnit 3

JUnit dzięki swojej olbrzymiej popularności odgrywa obecnie rolę standardu. Dostarczany jest, całkowicie zintegrowany, wraz z takimi IDE (Integrated Development Enviroment — zintegrowane środowisko programistyczne) jak: Eclipse, NetBeans, czy IntelliJ.

Framework ten został stworzony na potrzeby testów jednostkowych pisanych przez programistów na własny użytek. Wyróżnia go prostota użycia i budowy wewnętrznej. Jest jednym z pierwszych narzędzi tego typu, które zostało ogólnie udostępnione (na zasadach licencji Common Public License, dającej możliwości nieodpłatnego użytkowania) i stało się podstawą do tworzenia bardziej skomplikowanych bibliotek o większej funkcjonalności.



Rysunek 9 Przykładowy projekt Eclipse zawierający testy JUnit 3.

Zasada tworzenia testów w tym frameworku polega na pisaniu specjalnych klas zawierających scenariusze testowe, badających poprawność implementacji dla właściwych klas tworzonej aplikacji. Scenariusze te, są więc z punktu widzenia programisty, użytkownika danego IDE, częścią tego samego projektu (Rysunek 9). Są one jednakowe ze względu na język użyty do ich implementacji, aczkolwiek oddzielone od właściwego kodu z powodu innego przeznaczenia.

Klasy te, będące opisem scenariuszy testowych, zawierają wyłącznie kod odpowiedzialny za wykonanie testów, zebranie ich wyników oraz ich porównanie z wartościami oczekiwanymi. Metody te są uruchamiane w czasie testów przez framework. Walidacja wyników testów dokonywana jest przy pomocy funkcji udostępnianych przez bibliotekę JUnit.

```
 1 package sample.code;
 2
 3 public class Calculator {
 4
 5         protected int currentState;
 6
 7         public Calculator(int initialState) {
 8                 currentState = initialState;
 9         }
10
11         public int sum(int operand) {
12                 return currentState += operand;
13         }
14
15         public int getResult() {
16                 return currentState;
17         }
18
19 }
```

Listing 1 Przykładowa klasa podlegająca testom.

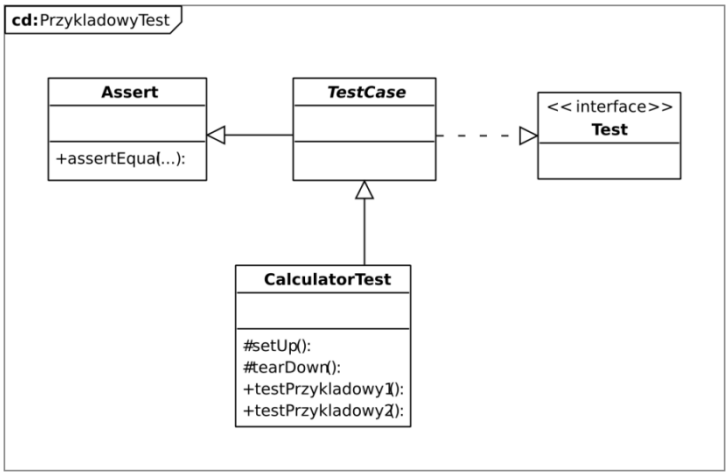
```
 1 package sample.code;
 2
 3 import junit.framework.TestCase;
 4
 5 public class CalculatorTest extends TestCase {
 6
 7         protected void setUp() throws Exception {
 8                 System.out.println("SetUpMethod");
 9                 super.setUp();
10         }
11
12         protected void tearDown() throws Exception {
```

```
13                  System.out.println("TearDownMethod");
14                  super.tearDown();
15          }
16
17          public void testPrzykladowy1() {
18                  System.out.println("Test1");
19                  Calculator calculator = new Calculator(0);
20                  assertEquals(10, calculator.sum(10));
21          }
22
23          public void testPrzykladowy2() {
24                  System.out.println("Test2");
25                  Calculator calculator = new Calculator(0);
26                  assertEquals(-10, calculator.sum(-10));
27          }
28
29  }
```

Listing 2 Przykładowy test dla klasy z Listing 1.

Widoczna powyżej klasa `Calculator` (Listing 1) może być w ten sposób przetestowana przy użyciu scenariuszy testowych klasy `CalculatorTest` (Listing 2). Zaimplementowane są tam dwa przypadki testowe: `testPrzykladowy1` i `testPrzykladowy2`, sprawdzające poprawność działania kodu klasy `Calculator` dla dodawania liczby dodatniej i ujemnej do zera. Wszystkie metody będące testami posiadają nazwy rozpoczynające się od ciągu `test` co wyróżnia je jako przypadki testowe.

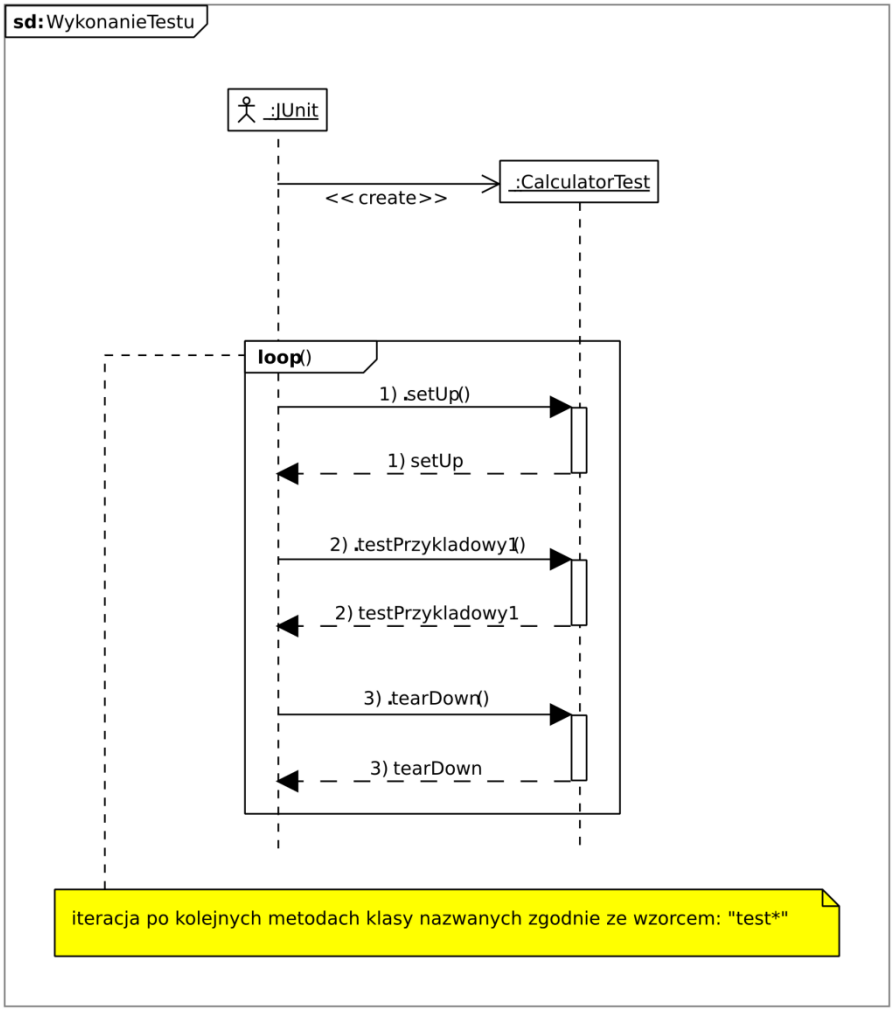
Implementacja funkcji `setUp` i `tearDown` jest opcjonalna, daje ona możliwość inicjacji stanu testowanej jednostki przed wykonaniem testów (co jeśli zakończyłoby się wyjątkiem wstrzymałoby właściwe wykonanie testu) i przywrócenie stanu po jego wykonaniu (niezależnie od jego wyniku).



Rysunek 10 Hierarcha klas dla przykładowego testu w JUnit 3.

Klasa `CalculatorTest` (Rysunek 10) dziedziczy po (będących częścią frameworku):

- Abstrakcyjnej klasie `TestCase` — fizycznie wykonującej kolejne przypadki testowe klasy po niej dziedziczącej, do których odnalezienia wykorzystuje mechanizm refleksji języka Java;
- Klasie `Assert` — dostarczającej metod wykorzystywanych do porównywania wyników testu, dzięki czemu metody te są bezpośrednio widoczne w ciele przypadków testowych;
- Interfejsie `Test` — będącym znacznikiem możliwości uruchomienia dla Framework JUnit.

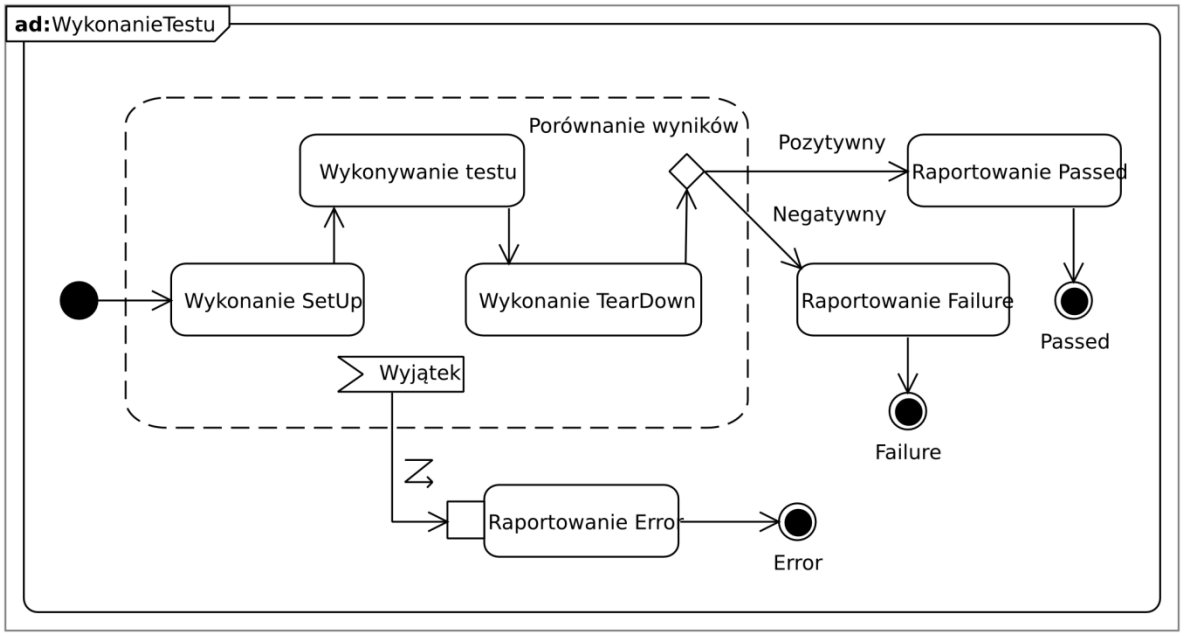


Rysunek 11 Wywołanie przykładowych testów w JUnit 3.

Wykonanie testów przez framework (Rysunek 11), odbywa się kolejno przez:

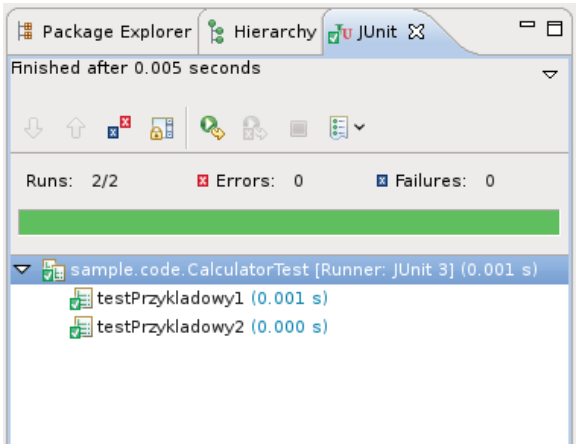
- utworzenie obiektu klasy implementującej testy;
- iteracje po przypadkach testowych z tej klasy;
  - wykonuje metodę setUp (niepowodzenie w tym kroku przerywa wykonanie i przechodzi do kolejnego przypadku testowego);
  - uruchamia właściwy kod przypadku użycia, który kończy się powrotem lub rzuceniem wyjątku (w przypadku nieprawidłowości w czasie wykonania testu);
  - wykonuje metodę tearDown;

- przechodzi do kolejnego przypadku testowego;



Rysunek 12 Możliwy przebieg testu w JUnit 3.

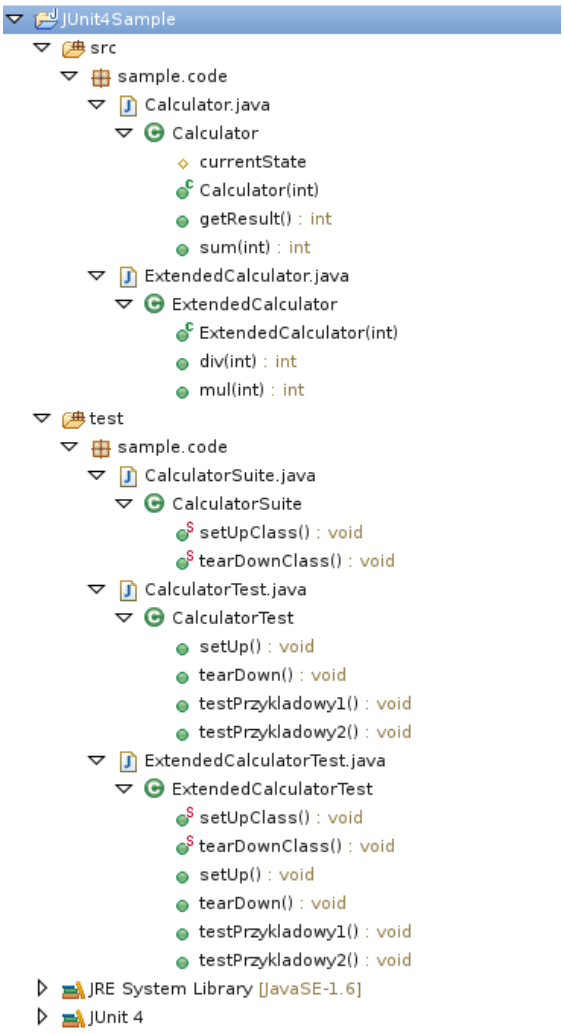
Każdorazowo po uruchomieniu tej sekwencji dla każdego z przypadków testowych JUnit3 kwalifikuje wynik jej wykonania (Rysunek 12) jako *Error*, *Failure* lub *Passed*. Ostatecznie dla wszystkich klas zawierających testy wyświetlana jest informacja o wynikach (Rysunek 13).



Rysunek 13 Wyniki wykonania przykładowych testów w JUnit 3.

3.2. JUnit 4

JUnit 4 jest frameworkiem udostępniającym niezbędną funkcjonalność dla szybkiego tworzenia testów jednostkowych. Wraz z pojawieniem się języka Java w wersji 5, zawierającego mechanizm adnotacji, pojawiła się możliwość bardziej przejrzystego sposobu deklarowania takich testów. JUnit w wersji 4 został znacząco zmodyfikowany, aby uwzględnić te możliwości.



Rysunek 14 Przykładowy projekt Eclipse zawierający testy JUnit 4.

```
1 package sample.code;
2
3 public class ExtendedCalculator extends Calculator {
4
5         public ExtendedCalculator(int initialState) {
```

```
 6                 super(initialState);
 7         }
 8
 9         public int mul(int operand) {
10                 return currentState *= operand;
11         }
12
13         public int div(int operand) {
14                 return currentState /= operand;
15         }
16
17 }
```

Listing 3 Przykładowa, dodatkowa, klasa podlegająca testom.

Rysunek 14 prezentuje widok projektu Eclipse zawierający wykorzystanie w przykładach biblioteki JUnit 4. Zawiera on testy, których implementacja scenariuszy przedstawiona jest na Listing 4 i Listing 5, dla odpowiadającym im klas z Listing 1 i Listing 3.

Adnotacje te umożliwiają odcięcie się od wcześniej wymuszonej w JUnit 3 konwencji nazewnictwa metod pełniących określone funkcje. Klasy zawierające testy nie muszą również dziedziczyć po klasie `TestCase`.

W JUnit 4 najczęściej wykorzystywanymi adnotacjami są:

- `@Test` — oznaczająca metodę w klasie która implementuje pojedynczy przypadek testowy. Adnotacja ta, jako parametr, może przyjmować klasę wyjątku (dziedziczącą po Throwable) który powinien wystąpić w czasie jego wykonania;
- `@Before` i `@After` — oznacza metodę będącą odpowiednikiem `setUp` i `tearDown` w JUnit 3, jednak metod tych może być wiele w ciele pojedynczej klasy;
- `@BeforeClass` i `@AfterClass` — metody przygotowujące (i przywracające pierwotny stan) środowisko testowe do testów, podobnie jak `@Before` i `@After`, jednak wykonywane tylko raz przed i po uruchomieniu wszystkich testów (a nie wielokrotnie dla każdego testu) z bieżąco wykonywanej klasy.

```java
package sample.code;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {

	public CalculatorTest() {
		System.out.println("CalculatorTest()");
	}

	@Before
	public void setUp() throws Exception {
		System.out.println("SetUpMethod");
	}

	@After
	public void tearDown() throws Exception {
		System.out.println("TearDownMethod");
	}

	@Test
	public void testPrzykladowy1() {
		System.out.println("Test1");
		Calculator calculator = new Calculator(0);
		assertEquals(10, calculator.sum(10));
	}

	@Test
	public void testPrzykladowy2() {
		System.out.println("Test2");
		Calculator calculator = new Calculator(0);
		assertEquals(-10, calculator.sum(-10));
	}

}
```

Listing 4 Implementacja w JUnit 4 testów z Listing 2.

```java
package sample.code;

import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class ExtendedCalculatorTest {

	public ExtendedCalculatorTest() {
		System.out.println("ExtendedCalculatorTest()");
```

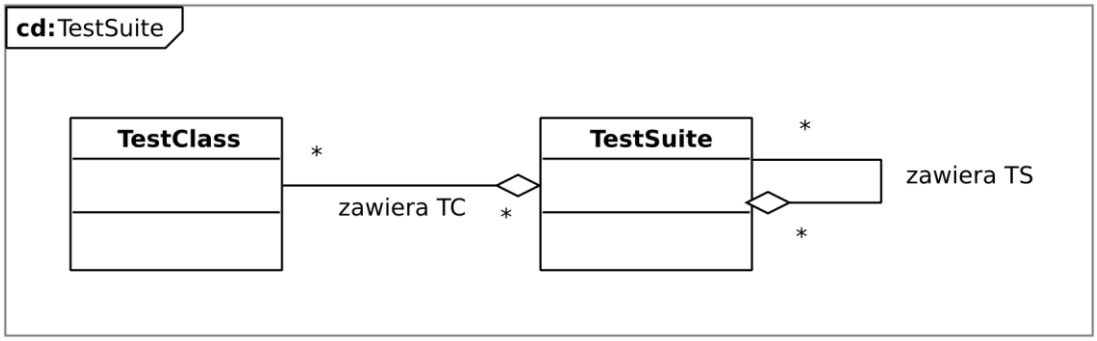
```
15          }
16
17          @Before
18          public void setUp() throws Exception {
19                  System.out.println("SetUpMethod - dla dodatkowych
    testów");
20          }
21
22          @After
23          public void tearDown() throws Exception {
24                  System.out.println("TearDownMethod - dla dodatkowych
    testów");
25          }
26
27          @BeforeClass
28          public static void setUpClass() throws Exception {
29                  System.out.println("SetUpMethod - dla wszystkich
    dodatkowych testów");
30          }
31
32          @AfterClass
33          public static void tearDownClass() throws Exception {
34                  System.out.println("TearDownMethod - dla wszystkich
    dodatkowych testów");
35          }
36
37          @Test
38          public void testPrzykladowy1() {
39                  System.out.println("DodatkowyTest1");
40                  ExtendedCalculator calculator = new
    ExtendedCalculator(10);
41                  assertEquals(100, calculator.mul(10));
42          }
43
44          @Test(expected=ArithmeticException.class)
45          public void testPrzykladowy2() {
46                  System.out.println("DodatkowyTest2");
47                  ExtendedCalculator calculator = new
    ExtendedCalculator(0);
48                  calculator.div(0);
49          }
50
51  }
```

Listing 5 Testy dla klasy z Listing 3.

Dzięki większej dowolności tworzenia metod inicjujących stan testów (oznaczonych przez `@Before` i `@After`) możliwe jest zadbanie o bardziej przejrzysty kod testów. Dzielenie fazy inicjacji na wiele metod realizujących to zadanie zwiększa czytelność, a także zmniejsza czas poświęcony analizie problemu jeśli ta inicjacja nie powiedzie się (błędy raportowane z każdej z faz są łatwiejsze do rozróżnienia).

Przy pomocy mechanizmu adnotacji możliwe jest też łatwe definiowanie `TestSuite` (zestawu testów), które grupują ze sobą wiele `TestCase` i innych `TestSuite` tworząc hierarchiczną strukturę (Rysunek 15). Grupowanie takie zależy całkowicie od intencji testera i może okazać się bardzo przydatne przy zdawaniu raportów z testów które również taki podział mogą odzwierciedlać (np. podział według wymagań biznesowych).

Taka struktura powstaje poprzez definicje `TestSuite` w klasie `CalculatorSuite` (Listing 6; Rysunek 16). Klasa ta zawiera też dwie metody (`setUpClass` i `tearDownClass`), które będą uruchomione jednorazowo przed i po uruchomieniu wszystkich przypadków testowych z tego TestSuite. Możliwe jest zatem wyprowadzenie do tej metod tej klasy inicjacji, która musi mieć miejsce jednorazowo przed wszystkimi testami z danego zestawu.

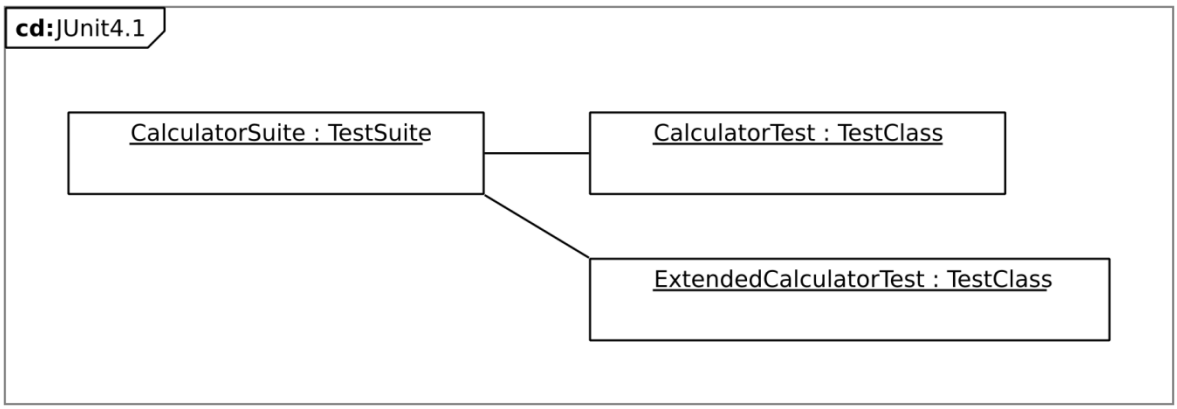


Rysunek 15 Zależność między TestSuite a TestCase w JUnit 4.

```
 1 package sample.code;
 2
 3 import org.junit.AfterClass;
 4 import org.junit.BeforeClass;
 5 import org.junit.runner.RunWith;
 6 import org.junit.runners.Suite;
 7 import org.junit.runners.Suite.SuiteClasses;
 8
 9 @SuiteClasses({CalculatorTest.class, ExtendedCalculatorTest.class})
10 @RunWith(Suite.class)
11 public class CalculatorSuite {
12
13         @BeforeClass
14         public static void setUpClass() throws Exception {
15                 System.out.println("SetUpMethod - dla suite");
16         }
17
```

```
18          @AfterClass
19          public static void tearDownClass() throws Exception {
20                  System.out.println("TearDownMethod - dla suite");
21          }
22
23  }
```

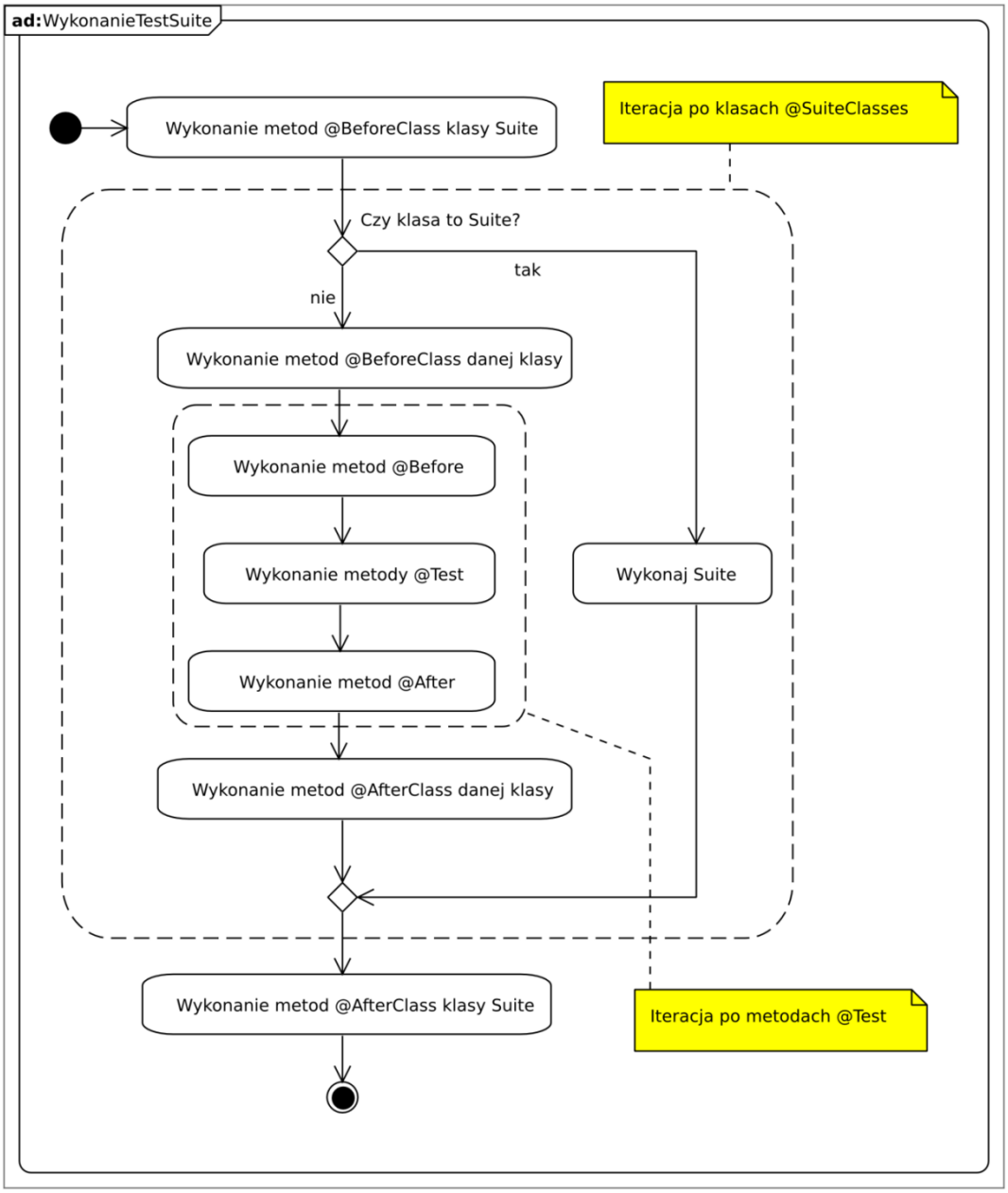
Listing 6 Przykładowy TestSuite w JUnit 4.



Rysunek 16 Przykładowy TestSuite w JUnit4.

Wykonanie testów zgrupowanych w `TestSuite` odbywa się zgodnie ze schematem (Rysunek 17): wykonywane są one w ciągu dwóch pętli, zewnętrznej iterującej po klasach wymienionych w parametrze adnotacji `@SuiteClasses` oraz wewnętrznej iterującej po metodach oznaczonych `@Test` każdej z tych klas.

Dzięki adnotacji `@RunWith`, kod odpowiedzialny za właściwe wykonanie adnotowanej w ten sposób klasy, jest delegowany do obiektu klasy z parametru tej adnotacji. W przypadku wskazania na użycie klasy `Suite` jej kod jest używany do uruchomienia przypadków testowych należących do tego zestawu testów. Jest to dobrym sposobem na dostosowanie frameworku do potrzeb danych testów — używając tej adnotacji i dostarczając własnego delegata (który wielokrotnie uruchamia testy), można przy małym nakładzie pracy z zestawu testów jednostkowych utworzyć testy wydajnościowe.



Rysunek 17 Wywołanie przykładowych testów w JUnit4.

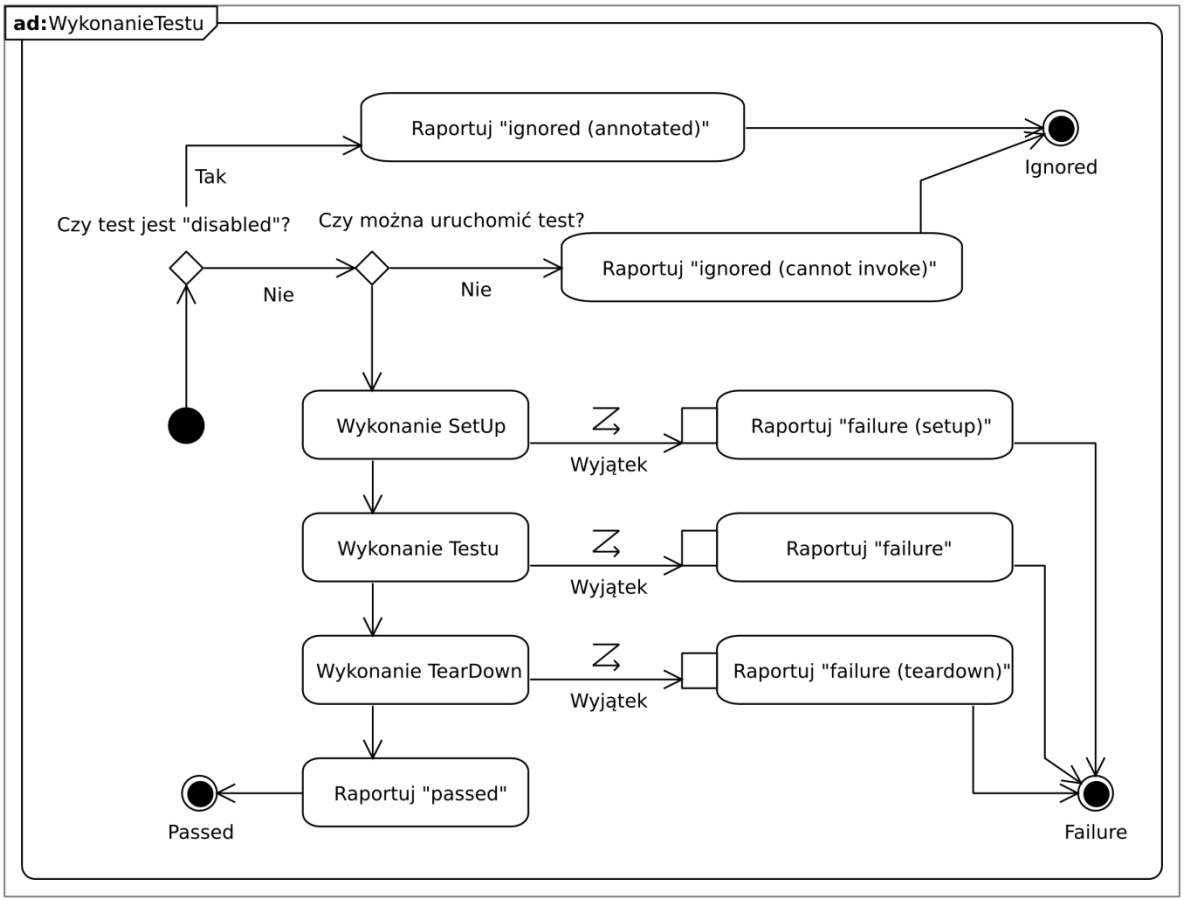
3.3. JTiger

Między wydaniem wersji trzeciej i czwartej frameworku JUnit pojawiło się wiele innych narzędzi, o zbliżonej funkcjonalności, aczkolwiek różniących się niektórymi cechami.

Przykładem takiego oprogramowania jest JTiger, stworzony jako rozszerzenie dla funkcjonalności dostępnej w JUnit 3. Był on pierwszym frameworkiem dla testów jednostkowych wykorzystującym mechanizm adnotacji wprowadzony do języka Java w wersji 5.

W odróżnieniu od JUnit używa on do podział testów przypisanych im w adnotacjach etykiety, a nie zestawów `TestSuite`. Testy podzielone są wówczas, nie według struktury hierarchicznej, ale według przynależności do zbiorów (kategorii), które mogą dowolnie na siebie zachodzić [9].

Również zbiór możliwych wyników testów jest inny. Rozróżnione zostały błędy powstałe w czasie inicjacji środowiska testowego, od błędów pochodzących z przebiegu właściwego testu (Rysunek 18). Umożliwia to szybsze dotarcie do przyczyn zgłaszania błędów przez framework, gdyż błędy wykryte w testowanym oprogramowaniu daje się natychmiast oddzielić od błędów kodu testującego.



Rysunek 18 Możliwy przebieg testu w JTiger.

3.4. TestNG

TestNG jest frameworkim powstałym przed czwartą wersją JUnit. Poza przedstawionym już wcześniej, wykorzystaniem adnotacji i podziałem testów na kategorię, wyróżniającą go cechą jest konfiguracja przebiegu testów przy pomocy plików XML.

```
1 <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
2 <suite name="TestNG">
3         <test verbose="2" name="sample.code.CalculatorTest"
  annotations="JDK">
```

```
4                   <classes>
5                           <class name="sample.code.CalculatorTest" />
6                   </classes>
7           </test>
8   </suite>
```

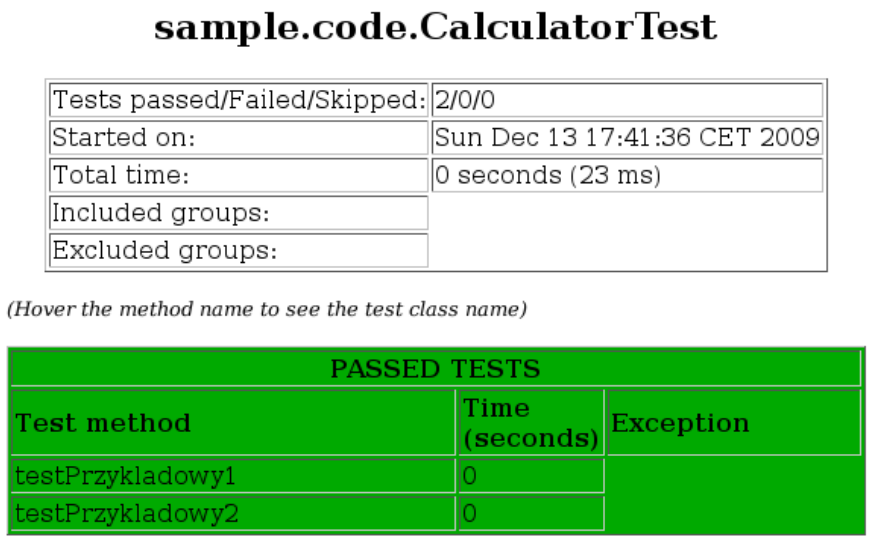
Listing 7 Plik XML definiujący testy dla TestNG.

Przykładowa konfiguracja (Listing 7) definiuje `TestSuite` złożony z przypadków testowych zaimplementowanych w klasie `CalculatorTest`. Pliki XML mogą również importować konfigurację znajdującą się w innym pliku. Daje to duże możliwości podziału tych plików według dowolnych kryteriów, a później łatwe ich łączenie ze sobą w złożone scenariusze testów.

W stosunku do JUnit 4, TestNG oferuje również rozszerzony wybór adnotacji służących do oznaczania metod przygotowujących środowisko do prowadzenia testów:

- `@BeforeSuite` i `@AfterSuite` — związane z wykonaniem `TestSuite`,
- `@BeforeTest` i `@AfterTest` — związane z wykonaniem pojedynczego testu,
- `@BeforeGroups` i `@AfterGroups` — związane z wykonaniem kategorii testów,
- `@BeforeClass` i `@AfterClass` — związane z wykonaniem testów z danej klasy,
- `@BeforeMethod` i `@AfterMethod` — związane z wykonaniem pojedynczej metody.

Udostępniony jest również testerom mechanizm `DataProvider`, pozwalający na odizolowanie danych wykorzystywanych w testach, od implementacji samych przypadków testowych. Możliwe jest to poprzez adnotacje (`@DataProvider`) metody zwracającej dane wykorzystywane w scenariuszu testów, lub poprzez zdefiniowanie tych danych w pliku konfiguracji XML. Daje to nie tylko możliwość prostego rozszerzania zestawu danych testowych, ale i możliwość ponownego użycia danych przypadków testowych dla innych testów (jako że użyte dane testowe mogą zmienić znaczenie samego testu).



sample.code.CalculatorTest

Tests passed/Failed/Skipped:	2/0/0
Started on:	Sun Dec 13 17:41:36 CET 2009
Total time:	0 seconds (23 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS		
Test method	**Time (seconds)**	**Exception**
testPrzykladowy1	0	
testPrzykladowy2	0	

Rysunek 19 Przykładowy raport z testów TestNG.

Wyjątkową cechą tego frameworku jest tworzenie dokładnych raportów w formacie HTML (Rysunek 19) z każdego wykonania testów, co może ułatwić proces raportowania z ich przebiegu, czy archiwizacji wyników.

TestNG jest jedynym z wymienionych tu frameworków, który nie został stworzony wyłącznie z myślą o testach jednostkowych, ale o bardziej szerokim jego zastosowaniu. Nadal jednak nie wspiera on wystarczająco testów złożonych projektów, gdzie pojawiają się setki złożonych scenariuszy testowych (testy integracyjne, systemowe).

3.5. Wady dostępnych frameworków

Omówione wcześniej rozwiązania obarczone są ograniczeniami wynikającymi z ich sztywnych założeń, bądź niewystarczających dla testerów i programistów funkcjonalności. Konsekwencją tego jest między innymi:

(a) tworzenie prostych testów jednostkowych zawsze wymaga implementacji kodu (dotyczy: JUnit, JTiger, TestNG) —może to być pracochłonne i uciążliwe przy dużej ilości niezłożonych przypadków testowych;

(b) sztywne zaszywanie zależności między obiektami `DataProvider`, a testami od nich zależnymi (TestNG), lub całkowity brak takiego mechanizmu (JUnit, JTiger) — ogranicza możliwości ponownego użycia kodu;

(c) konfiguracja frameworku wyłącznie przez zmiany w kodzie źródłowym, bez możliwości wprowadzania zmian po ich kompilacji (JUnit, JTiger) — utrudnia selektywne wykonanie scenariuszy testowych w razie takiej potrzeby;

(d) brak integracji z bibliotekami logującymi, logowanie jest zaszyte w aplikacji, bez możliwości jego jakiejkolwiek konfiguracji (JUnit, JTiger, TestNG) — powoduje między innymi brakiem możliwości zapisywania sformatowanego logu do pliku;

(e) niewystarczające rozróżnienie pomiędzy różnymi rezultatami testów, jedyne możliwe wyniki to „sukces” lub „błąd” (JUnit, TestNG) — utrudnia rozróżnienie przyczyn niepowodzenia testu;

(f) brak narzędzi ułatwiających edycję konfiguracji frameworku zapisywanej w pliku XML (TestNG) — wymaga znajomości jego składni i utrudnia tym samym konfigurację;

(g) brak możliwości uruchomienia frameworku bez dostarczenia pliku XML z konfiguracją (TestNG) — wydłuża czas spędzony na przygotowaniu zestawu testów przed ich wykonaniem;

(h) niewystarczające możliwości dostosowania do potrzeb użytkownika, zbyt ściśle powiązane ze sobą komponenty tych frameworków (JUnit, JTiger, TestNG) — wiele potrzebnych przez użytkowników rozszerzeń może nie być możliwych do zaimplementowania;

(i) brak możliwości porównywania wielu wyników zebranych w czasie wykonania testu, porównywanie jest przerywane zaraz po odnalezieniu pierwszej różnicy (JUnit, JTiger, TestNG) — jest to zgodne z założeniami testów jednostkowych, ale niepraktyczne przy innych rodzajach testów.

4. Koncepcja nowego rozwiązania

Framework testowy powinien dawać możliwość szybkiego jego użycia do testów jednostkowych, tak by programiści mogli go używać równolegle z tworzeniem kodu. Musi się to odbywać możliwie łatwo i szybko, aby zmniejszyć niezbędny do tego nakład pracy. Zakłada się też, że powinien on umożliwiać implementacje, złożonych scenariuszy testów, jakie niezbędne mogą być w czasie testów integracyjnych czy systemowych.

Dodatkową cechą takiego frameworku powinna być możliwość jego rozszerzania, tak aby możliwe było poszerzanie jego funkcjonalności wraz ze wzrostem złożoności oprogramowania, którego testy ma wspierać. Taka cecha dawałaby też większe prawdopodobieństwo możliwości jego zastosowania w zadaniach, które nie są jeszcze znane w czasie jego projektowania.

4.1. Wymagania funkcjonalne i niefunkcjonalne

(I) Odpowiedzialność za wykonanie i gromadzenie wyników testów powinna należeć do frameworku.

(II) Framework powinien udostępniać zestaw funkcji porównujących wyniki testów. Użytkownik powinien mieć możliwość zdecydowania czy znalezienie różnicy w wynikach ma skutkować natychmiastowym przerwaniem testu (i).

(III) Udostępniona powinna być możliwość grupowania TestCase w TestSuite, dla zwiększenia czytelności dużych zbiorów przypadków testowych.

(IV) Framework powinien integrować się z narzędziami logującymi, poprzez udostępnienie jednorodnego interfejsu umożliwiającego logowanie z poziomu kodu scenariuszy testowych (d).

(V) Konfiguracja za pomocą mechanizmu adnotacji, powinna umożliwiać programistom szybkie tworzenie testów jednostkowych dla tworzonego przez nich kodu, bez konieczności tworzenia plików XML z konfiguracją (g).

(VI) Konfiguracja frameworku powinna być możliwa również za pomocą plików XML. Umożliwiłoby to jej modyfikacje bez potrzeby ponownej kompilacji, przy pomocy dowolnego edytora, lub za pomocą języków skryptowych (c).

(VII) Edycja plików XML z konfiguracją powinna być możliwa również w środowisku graficznym będącym częścią frameworku (f).

(VIII) Mechanizm konfiguracji musi umożliwiać uruchamianie zarówno wybranych, pojedynczych scenariuszy testów jak i ich zestawów, dając dużą dowolność testerom uruchamiającym przypadki testowe.

(IX) Proste testy jednostkowe powinny móc być tworzone w graficznym środowisku użytkownika, poprzez podanie danych wejściowych, określenie przedmiotu testu (testowanej metody danej klasy) i oczekiwanych rezultatów. Odbywałoby się to bez konieczności jego implementacji jako alternatywne podejście do tworzenia tego rodzaju testów (a).

(X) Framework musi być złożony z wymiennych modułów, co umożliwi jego rozszerzanie i modyfikacje w razie takiej potrzeby przez programistów używających go (h). Podział na komponenty zwiększa też możliwości ich ponownego użycia w przyszłości.

(XI) Udostępniony powinien być mechanizm zmiany zestawów kategorii do których klasyfikowane są wyniki testów po ich wykonaniu. Umożliwi to zwiększenie granulacji wyników testów w razie takiej konieczności (e).

(XII) Programista powinien móc użytkować framework bez jego pełnej znajomości. Użytkowanie podstawowej funkcjonalności powinno przypominać korzystanie z takich rozwiązań jak JUnit czy TestNG, zapewniając zgodność z przyzwyczajeniami ich użytkowników.

(XIII) Umożliwione powinno zostać korzystanie z obiektów dostarczających dane testowe dla scenariuszy testów, obiekty te powinny być przypisywane do scenariuszy testowych na podstawie konfiguracji pobieranej z pliku XML (b).

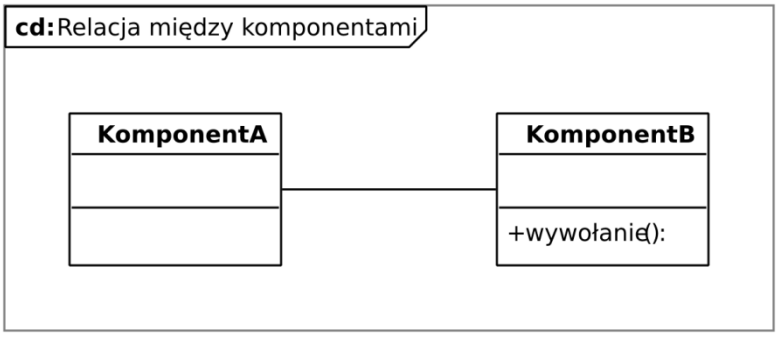
4.2. Wykorzystane wzorce projektowe i paradygmaty

4.2.1. Inverse of Control

Wzorzec Inverse of Control (odwrócenie sterowania) jest paradygmatem (określanym też jako wzorzec projektowy) polegającym na delegacji odpowiedzialności za kontrolę wykonania z komponentów–składowych systemu do frameworku (zwanego kontenerem).

Frameworki kontrolujące przebieg testów, omówione w rozdziale 3, realizują taki wzorzec. Dzięki temu programiści nie są obarczeni koniecznością tworzenia kodu nadzorującego wykonanie testów, a jedynie implementację scenariuszy testowych. Wykonanie tych scenariuszy należy do frameworku którego używają. Dzięki temu skraca się czas implementacji oraz zmniejsza prawdopodobieństwo wystąpienia błędów.

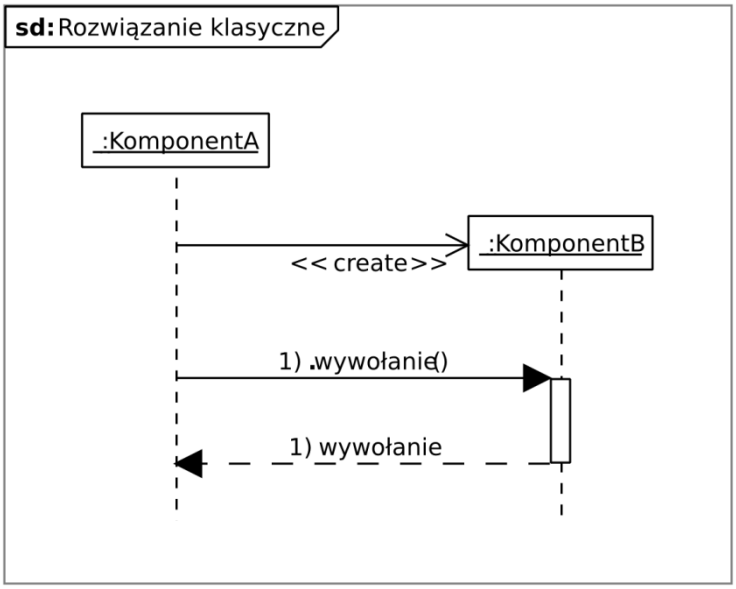
4.2.2. Dependency Injection



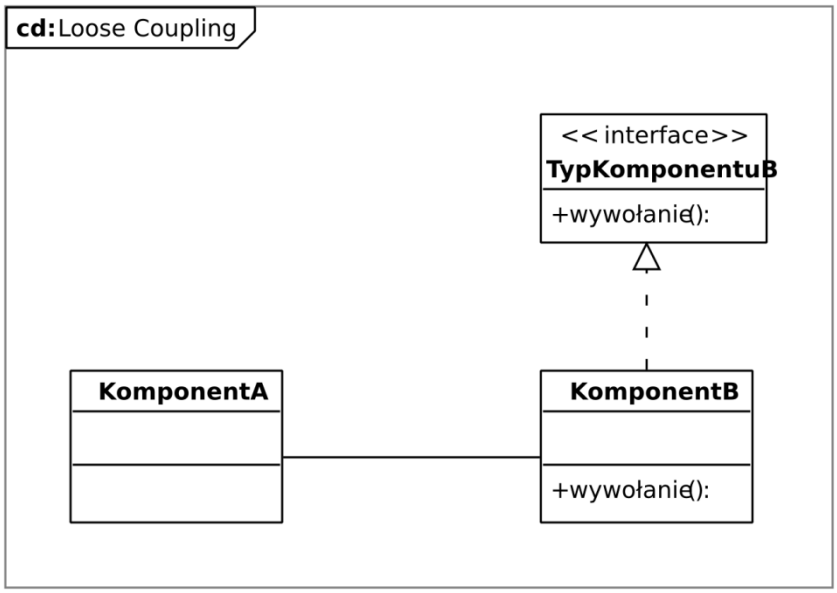
Rysunek 20 Relacja między przykładowymi komponentami.

Dependency Injection (wstrzykiwanie zależności) jest wzorcem projektowym spokrewnionym z IoC (Inverse of Control). Używany jest, gdy oprogramowanie podzielone zostało na moduły (Rysunek 20) pozostające ze sobą w relacji tak, że jeden z nich (`KomponentA`) wykorzystuje funkcjonalność udostępnianą przez drugi (`KomponentB`).

Bez zastosowania tego wzorca, w czasie wykonania programu, `KomponentA` tworzy instancje `KomponentB` i wywołuje na nim niezbędne operacje (Rysunek 21). Jego zastosowanie polega na wykorzystaniu kontenera przechowujący instancje komponentów wchodzących w skład danego systemu. Komponenty te komunikują się ze sobą za pośrednictwem interfejsów (Rysunek 22), bez znajomości implementacji stojącej za nimi (tzw. Loose Coupling). Korzystają one z referencji do pozostałych składowych systemu, lecz nigdy nie tworzą nowych ich instancji.



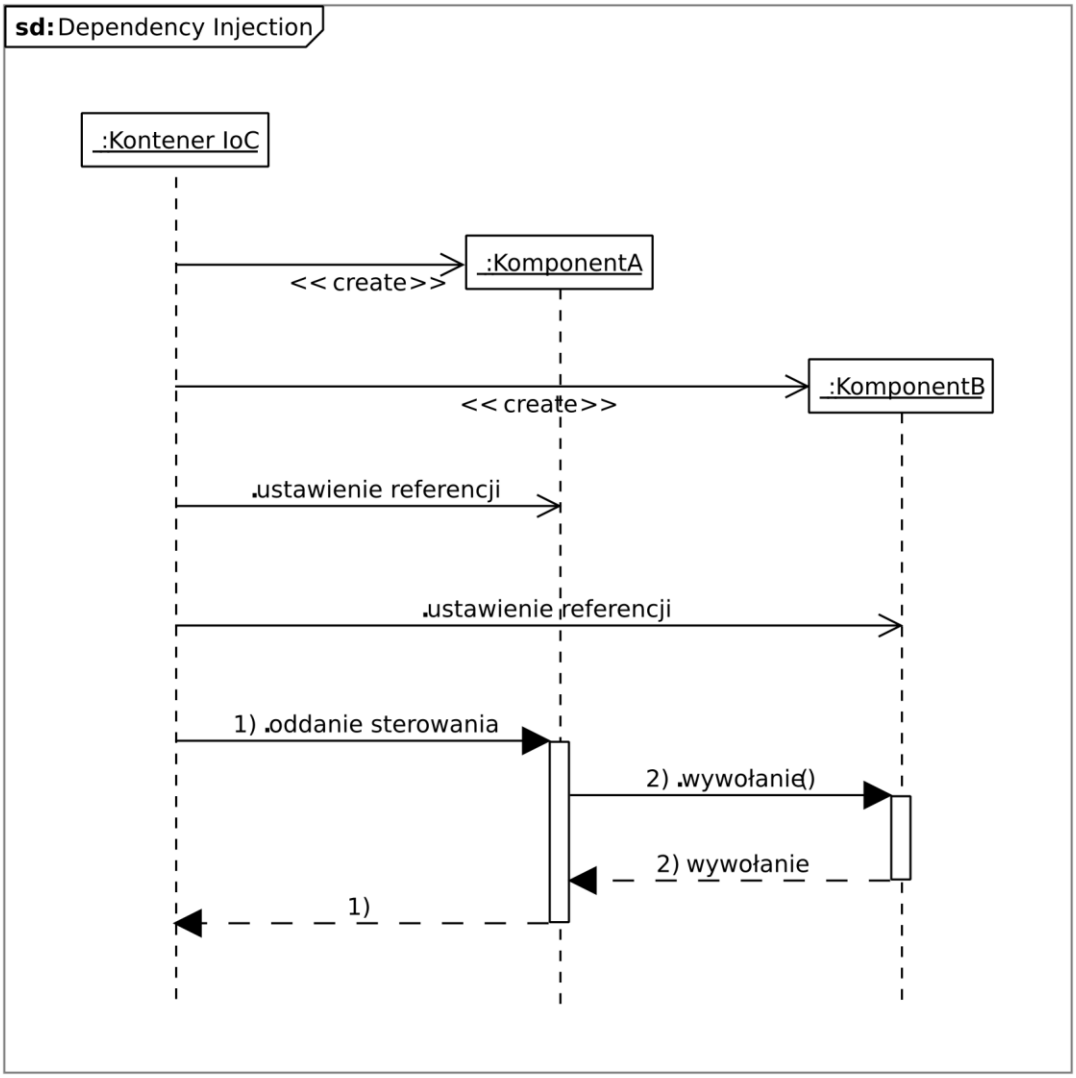
Rysunek 21 Klasyczne użycie komponentów (Rysunek 20).



Rysunek 22 Zależność pomiędzy komponentami z określonym interfejsem.

Zadaniem kontenera Inverse of Control jest w czasie wykonania programu tworzenie instancji wszystkich niezbędnych komponentów i powiązanie ich ze sobą, poprzez ustawianie wszystkich wymaganych referencji w tych obiektach (Rysunek 23). Wówczas, gdy

sterowanie przejmuje `KomponentA`, posiada on już referencje do obiektu implementującego interfejs `TypKomponentuB` (`KomponentB`) i może za nim wykonywać potrzebne operacje.

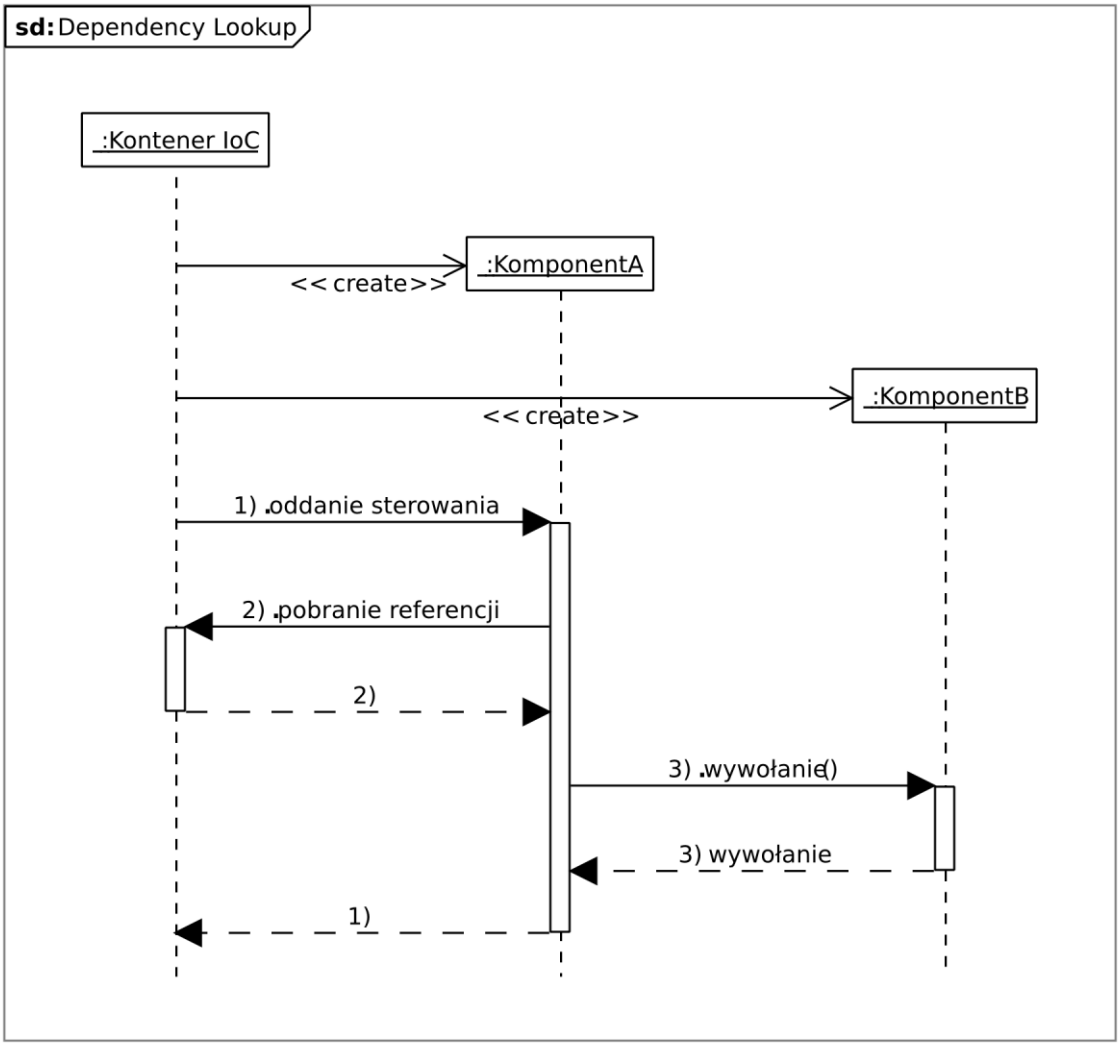


Rysunek 23 Użycie komponentów (Rysunek 20) zgodnie ze wzorcem Dependency Injection.

Dzięki takiemu podejściu możliwe jest tworzenie modułów oprogramowania implementujących dane interfejsy w dowolnej kolejności. Ułatwia to implementacje, czyni architekturę systemu bardziej przejrzystą oraz daje możliwość bezinwazyjnej wymiany tych składowych programu w przyszłości.

Możliwe jest też zastosowanie tego podejścia w frameworkach dla testów automatycznych. Po rozdzieleniu frameworku na pojedyncze moduły spełniające określone funkcje ich wiązanie ze sobą można powierzyć kontenerowi IoC. Dzięki takiemu podejściu framework powinien stać się narzędziem dającym łatwiej się modyfikować i dostosowywać do wymogów użytkowników (programistów i testerów), którzy automatyzują scenariusze testów w oparciu o jego szkielet.

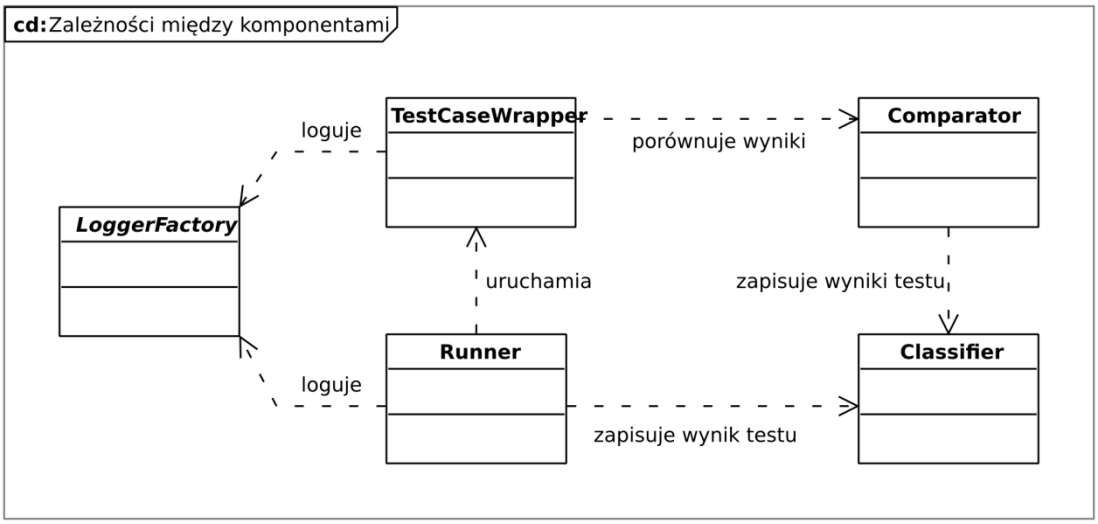
4.2.3. Dependency Lookup



Rysunek 24 Użycie komponentów (Rysunek 20) zgodnie ze wzorcem Dependency Lookup.

Analogicznym dla Dependency Injection rozwiązaniem jest Dependency Lookup. Polega ono na stworzeniu rejestru usług, który przechowuje instancje wszystkich komponentów systemu. W czasie wykonania (Rysunek 24), gdy `KomponentA` wymaga implementacji `TypKomponentuB` odpytuje o nią kontener, który zwraca odpowiedni obiekt (`KomponentB`).

4.2.4. Loose Coupling



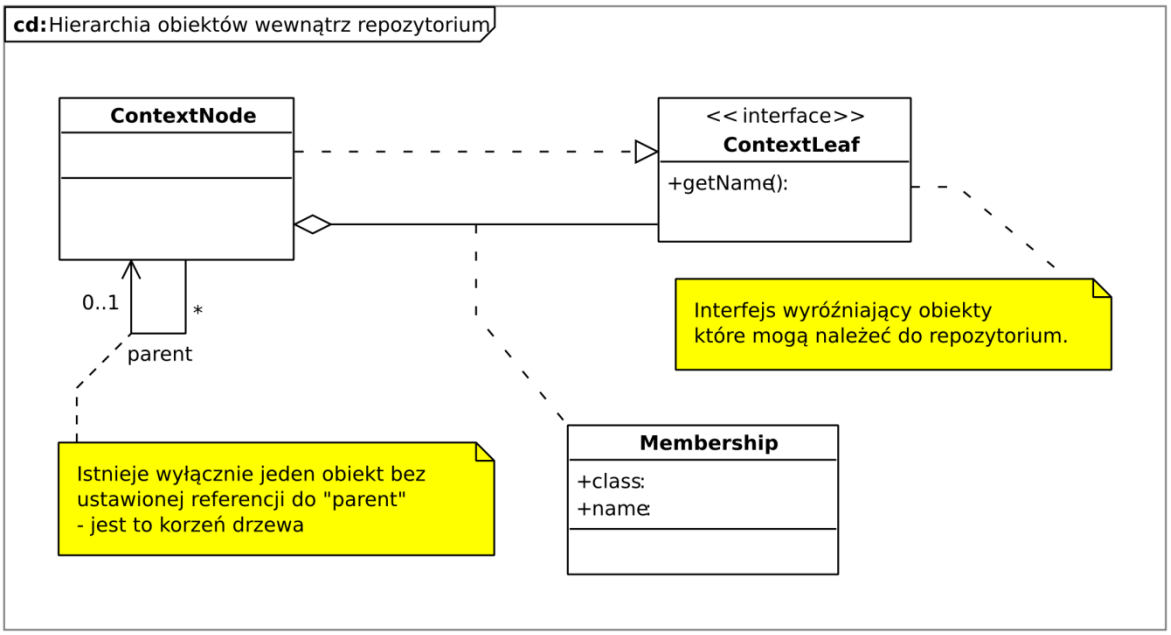
Rysunek 25 Loose Coupling komponentów frameworku.

Loose Coupling jest pojęciem związanym z High Cohesion. Terminy te oznaczają „dobre praktyki" wykorzystywane w inżynierii oprogramowania: niską zależność jednostek składających się na program oraz wysoką ich specjalizację (zawężenie ich odpowiedzialności). Korzystając z opisanych wyżej wzorców (rozdziały 4.2.1, 4.2.2, 4.2.3) możliwe jest tworzenie oprogramowania ściśle przestrzegającego tych zasad.

Framework testowy, jak każde inne oprogramowanie, może również wykorzystywać te szablony. Efektem tego jest wyselekcjonowanie komponentów z których jest on złożony oraz luźnie powiązania między nimi (Rysunek 25). Każdy z tych komponentów realizuje ściśle określone funkcje. Daje to później większe możliwości ponownego wykorzystania kodu oraz zmian w poszczególnych modułach bez wpływu na elementy od nich zależne.

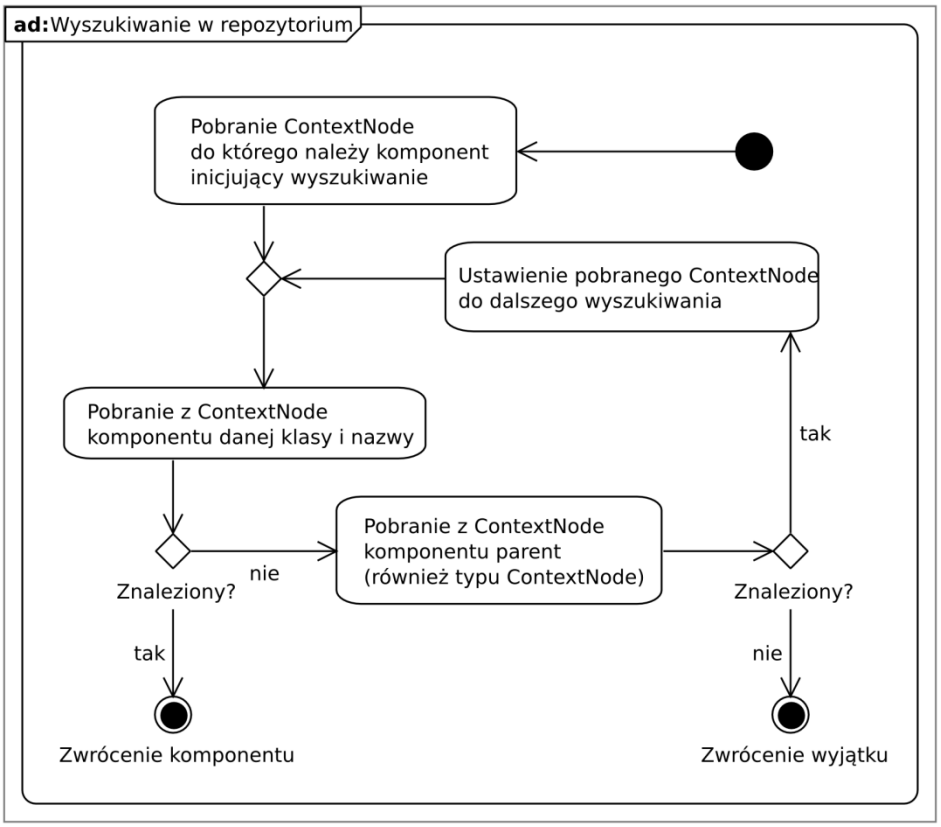
4.3. Repozytorium kontenera frameworku testowego

Prototyp stworzony na potrzeby tej rozprawy zawierać będzie zastosowanie kontenera IoC z Dependency Lookup, jako rozwiązania nieco mniej złożonego w porównaniu do wzorca Dependency Injection. Kontener ten jednak cechować się będzie rozszerzoną funkcjonalnością wyszukiwania zależnych obiektów w jego wnętrzu.

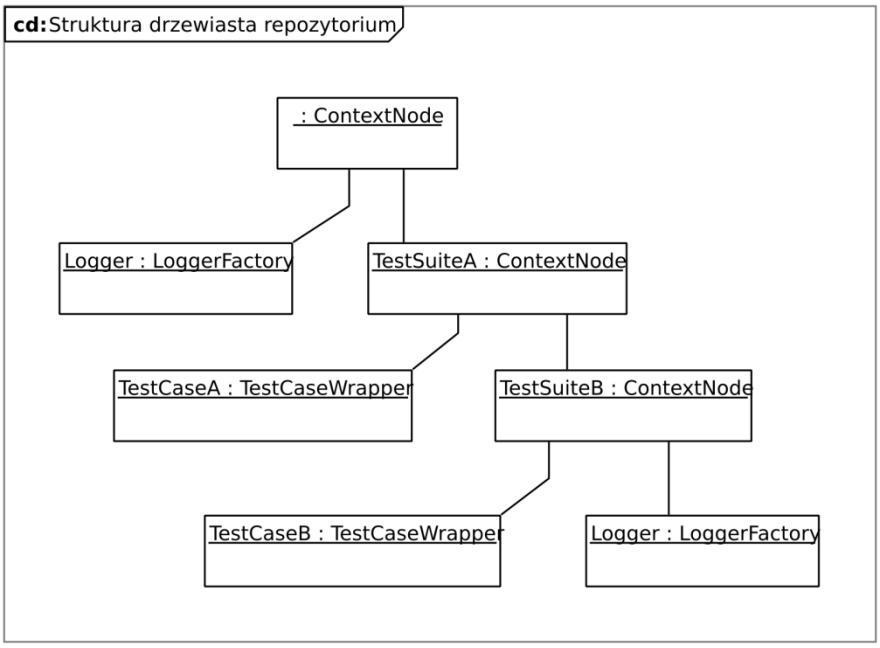


Rysunek 26 Hierarchia obiektów należących do drzewiastego repozytorium Dependency Lookup.

Komponenty reprezentowane będą przez klasy (Rysunek 26) implementujące interfejs `ContextLeaf` (komponenty frameworku testowego) oraz klasę `ContextNode` grupującą je ze sobą. `ContextNode` również powinna implementować ten interfejs, co umożliwi dołączanie jej instancji do innego `ContextNode` — dzięki temu utworzą one strukturę drzewiastą o dowolnym stopniu zagłębienia (Rysunek 28).



Rysunek 27 Wyszukiwanie komponentów w repozytorium Dependency Lookup.



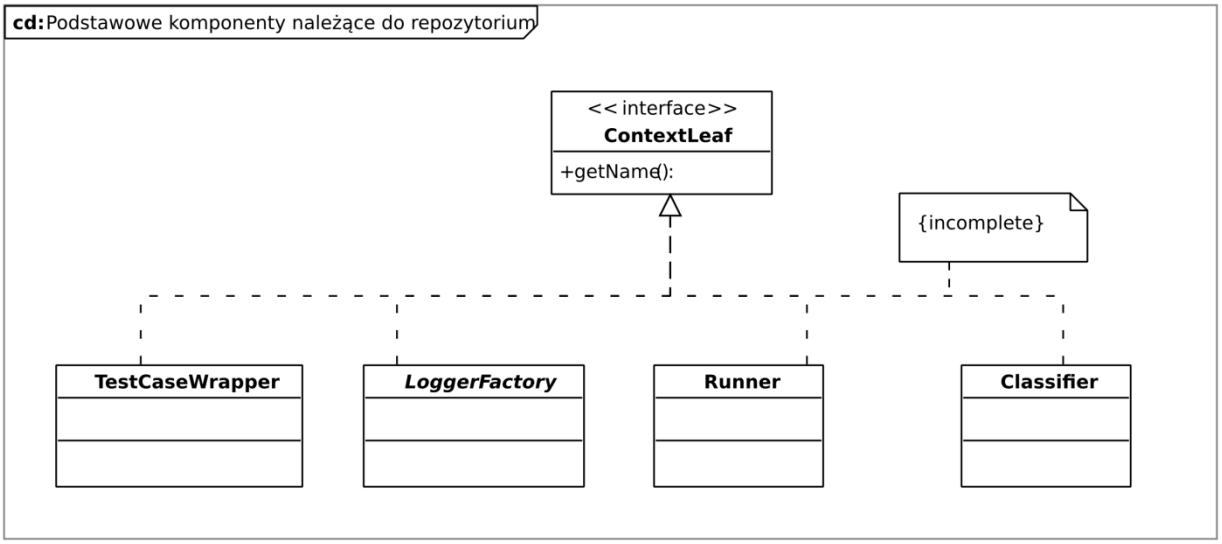
Rysunek 28 Przykładowa struktura repozytorium Dependency Lookup.

Każdy obiekt `ContextLeaf` dla ułatwienia wyszukiwania identyfikowany powinien być wewnątrz `ContextNode` poprzez parę wartości: klasę, której jest instancją oraz nazwę, która go określa (para ta jest unikalna w obrębie `ContextNode`). Odnajdywanie zależności komponentów (Rysunek 27) odbywać się powinno zawsze począwszy od przeszukania `ContextNode` do którego one należą, poprzez wszystkie nadrzędne `ContextNode` (Rysunek 27).

`ContextNode` realizować ma tę samą funkcjonalność co `TestSuite`, znany z innych frameworków — ma on grupować ze sobą przypadki testowe. Jednocześnie dzięki kolejności przeszukiwania repozytorium (Rysunek 28) możliwe będzie przysłanianie obiektów znajdujących się u jego korzenia przez obiekty leżące głębiej w tej strukturze. Dzięki temu obiekty żądające referencji do komponentów mogą uzyskiwać różne wyniki (referencje) w zależności od umiejscowienia w tym drzewie (Rysunek 28 — obiekty typu `TestCaseWrapper` żądając implementacji typu `LoggerFactory` uzyskają różne referencje, gdyż `Logger` należący do `TestSuiteB` „przysłania" `Logger` będący u korzenia drzewa).

4.4. Komponenty frameworku testowego

Framework składać się ma z kodu realizującego IoC oraz z komponentów, które zgodnie z konfiguracją wczytywaną z pliku XML (lub adnotacji Java 6), umieszczane będą wewnątrz repozytorium kontenera Dependency Lookup (Rysunek 29): `TestCaseWrapper` i `UnitCaseWrapper`, `LoggerFactory`, `Runner`, `Classifier`. Uzupełnieniem ich powinny być typy nienależące do tego repozytorium: instancje typu `Comparator`, komponenty odpowiedzialne za wczytywanie konfiguracji i inicjacje repozytorium oraz składowe interfejsu graficznego.



Rysunek 29 Podstawowe komponenty należące do repozytorium Dependency Lookup.

4.4.1. TestCaseWrapper

`TestCaseWrapper` ma opakowywać pojedynczy TestCase. Poza wskazaniem na kod scenariusza w czasie wykonania testu, zawierać powinien wskazanie na zbiór jego wyników zebranych w czasie uruchomienia (o ile TestCase był już uruchomiony), jego nazwę ułatwiająca identyfikację oraz flagę służącą do jego tymczasowego wyłączania ze zbioru scenariuszy (test taki po uruchomieniu otrzyma wynik `skip`).

4.4.2. UnitCaseWrapper

`UnitCaseWrapper` ma być klasą dziedziczącą po `TestCaseWrapper`. Jej zadaniem będzie opakowanie pojedynczego testu jednostkowego. Zawiera ma:

- dane niezbędne do utworzenia instancji klasy której metoda będzie podlegać testowi (dane, które będą użyte jako argumenty do wywołania jej konstruktora);
- nazwę metody, która będzie wywołana w czasie testu;
- dane, które będą wstawione jako jej argumenty;
- oczekiwaną klasę obiektu zwracanego jako jej wyniku;

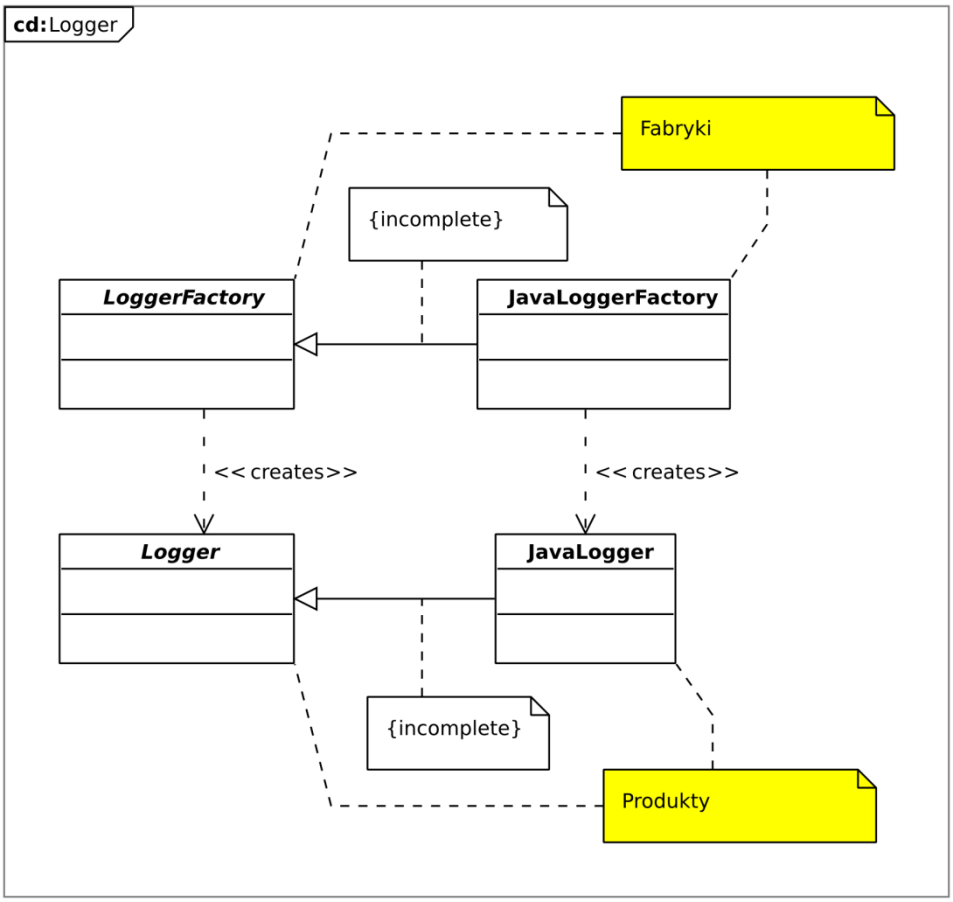
- tekstowa reprezentacją zwracanej wartości;
- bądź rodzaj wyjątku, który powinien być rzucony w czasie jego wykonania (umożliwia to sprawdzenie czy metoda zwraca dany wyjątek zgodnie z oczekiwaniami).

4.4.3. LoggerFactory

`LoggerFactory` (Rysunek 30) ma być abstrakcyjną fabryką, tworząca obiekty typu `Logger`. Każda z konkretnych implementacji klasy `Logger` realizować powinna wzorzec fasady dla dostępnych bibliotek logujących (takich jak Java Logging API, czy Log4j), dając ujednolicony interfejs do ich najczęściej używanej funkcjonalności. Zapewnić to ma odseparowanie kodu scenariusza testowego (i innych komponentów frameworku) od wykorzystywanych bibliotek logujących.

Wydzielenie natomiast abstrakcyjnej `LoggerFactory` da możliwość dodawania kolejnych implementacji tej fabryki do kontenera IoC. Dzięki temu komponenty frameworku z niej korzystające również nigdy nie będą zależne od jej implementacji. Biblioteki logujące będą mogły zatem zostać wymienione w dowolnym czasie, bez wpływu na działanie frameworku.

Udostępnienie tego komponentu umożliwi korzystanie z zalet bibliotek logujących. Fasada je ujednolicająca mapować będzie nazwy `TestCase` dzięki czemu możliwe będzie, korzystając z możliwości konfiguracji tych bibliotek, modyfikowanie poziomu logowania dla wybranych `TestCase`, czy zapisywanie części logów do pliku.



Rysunek 30 Wzorzec Abstract Factory dla komponentów logujących.

4.4.4. Runner

`Runner` ma być komponentem odpowiedzialnym za uruchamianie testów zawartych w `ContextLeaf` do którego będzie należeć. Uruchamiać on będzie metody scenariuszy testów przy pomocy mechanizmu refleksji języka Java. Będzie przechowywać również wszystkie nieprzewidziane wyjątki i w razie ich wystąpienia zgłaszać je do klasyfikatora, aby zanotować niepowodzenie testu. Zawierać on będzie też referencje do `ContextNode` i `TestCaseWrapper` odpowiednich dla aktualnie uruchomionego testu, co umożliwi rozpoznanie ich w czasie uruchomienia z poziomu dowolnego innego komponentu.

4.4.5. Classifier

`Classifier` będzie komponentem odpowiedzialnym za klasyfikację wyników `TestCase` oraz zapisanie go w obiekcie `TestCaseWrapper` tego testu. Klasyfikacja polegać będzie na przypisaniu do każdego scenariusza kategorii:

- `pass` — testy wykonane pomyślnie;
- `passInformation` — testy wykonane pomyślnie, scenariusz zwrócił dane do dalszej analizy;
- `passWarning` — testy wykonane pomyślnie, scenariusz zwrócił dane wymagające uwagi;
- `failAssertion` — test zakończony niepowodzeniem, wynik testu odbiega od oczekiwanego (wykryty błąd systemu podlegającego testowi);
- `failFramework` — test zakończony niepowodzeniem, pojawił się nieoczekiwany błąd (wyjątek) w kodzie scenariusza testowego (oznacza prawdopodobieństwo błędu w scenariuszu);
- `failExternal` — test zakończony niepowodzeniem ze względu na błędy lub niedostępność zewnętrznego systemu niezbędnego do wykonania testów (np. dostępność bazy danych);
- `skipPrev` — test pominięty ze względu na wcześniej wykryte błędy;
- `skip` — test pominięty (gdy został tymczasowo wyłączony).

Poprzez dziedziczenie po typie `Classifier`, możliwa będzie implementacja dodatkowych kategorii. Framework powinien je obsługiwać gdy tylko takowe zostaną dodane do repozytorium Dependency Lookup.

4.4.6. Assercje i komparatory

Framework zwierać też ma zestaw komparatorów, pomocnych przy porównywaniu wyników w scenariuszach testów. Mają to być statyczne metody dostępne z poziomu kodu `TestCase`, zdefiniowane w klasach `Assert` i `Compare` bez możliwości utworzenia ich instancji (zatem dostępne z dowolnego miejsca bez konieczności odpytywania kontenera IoC o ich implementacje). Klasa `Compare` wykonywać ma porównania i zapisywać ich wyniki

przy pomocy komponentu `Classifier`. `Assert`, dodatkowo, w przypadku wykrycia różnicy między wartościami oczekiwanymi, a zgromadzonymi przez test, powinna przerywać jego dalsze wykonanie.

4.4.7. Konfiguracja XML

Framework pozwalać ma na tworzenie repozytorium Dependency Lookup na podstawie pliku XML, określającego wszystkie obiekty jakie mają się w nim znaleźć. Plik ten składałby się z kolejno wymienionych elementów, w hierarchicznej strukturze, z których każdy reprezentuje komponent jaki po uruchomieniu frameworku dodany zostanie do repozytorium.

Framework zaraz po uruchomieniu powinien wczytać taki plik XML i na jego podstawie zainicjować zawartość repozytorium odpowiednimi obiektami. Użycie plików XML ma na celu dostarczenie jasnych zasad jego gramatyki oraz definicji XMLSchema wstępnie walidującej jego poprawność.

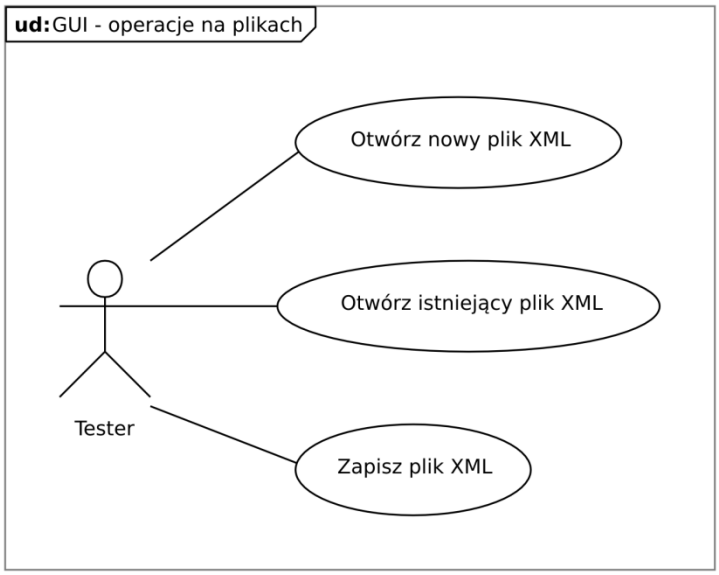
4.4.8. Konfiguracja przy pomocy adnotacji

Istnieć też powinna możliwość konfiguracji frameworku przy pomocy mechanizmu adnotacji języka Java. Za pomocą tego mechanizmu możliwe byłoby wyróżnianie metod będących implementacją `TestCase` oraz definiowanie ich zbiorów jako `TestSuite` (grupowanie ze sobą wielu `TestCase`). Ułatwiłoby to czytanie kodu, gdyż metody testów jednostkowych byłby wyraźnie oznaczone oraz bardziej czytelne dla programistów tworzących te testy równolegle z kodem oprogramowania.

W przypadku uruchamiania frameworku bez podania nazwy pliku konfiguracyjnego, ale z zestawem klas adnotowanych `TestCase` lub `TestSuite` powinna być użyta domyślna konfiguracja frameworku, zaszyta w jego kodzie.

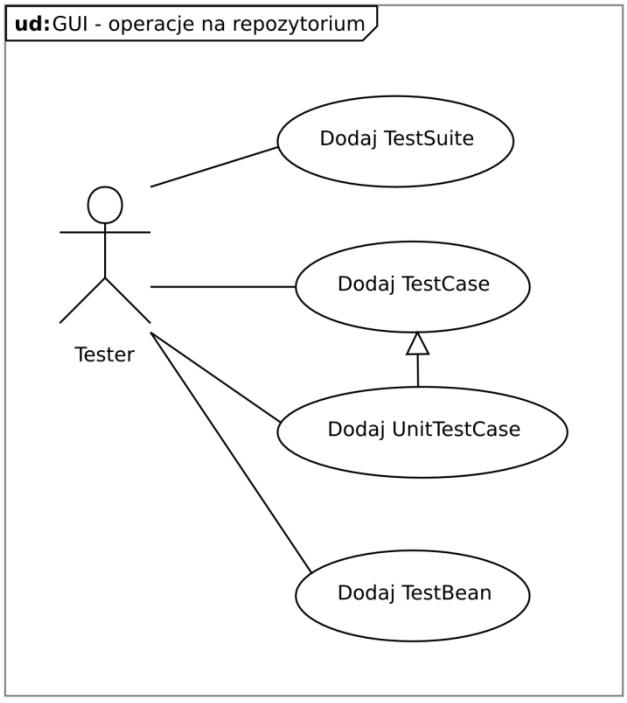
4.5. Interfejs użytkownika

Framework powinien umożliwiać tworzenie plików XML w przystępny dla użytkownika sposób. Odbywać się to powinno poprzez graficzny interfejs użytkownika.



Rysunek 31 Przypadki użycia dla GUI (operacje na plikach)

Interfejs ten ma umożliwiać proste operacje na plikach zawierających konfiguracje, w tym: ich tworzenie, otwieranie oraz zapisywanie (Rysunek 31).

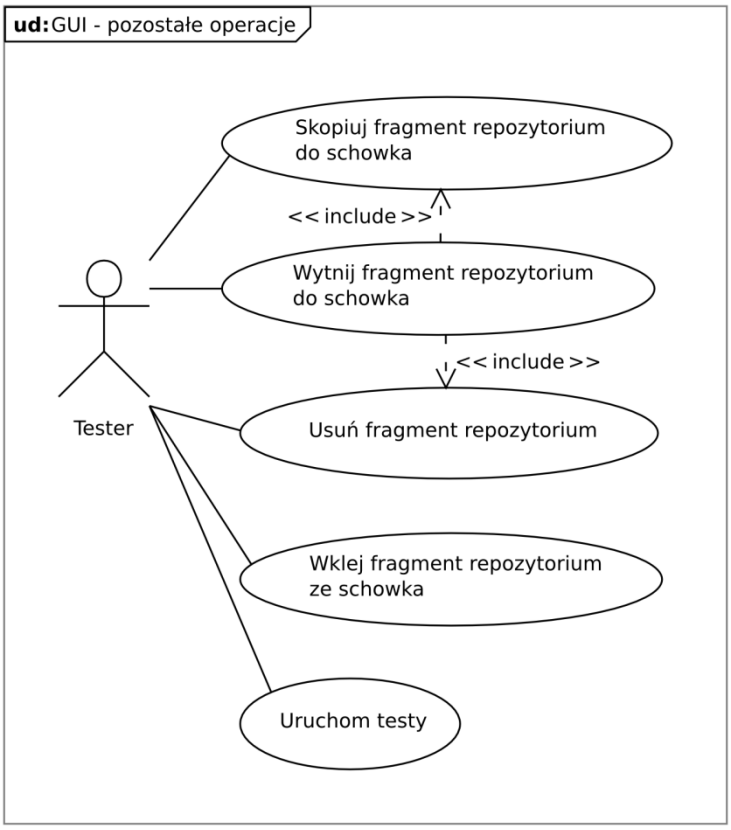


Rysunek 32 Przypadki użycia dla GUI (operacje na repozytorium)

Głowna funkcjonalnością tego interfejsu ma być tworzenie drzewa odzwierciedlającego repozytorium Dependency Lookup. Powinno to być zrealizowane poprzez umożliwienie

użytkownikowi (Rysunek 32) definiowania struktur `TestSuite`, dodawanie do nich dowolnych komponentów, w szczególności typu `TestCase` i `UnitTestCase` poprzez wybieranie ich implementacji z klas.

Aby umożliwić szybszą edycję, użytkownik mógłby korzystać ze schowka systemowego do przechowywania fragmentów tworzonej konfiguracji (Rysunek 33). Możliwe powinno być też uruchamianie stworzonych w ten sposób scenariuszy testów bezpośrednio z interfejsu graficznego.



Rysunek 33 Przypadki użycia dla GUI (pozostałe operacje)

5. Opis stworzonego prototypu

Częścią niniejszej pracy jest w pełni funkcjonalny prototyp, który został zrealizowany na podstawie zaprezentowanych wcześniej wymagań oraz koncepcji rozwiązania. Składa się on z implementacji kontenera realizującego wzorce Dependancy Lookup i Inverse of Control, zestawu podstawowych komponentów, które mogą być w nim umieszczone, a także graficznego interfejsu użytkownika.

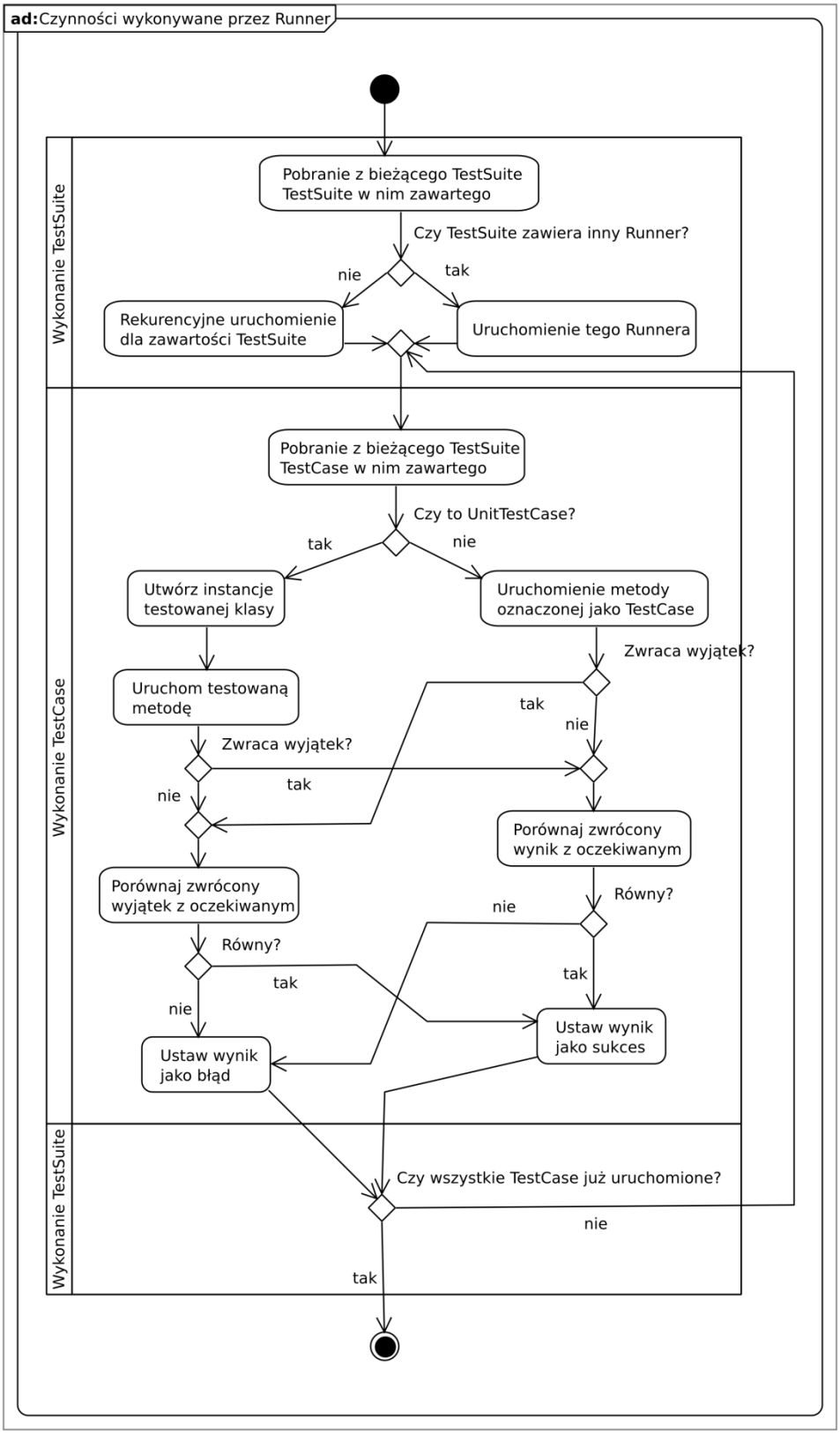
5.1. Stan implementacji komponentów frameworku

Na potrzeby tego prototypu wzorzec Inverse of Control został zaimplementowany z użyciem `HashMap` dostępnych w bibliotece języka Java. Zapewnia to możliwość błyskawicznego odnajdywania komponentów we wnętrzu kontenera, jednocześnie zachowując prostotę jego budowy.

Podział na `TestCase` i `TestSuite` został zrealizowany poprzez reprezentowanie ContextNode jako struktury odpowiadającej `TestSuite`. Dzięki temu spełnione zostały wymagania dotyczące grupowania `TestCase`. `TestSuite` może też zawierać inne komponenty do których możliwe jest odwoływanie się w dowolnym fragmencie kodu scenariusza testowego.

Zaimplementowany został jeden typ komponentu `Runner`. Posiada on podstawową funkcjonalność uruchamiania przypadków testowych oraz zapisywania ich wyników. Wyszukuje on w drzewie obiekty typu `TestCaseWrapper` lub `UnitTestCaseWrapper` i uruchamia testy przez nie definiowane (Rysunek 34). Jeśli w którejś z gałęzi repozytorium odnajdzie on inny komponent typu `Runner` wykonanie testów z tej gałęzi powierzane jest odnalezionemu komponentowi. Dzięki temu dla różnych gałęzi drzewa można przypisać różne rodzaje tego komponentu.

Korzystając z fasady omówionej w rozdziale 4.4.3 możliwe jest logowanie na konsole (jeśli framework uruchomiony został z linii komend), do dziennika logów który wyświetlany jest przez graficzny interfejs użytkownika, lub przeniesienie odpowiedzi za tworzenie logów do Java Logging API.



Rysunek 34 Czynności wykonywane przez Runner

Testerom i programistom został udostępniony zestaw metod `Assert` i `Compare` umożliwiających porównywanie wyników testów z wartościami oczekiwanymi. W prototypie znajduje się implementacja pozwalająca porównywać wszystkie typy proste, jak i typy złożone na podstawie wykonania udostępnionej przez nie metody `equals()`.

Do dyspozycji użytkownika oddane zostają dwie wersje komponentów typu `Classifier`:

- `GenericClassifier` — realizujący domyślny sposób klasyfikacji wyników, zgodny z koncepcją przedstawioną w rozdziale 4.4.5;
- `BinaryClassifier` — odpowiedzialny za uproszczoną klasyfikację o dwóch wartościach: `true` lub `false`, przypisywanych do każdego wykonanego testu. Demonstruje on możliwości elastycznego dopasowania frameworku poprzez wymianę jego komponentów.

Obiektem odpowiedzialnym za tworzenie kontenera i umieszczenie w nim wszystkich komponentów jest implementacja typu `Executive`. Prototyp zawiera dwie jego wersje: uruchamianą z poziomu linii komend oraz uruchamianą w środowisku graficznym. Jest on odpowiedzialny za wczytanie konfiguracji z plików XML i adnotacji, oddanie sterowania do odpowiedniego obiektu `Runner` (znajdującego się u korzenia drzewa repozytorium Dependancy Lookup) oraz wyświetlanie wyników wykonywania scenariuszy testowych.

Konfiguracja przy pomocy pliku XML została zaimplementowana korzystając z funkcjonalności biblioteki Apache XMLBeans. Zapewnia ona wczytywanie, zapisywanie oraz walidację poprawności konfiguracji. Dostępna jest definicja XMLSchema zawartości tego rodzaju dokumentów XML, co w czasie ich edycji w dowolnym edytorze umożliwia na bieżąco sprawdzanie poprawności jego składni oraz podpowiadanie możliwych jego elementów.

Konfiguracja przy pomocy adnotacji jest uzupełnieniem dla plików XML. Framework udostępnia następujące adnotacje, przypominające dostępne w frameworku JUnit (XII):

- `@TestCase` — służąca jako oznaczenie pojedynczego przypadku testowego; dzięki tej adnotacji możliwe jest odnajdywanie przez framework implementacji przypadków testowych;

- `@TestSuite` — grupujący przypadki testowe. Umożliwia szybkie grupowanie ze sobą kilku `TestCase` na potrzeby testów jednostkowych, bez konieczności używania konfiguracji XML.

Adnotacje te mogą zawierać parametr określający nazwę definiowanego przy ich pomocy przypadku testowego bądź `TestSuite` (domyślnie nazwa ta tworzona jest na podstawie nazwy klasy lub metody wobec której zastosowana jest adnotacja). Wyszukiwanie adnotacji w ciele klas, uruchamianie metod i tworzenie instancji testowanych klas zostało zaimplementowane przy pomocy mechanizmu refleksji języka Java.

5.2. Przykładowe zastosowanie frameworku

```
 1 package sample.code;
 2
 3 public class ExtendedCalculator extends Calculator {
 4
 5         public ExtendedCalculator(int initialState) {
 6                 super(initialState);
 7         }
 8
 9         public int mul(int operand) {
10                 return currentState *= operand;
11         }
12
13         public int div(int operand) {
14                 return currentState /= operand;
15         }
16
17 }
 1 package sample.code;
 2
 3 public class Calculator {
 4
 5         protected int currentState;
 6
 7         public Calculator(int initialState) {
 8                 currentState = initialState;
 9         }
10
```

```
11          public int sum(int operand) {
12                  return currentState += operand;
13          }
14
15          public int getResult() {
16                  return currentState;
17          }
18
19  }
```

Listing 8 Klasy użyte jako przedmiot testów do prezentacji frameworku.

Listing 8 zawiera dwie przykładowe klasy poddane testom przy pomocy omawianego tu prototypu frameworku. Testy te zostały zaimplementowane przy wsparciu mechanizmu adnotacji języka Java, wyróżniającego metody pełniące funkcje przypadków testowych, których implementacja została również podzielona na dwie klasy (Listing 9). Przedstawiają one zarówno scenariusze pozytywne (`simpleTestCase`, `division`, `multiplication`) jak i jeden błędny (`failerTestSuite`), który ze względu na wprowadzony tam (i zamierzony) błąd powinien skutkować zgłoszeniem błędu po wykonaniu testu. Każda z klas je definiujących (Listing 9) pełni rolę `TestSuite`, grupując ze sobą metody znajdujące się w ich ciele i oznaczone adnotacją `@TestCase`. Aby wykonać wszystkie testy należy posłużyć się dodatkowym, nadrzędnym `TestSuite` (III) (Listing 10). Uruchomienie tak zdefiniowanego zestawu przypadków testowych nie wymaga użycia konfiguracji XML (V). Nazwę klasy zawierającej przypadki testowe można podać jako argumentu w linii komend uruchamiającej framework, co skutkuje uruchomieniem tych testów (podobnie jak w JUnit (XII)).

Przypadek testowy `alwaysFail` klasy `ExtendedCalculatorTest` (Listing 9) prezentuje możliwość zawarcia kilku porównań wartości oczekiwanych z wynikiem testu. Porównanie takie (zrealizowane przy pomocy metod klasy `Compare`) nie przerywa wykonania testu w przypadku znalezienia błędu, ale kontynuuje jego wykonanie raportując każdy z wykrytych błędów niezależnie (II).

```
1  package sample.code;
2
3  import org.cafet.comparator.Assert;
4  import org.cafet.config.annotation.TestCase;
5  import org.cafet.config.annotation.TestSuite;
6
```

```
 7 @TestSuite
 8 public class CalculatorTest {
 9
10         @TestCase
11         public static void simpleTestCase() {
12                 Calculator calculator = new Calculator(0);
13                 calculator.sum(100);
14                 Assert.assertEquals(100, calculator.getResult());
15         }
16
17         @TestCase
18         public static void failerTestCase() {
19                 Calculator calculator = new Calculator(100);
20                 calculator.sum(100);
21                 Assert.assertEquals(100, calculator.getResult()); //
   always fails
22         }
23
24 }
 1 package sample.code;
 2
 3 import org.cafet.comparator.Assert;
 4 import org.cafet.comparator.Compare;
 5 import org.cafet.config.annotation.TestCase;
 6 import org.cafet.config.annotation.TestSuite;
 7
 8 @TestSuite
 9 public class ExtendedCalculatorTest {
10
11         @TestCase
12         public static void division() {
13                 ExtendedCalculator calculator = new
   ExtendedCalculator(10);
14                 calculator.div(2);
15                 Assert.assertEquals(5, calculator.getResult());
16         }
17
18         @TestCase
19         public static void multiplication() {
20                 ExtendedCalculator calculator = new
   ExtendedCalculator(10);
21                 calculator.div(2);
22                 Assert.assertEquals(5, calculator.getResult());
23         }
24
25         @TestCase
26         public static void alwaysFail() {
27                 ExtendedCalculator calculator = new
   ExtendedCalculator(10);
28                 calculator.div(2);
29                 // 1st check
```

```
30                  Compare.assertEquals("Expected: 10, found:" +
    calculator.getResult(), 10, calculator.getResult()); // always fails
31                  // 2nd check also executed
32                  Compare.assertEquals("Expected: 15, found:" +
    calculator.getResult(), 15, calculator.getResult()); // always fails
33          }
34  }
```

Listing 9 Klasy implementujące przykładowe testy.

```
1  package sample.code;
2
3  import org.cafet.config.annotation.TestSuite;
4
5  @TestSuite(name="CalculatorTests", include={CalculatorTest.class,
   ExtendedCalculatorTest.class})
6  public class Suite {
7  }
```

Listing 10 Klasa grupująca testy w `TestSuite`.

Po wykonaniu zaprezentowanych wcześniej przypadków testowych w środowisku tekstowym framework przedstawi dziennik z ich wykonania (Listing 11). Przedstawia on kolejne wykonywane `TestSuite` oraz przypadki testowe, po których każdorazowo zapisywany jest ich wynik (oraz stacktrace w przypadku błędu). Po zakończeniu procesu ich wykonania wypisywane jest podsumowanie określające ich ilość oraz poszczególnych wyników testów (wraz procentem ich udziałem wśród wszystkich testów wykonanych) (I). Dziennik ten tworzony jest przez framework używając określonej w konfiguacji biblioteki logującej (IV). Dla testu `alwaysFail` (zawierającego kilka porównań) w dzienniku widoczna jest lista wykrytych błędów przez ten przypadek.

```
Jan 31, 2010 11:50:37 PM
FINER: Entering TestSuite in
Jan 31, 2010 11:50:37 PM
FINER: Entering TestSuite in CalculatorTests
Jan 31, 2010 11:50:37 PM
FINER: Entering TestSuite in CalculatorTests.ExtendedCalculatorTest
Jan 31, 2010 11:50:37 PM
FINER: Executing: division in CalculatorTests.ExtendedCalculatorTest
Jan 31, 2010 11:50:37 PM
FINE: Executed: division in CalculatorTests.ExtendedCalculatorTest
                      result: PASSED


Jan 31, 2010 11:50:37 PM
```

```
FINER: Executing: alwaysFail in CalculatorTests.ExtendedCalculatorTest
Jan 31, 2010 11:50:37 PM
INFO: Executed: alwaysFail in CalculatorTests.ExtendedCalculatorTest
                      result: FAILED_ASSERTION (!)

    java.lang.AssertionError: Expected: 10, found:5

     sample.code.ExtendedCalculatorTest.alwaysFail(ExtendedCalculatorTest.j
     ava:30)
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

     sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
     ava:39)
    java.lang.AssertionError: Expected: 15, found:5

     sample.code.ExtendedCalculatorTest.alwaysFail(ExtendedCalculatorTest.j
     ava:32)
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

     sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
     ava:39)

Jan 31, 2010 11:50:37 PM
FINER: Executing: multiplication in CalculatorTests.ExtendedCalculatorTest
Jan 31, 2010 11:50:37 PM
FINE: Executed: multiplication in CalculatorTests.ExtendedCalculatorTest
                      result: PASSED


Jan 31, 2010 11:50:37 PM
FINER: Leaving TestSuite in CalculatorTests.ExtendedCalculatorTest
Jan 31, 2010 11:50:37 PM
FINER: Entering TestSuite in CalculatorTests.CalculatorTest
Jan 31, 2010 11:50:37 PM
FINER: Executing: failerTestCase in CalculatorTests.CalculatorTest
Jan 31, 2010 11:50:37 PM
INFO: Executed: failerTestCase in CalculatorTests.CalculatorTest
                      result: FAILED_ASSERTION (!)

    java.lang.AssertionError: null
        sample.code.CalculatorTest.failerTestCase(CalculatorTest.java:21)
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

     sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
     ava:39)

Jan 31, 2010 11:50:37 PM
FINER: Executing: simpleTestCase in CalculatorTests.CalculatorTest
Jan 31, 2010 11:50:37 PM
FINE: Executed: simpleTestCase in CalculatorTests.CalculatorTest
                      result: PASSED


Jan 31, 2010 11:50:37 PM
FINER: Leaving TestSuite in CalculatorTests.CalculatorTest
Jan 31, 2010 11:50:37 PM
FINER: Leaving TestSuite in CalculatorTests
Jan 31, 2010 11:50:37 PM
FINER: Leaving TestSuite in
```

```
Jan 31, 2010 11:50:37 PM
FINE: Statistics:
              FAILED_ASSERTION  40.00%      2
                        PASSED  60.00%      3
                                total:      5
```

Listing 11 Rezultat uruchomienia testów z Listing 10.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <cft:cafet xmlns:cft="http://cafet.org/cafet"
 3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4         xsi:schemaLocation="http://cafet.org/cafet cafet.xsd ">
 5         <cft:suite>
 6                 <cft:name>CalculatorUnitTestsFromXML</cft:name>
 7
 8                 <cft:test xsi:type="cft:UnitTest">
 9                         <cft:class>sample.code.Calculator</cft:class>
10                         <cft:name>addPositiveNumber</cft:name>
11                         <cft:method>sum</cft:method>
12                         <cft:constructorParam
   type="int">10</cft:constructorParam>
13                         <cft:methodParam
   type="int">10</cft:methodParam>
14                         <cft:expectedResult>20</cft:expectedResult>
15                 </cft:test>
16
17                 <cft:test xsi:type="cft:UnitTest">
18                         <cft:class>sample.code.Calculator</cft:class>
19                         <cft:name>addNegativeNumber</cft:name>
20                         <cft:method>sum</cft:method>
21                         <cft:constructorParam
   type="int">10</cft:constructorParam>
22                         <cft:methodParam type="int">-
   10</cft:methodParam>
23                         <cft:expectedResult>0</cft:expectedResult>
24                 </cft:test>
25
26                 <cft:test xsi:type="cft:UnitTest">
27
   <cft:class>sample.code.ExtendedCalculator</cft:class>
28                         <cft:name>divisionByZero</cft:name>
29                         <cft:method>div</cft:method>
30                         <cft:constructorParam
   type="int">10</cft:constructorParam>
31                         <cft:methodParam
   type="int">0</cft:methodParam>
32
   <cft:expectedException>java.lang.ArithmeticException</cft:expectedExce
   ption>
33                 </cft:test>
34
35         </cft:suite>
36 </cft:cafet>
```

Listing 12 XML definiujący przykładowe testy jednostkowe.

Zgodnie z założeniami (VI) możliwe jest też definiowanie `TestSuite` oraz testów (w szczególności testów jednostkowych) przy pomocy plików XML. Przykład przedstawiony powyżej (Listing 12) zawiera definicje `TestSuite` o nazwie `CalculatorUnitTestsFromXML`, zawierającego definicje trzech przypadków testów jednostkowych: 2 pozytywnych (`addPositiveNumber`, `addNegativeNumber`) oraz jednego negatywnego (`divisionByZero`), gdzie oczekiwane jest pojawienie się wyjątku związanego z dzieleniem przez zero. Zmiany tak tworzonej konfiguracji nie wymagają ponownej kompilacji kodu `TestCase`.

Podobnie jak przy scenariuszach definiowanych adnotacjami, po ich uruchomieniu prezentowany jest raport przedstawiający wyniki ich wykonania (Listing 13).

```
Jan 24, 2010 7:12:35 PM
FINER: Entering TestSuite
Jan 24, 2010 7:12:35 PM
FINER: Entering TestSuite in CalculatorUnitTestsFromXML
Jan 24, 2010 7:12:35 PM
FINER: Executing: addPositiveNumber in CalculatorUnitTestsFromXML
Jan 24, 2010 7:12:35 PM
FINE: Executed: addPositiveNumber in CalculatorUnitTestsFromXML
                      result: PASSED


Jan 24, 2010 7:12:35 PM
FINER: Executing: addNegativeNumber in CalculatorUnitTestsFromXML
Jan 24, 2010 7:12:35 PM
FINE: Executed: addNegativeNumber in CalculatorUnitTestsFromXML
                      result: PASSED


Jan 24, 2010 7:12:35 PM
FINER: Executing: divisionByZero in CalculatorUnitTestsFromXML
Jan 24, 2010 7:12:35 PM
FINE: Executed: divisionByZero in CalculatorUnitTestsFromXML
                      result: PASSED
```

```
Jan 24, 2010 7:12:35 PM
FINER: Leaving TestSuite in CalculatorUnitTestsFromXML
Jan 24, 2010 7:12:35 PM
FINER: Leaving TestSuite
Jan 24, 2010 7:12:35 PM
FINE: Statistics:
                        PASSED 100.00%      3
                                 total:     3
```

Listing 13 Wyniki wykonania testów z Listing 12.

Framework umożliwa też tworzenie klas spełniających rolę obiektów typu `DataProvider` (XIII). Klasy takie muszą implementować interfejs `ContextLeaf`, dzięki czemu ich instancje mogą być w czasie wykonania testów umieszczone wewnątrz repozytorium Dependency Lookup, a następnie wykorzystane w kodzie przypadków testowych. `CalculatorDataProvider` (Listing 14) jest przykładem interfejsu takiej klasy, zaimplementowanej poniżej jako `RandomDataProvider`. Służy ona jako generator losowych danych testowych dla testu `simpleTestCase` (klasa `ParameterizedCalculatorTest`). Kod tego przypadku testowego wykorzystuje w czasie swojego przebiegu funkcje biblioteki logującej, która ukryta zostaje za fasadą `LoggerFacade` (IV).

```
 1 package sample.code;
 2
 3 import org.cafet.ctx.ContextLeaf;
 4
 5 public interface CalculatorDataProvider extends ContextLeaf {
 6
 7         public class Data {
 8                 public int initValue;
 9                 public int paramValue;
10                 public int resultValue;
11         }
12
13         public Data createData();
14
15 }
 1 package sample.code;
 2
 3 import java.util.Random;
 4
 5 public class RandomDataProvider implements CalculatorDataProvider {
 6
 7         private Random random = new Random();
 8         private String name;
 9
```

```
10          public RandomDataProvider(String name) {
11                  this.name = name;
12          }
13
14          public Data createData() {
15                  Data data = new Data();
16                  data.initValue = random.nextInt();
17                  data.paramValue = random.nextInt();
18                  data.resultValue = data.initValue + data.paramValue;
19                  return data;
20          }
21
22          @Override
23          public String getName() {
24                  return name;
25          }
26
27  }
```

```
 1  package sample.code;
 2
 3  import org.cafet.comparator.Assert;
 4  import org.cafet.config.annotation.TestCase;
 5  import org.cafet.logger.LoggerFacade;
 6  import org.cafet.logger.LoggerFactory;
 7  import org.cafet.runner.Runner;
 8
 9  import sample.code.CalculatorDataProvider.Data;
10
11  public class ParameterizedCalculatorTest {
12
13          @TestCase
14          public static void simpleTestCase() {
15                  // get logger using default logger factory
16                  LoggerFacade logger =
    Runner.getTestBean(LoggerFactory.class, null).getLogger();
17                  // get data provider (named: "dataprovider") for
    generating testcase data
18                  CalculatorDataProvider dataProvider =
    Runner.getTestBean(CalculatorDataProvider.class, "dataprovider");
19
20                  // get data for the test
21                  Data testData = dataProvider.createData();
22
23                  // log the test data
24                  logger.info("Using initial value:" +
    testData.initValue + " param value:" + testData.paramValue);
25
26                  Calculator calculator = new
    Calculator(testData.initValue);
27                  calculator.sum(testData.paramValue);
28                  int result = calculator.getResult();
29
```

```
30                  // log the test data
31                  logger.info("Expected:" + testData.resultValue + "
   got:" + result);
32
33                  Assert.assertEquals(testData.resultValue, result);
34          }
35
36  }
```

Listing 14 Przykładowy test wraz z obiektem typu `DataProvider`.

Aby tak sparametryzowany test mógł być wykonany niezbędne musi być powiązanie go z odpowiednim obiektem typu `DataProvider`. Wiązanie to realizowane jest poprzez plik konfiguracyjny XML (Listing 15) — obiekt tego typu osadzany jest we wnętrzu repozytorium Dependency Lookup wraz z odpowiednim przypadkiem testowym. Dla dowiedzenia losowego działania takiego testu, został on (poprzez niewielką zmianę w konfiguracji (VIII)) załączony do repozytorium dwukrotnie, pod nazwami: `1st` i `2nd`. W czasie jego uruchomienia w dzienniku zapisywane są dane użyte do testów. Zgodnie z oczekiwaniem z każdym kolejnym przebiegiem testów dane te są inne (Listing 16).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <cft:cafet xmlns:cft="http://cafet.org/cafet"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://cafet.org/cafet cafet.xsd ">
5          <cft:suite>
6                  <cft:name>CalculatorTestsFromXML</cft:name>
7
8                  <cft:bean>
9                          <cft:name>dataprovider</cft:name>
10
   <cft:class>sample.code.RandomDataProvider</cft:class>
11
   <cft:typeClass>sample.code.CalculatorDataProvider</cft:typeClass>
12                 </cft:bean>
13
14                 <cft:test>
15
   <cft:class>sample.code.ParameterizedCalculatorTest</cft:class>
16                         <cft:name>1st</cft:name>
17                         <cft:method>simpleTestCase</cft:method>
18                 </cft:test>
19
20                 <cft:test>
21
   <cft:class>sample.code.ParameterizedCalculatorTest</cft:class>
22                         <cft:name>2nd</cft:name>
23                         <cft:method>simpleTestCase</cft:method>
24                 </cft:test>
```

```
  25
  26          </cft:suite>
  27 </cft:cafet>
```

Listing 15 Konfiguracja XML dla testu wykorzystującego obiekt typu `DataProvider`.

```
Jan 31, 2010 11:54:34 PM
FINER: Entering TestSuite in
Jan 31, 2010 11:54:34 PM
FINER: Entering TestSuite in CalculatorTestsFromXML
Jan 31, 2010 11:54:34 PM
FINER: Executing: 2nd in CalculatorTestsFromXML
Jan 31, 2010 11:54:34 PM CalculatorTestsFromXML.2nd
     (ParameterizedCalculatorTest.java:24)
FINE: Using initial value:-1341353998 param value:-1129414937
Jan 31, 2010 11:54:34 PM CalculatorTestsFromXML.2nd
     (ParameterizedCalculatorTest.java:31)
FINE: Expected:1824198361 got:1824198361
Jan 31, 2010 11:54:34 PM
FINE: Executed: 2nd in CalculatorTestsFromXML
                      result: PASSED


Jan 31, 2010 11:54:34 PM
FINER: Executing: 1st in CalculatorTestsFromXML
Jan 31, 2010 11:54:34 PM CalculatorTestsFromXML.1st
     (ParameterizedCalculatorTest.java:24)
FINE: Using initial value:-640887107 param value:2066466608
Jan 31, 2010 11:54:34 PM CalculatorTestsFromXML.1st
     (ParameterizedCalculatorTest.java:31)
FINE: Expected:1425579501 got:1425579501
Jan 31, 2010 11:54:34 PM
FINE: Executed: 1st in CalculatorTestsFromXML
                      result: PASSED


Jan 31, 2010 11:54:34 PM
FINER: Leaving TestSuite in CalculatorTestsFromXML
Jan 31, 2010 11:54:34 PM
FINER: Leaving TestSuite in
Jan 31, 2010 11:54:34 PM
FINE: Statistics:
                       PASSED 100.00%     2
                              total:      2
```

Listing 16 Wyniki testu wykorzystującego obiekt typu `DataProvider`.

Framework umożliwia też wymianę swoich komponentów składowych na ich alternatywne implementacje (X). Przy pomocy tego mechanizmu możliwa jest między innymi wymiana zestawu wyników które przypisywane są do testów po ich uruchomieniu (XI). Odbywa się to poprzez edycję pliku XML z konfiguracją frameworku (Listing 17). Zgodnie

z przedstawioną wcześniej koncepcją, komponentem za to odpowiedzialnym jest `Classifier`. W przypadku jego zastąpienia inną implementacją (z domyślnego `GenericClassifier` na `BinaryClassifier`) zmienione zostają rezultaty z wykonania testów przedstawiając nowe wyniki (Listing 18).

```
<?xml version="1.0" encoding="UTF-8"?>
<cft:cafet xmlns:cft="http://cafet.org/cafet"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://cafet.org/cafet cafet.xsd ">

        <cft:bean>
                <cft:name></cft:name>

<cft:class>org.cafet.classifier.binary.BinaryClassifier</cft:class>

<cft:typeClass>org.cafet.classifier.Classifier</cft:typeClass>
        </cft:bean>

        <cft:suite>
                <cft:name>AlternativeCalculatorTestsFromXML</cft:name>

                <cft:bean>
                        <cft:name>dataprovider</cft:name>

<cft:class>sample.code.RandomDataProvider</cft:class>

<cft:typeClass>sample.code.CalculatorDataProvider</cft:typeClass>
                </cft:bean>

                <cft:test>

<cft:class>sample.code.ParameterizedCalculatorTest</cft:class>
                        <cft:name>1st</cft:name>
                        <cft:method>simpleTestCase</cft:method>
                </cft:test>

                <cft:test>

<cft:class>sample.code.ParameterizedCalculatorTest</cft:class>
                        <cft:name>2nd</cft:name>
                        <cft:method>simpleTestCase</cft:method>
                </cft:test>

        </cft:suite>
</cft:cafet>
```

Listing 17 Wyniki testu wykorzystującego obiekt typu `DataProvider`.

```
Jan 31, 2010 11:55:43 PM
FINER: Entering TestSuite in
Jan 31, 2010 11:55:43 PM
FINER: Entering TestSuite in AlternativeCalculatorTestsFromXML
Jan 31, 2010 11:55:43 PM
FINER: Executing: 2nd in AlternativeCalculatorTestsFromXML
Jan 31, 2010 11:55:43 PM AlternativeCalculatorTestsFromXML.2nd
     (ParameterizedCalculatorTest.java:24)
FINE: Using initial value:1931522878 param value:-1685056500
Jan 31, 2010 11:55:43 PM AlternativeCalculatorTestsFromXML.2nd
     (ParameterizedCalculatorTest.java:31)
FINE: Expected:246466378 got:246466378
Jan 31, 2010 11:55:43 PM
FINE: Executed: 2nd in AlternativeCalculatorTestsFromXML
                        result: true
Jan 31, 2010 11:55:43 PM
FINER: Executing: 1st in AlternativeCalculatorTestsFromXML
Jan 31, 2010 11:55:43 PM AlternativeCalculatorTestsFromXML.1st
     (ParameterizedCalculatorTest.java:24)
FINE: Using initial value:-1639409052 param value:1097196406
Jan 31, 2010 11:55:43 PM AlternativeCalculatorTestsFromXML.1st
     (ParameterizedCalculatorTest.java:31)
FINE: Expected:-542212646 got:-542212646
Jan 31, 2010 11:55:43 PM
FINE: Executed: 1st in AlternativeCalculatorTestsFromXML
                        result: true
Jan 31, 2010 11:55:43 PM
FINER: Leaving TestSuite in AlternativeCalculatorTestsFromXML
Jan 31, 2010 11:55:43 PM
FINER: Leaving TestSuite in
Jan 31, 2010 11:55:43 PM
FINE: Statistics:
                          true 100.00%     2
                                total:     2
```

Listing 18 Wyniki testu wykorzystującego obiekt typu `DataProvider`.

5.3. Stan implementacji interfejsu graficznego

Interfejs użytkownika, wzorowany na środowisku Eclipse, składa się z dwóch ekranów. Pierwszy, główny, służy edycji plików XML opisujących konfiguracje frameworku (VII) (Rysunek 35). Podzielony jest on na trzy panele:

- pierwszy, znajdujący się w lewej części ekranu, zawiera drzewo reprezentujące kolejne elementy (pliki jar oraz katalogi) wchodzące w classpath środowiska;

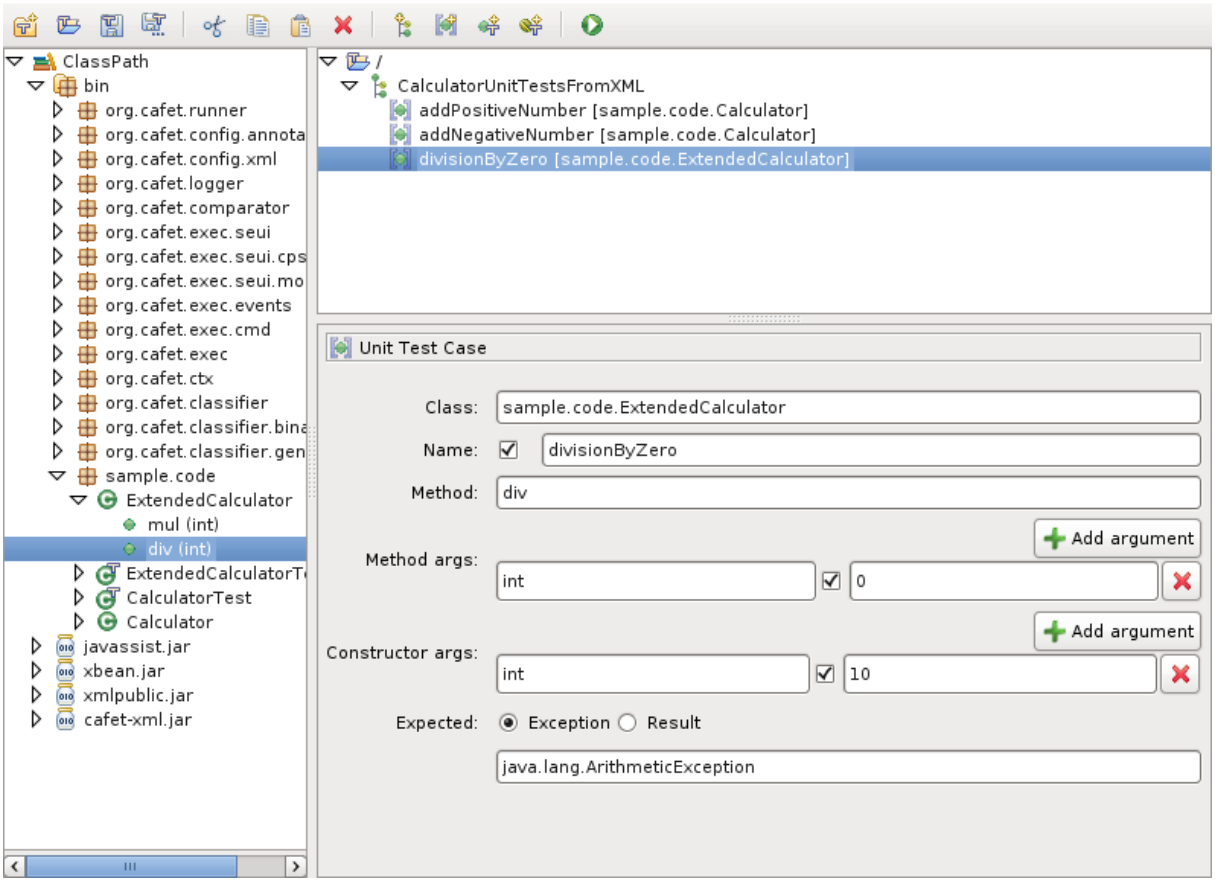
- drugi, wyświetlany w prawej-górnej części ekranu, przedstawia drzewo obiektów, które znajdą się w repozytorium Dependancy Lookup w czasie wykonywania testów;
- trzeci panel znajdujący się u dołu okna i umożliwiający edycję właściwości elementu wybranego w drugim panelu.

Drzewo przedstawiające elementy należąc do classpath może być rozwijane prezentując kolejne trzy poziomy uszczegółowienia: listę zdefiniowanych pakietów dla każdego z elementów classpath, klasy należące do tego pakietu oraz metody tych klas. Operując schowkiem systemowym użytkownik ma możliwość kopiowania nazw klas i metod z tego drzewka do właściwości edytowanych elementów, co odbywa się w panelu u dołu okna. Dzięki takiemu rozwiązaniu w czasie edycji plików XML nie jest wymagana od użytkownika pamięciowa znajomość nazw wszystkich klas które mają być włączone do repozytorium.

Interfejs użytkownika umożliwia operacje na definiowanym repozytorium korzystając z przycisków znajdujących się na pasku narzędzi, u góry głównego okna, kolejno:

- tworzenie nowego pliku XML (nowego repozytorium),
- otwarcie istniejącego pliku XML,
- zapis edytowanego pliku,
- zapis edytowanego repozytorium pod nową nazwą pliku,
- wycięcie do systemowego schowka części (gałęzi) drzewa repozytorium,
- skopiowanie do schowka jego części,
- wklejenie wyciętego wcześniej fragmentu,
- usunięcie zaznaczonych obiektów z repozytorium,
- wstawienie nowego TestSuite do repozytorium,
- wstawienie nowego UnitTestCase,
- wstawienie nowego TestCase,
- wstawienie nowego obiektu repozytorium (np. obiektu klasy spełniającej rolę DataProvider),
- uruchomienie przypadków testowych.

Po zaznaczeniu elementu repozytorium typu takiego jak (`TestSuite`, `UnitTestCase` czy `TestCase`) możliwa jest edycja wszystkich właściwości (takich jak jego nazwa) tego elementu w dolnym panelu ekranu. Zestaw właściwości podlegających edycji zależy od rodzaju elementu gwarantując poprawność tworzonej w ten sposób konfiguracji.

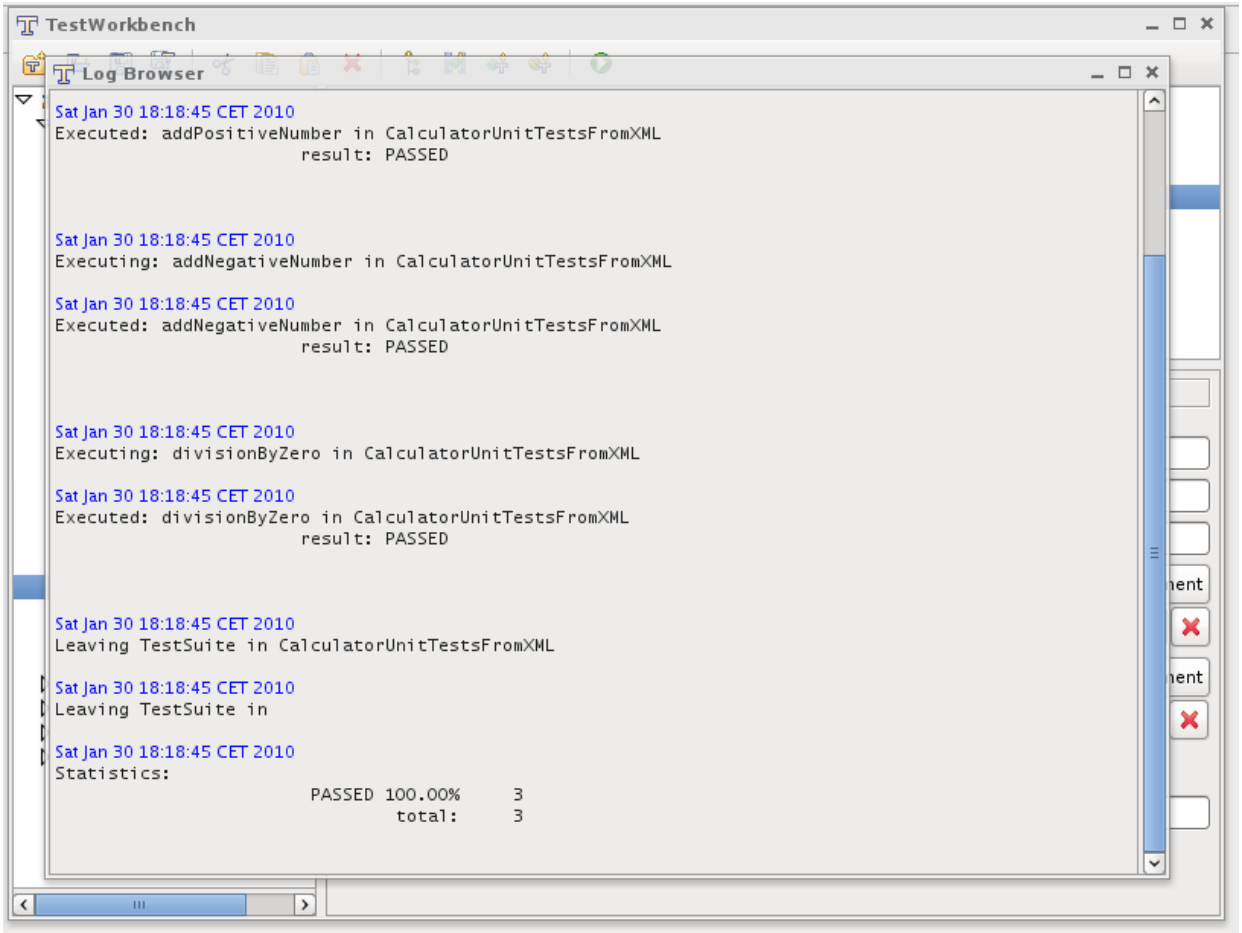


Rysunek 35 Główny ekran interfejsu graficznego (widoczna definicja przykładowego `UnitTestCase`).

Po wybraniu elementu repozytorium typu UnitTestCase (Rysunek 35), możliwe jest określenie: nazwy testowanej klasy, testowanej metody, argumentów przekazywanych tej metodzie, oraz argumentów dla konstruktora tej klasy (o ile wybrana metoda nie jest metodą statyczną) (IX).

Drugi z ekranów prezentuje log wykonania testów, wraz z jego wynikami (Rysunek 36). Przypomina on zawartością swój odpowiednik, tworzony w czasie wykonania

testów w środowisku tekstowym. Wyświetlany jest on w osobnym oknie, zaraz po kliknięciu przez użytkownika przycisku uruchamiającego testy z paska narzędzi głównego ekranu.



Rysunek 36 Dziennik wykonania testów w interfejsie graficznym.

5.4. Zalety i wady przyjętych rozwiązań

Zalety:

- Możliwość szybkiego rozpoczęcia użytkowania frameworku bez pełnej wiedzy o jego budowie czy zasadach funkcjonowania.
- Repozytorium Dependancy Lookup dostarcza funkcjonalność równoważną dla `DataProvider` znanych z rozwiązania TestNG.
- Funkcjonalność framework może być zmieniana poprzez wymianę modułów rezydujących w repozytorium Dependancy Lookup (umożliwiając późniejszą rozbudowę czy dostosowanie do wymogów użytkowników).

Wady:

- Brak możliwości kojarzenia ze sobą pozytywnych wyników wykonania testów jeśli zostały one ze brane za pomocą różnych obiektów `Classifier`.
- Brak możliwości wymuszenia kolejności wykonywania TestCase poprzez określanie zależności pomiędzy nimi — np. gdy po wykryciu błędu przez testy badające spójność bazy danych wiadomym jest, że wszystkie testy zależne od niej zależne nie mają sensu.

5.5. Proponowane kierunki rozwoju

Dzięki dużej modułowości prototyp powinien dawać duże możliwości dalszego rozwoju, proponowane jest zrealizowanie poniższych, dodatkowych, założeń:

- implementacja mechanizmu umożliwiającego tworzenie raportów (np. w postaci plików HTML) z przebiegu wykonanych testów;
- wzbogacenie o mechanizmy wspierające prowadzenie testów wydajnościowych — zrealizowane to mogłoby być poprzez implementacje zestawu obiektów `Runner` (który umożliwiałby prowadzenie pomiaru wydajności) i `Classifier` (porównujących zmierzone wyniki z oczekiwanymi);
- wskazane byłoby też zintegrowanie z innymi bibliotekami logującymi, zwłaszcza z popularnym log4j, dającym dużo większe możliwości w porównaniu z Java Logging API;
- przydatną funkcjonalnością byłaby z pewnością możliwość integracji z popularnymi środowiskami programistycznymi jak Eclipse, czy NetBeans;
- stworzenie funkcjonalności umożliwiającej tworzenie raportów z przebiegu testów;
- integracja z innymi frameworkami — możliwość uruchamiania przypadków testowych TestNG i JUnit z poziomu frameworku.

6. Podsumowanie

Zapewnienie jakości jest dziś niezbędnym elementem przy produkcji oprogramowania. W związku z tym, także automatyzacja testów odgrywa dziś coraz większą rolę. Zmniejszanie niezbędnego do tego nakładu pracy jest z pewnością celem zespołów osób zarządzających testami. Ale jest to też wyzwaniem dla twórców oprogramowania wspierającego tego rodzaju prace.

Framework realizujący wzorzec Inverse of Control wraz z Depedency Injection pozwala programistom i testerom skupiać się na tworzeniu testów dla realizowanych projektów, bez potrzeby budowania od podstaw środowiska w którym będą one wykonywane, lecz raczej tworzyć je zgodnie z własnymi potrzebami korzystając z gotowych elementów wchodzących w jego skład.

Służyć ma do tego prototyp stworzony wraz z niniejszą pracą. Jest on dobrą, choć nieco uproszczoną, podstawą do budowania środowiska, w którym wykonywane będą testy. Zawiera on najbardziej podstawowe komponenty takiego środowiska i umożliwia łatwe tworzenie, lub dodawanie już zaimplementowanych (jako rozszerzeń), kolejnych jego składowych. Powinien znaleźć zastosowanie zarówno przy prostych testach jednostkowych (gdzie istotna jest prostota i szybkość tworzenia prostych testów) jak i złożonych scenariuszach testów systemowych (gdzie liczą się jego cechy jak możliwość rozszerzania jego funkcjonalności, czy ponowne wykorzystanie kodu).

Bibliografia

[1] Douglas Isbell, Mary Hardin, Joan Underwood, Mars Climate Orbiter Mission Overview, *Mars Climate Orbiter Team Finds Likely Causes Of Loss* , 1999. http://marsprogram.jpl.nasa.gov/msp98/news/mco990930.html

[2] David Gelperin, Bill Hetzel, Communications Of the ACM, *Growth Of Software Testing*, June 1988, Volume 31, Number 6

[3] Christian Bucanac, Quality management course, *The V-Model*. http://www.bucanac.com/documents/The_V-Model.pdf

[4] Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler, *The art of software testing*, Second Edition, Test-Case Design; page 43; Word Association, 2004

[5] Michael Singer Dobson, *The triple constraints in project management*, The triple constraints: Time, Cost, and Performance; page 7; Management Concepts, 2004

[6] Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler, *The art of software testing*, Second Edition, The Psychology of Testing, page 5, Word Association, 2004

[7] Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler, *The art of software testing*, Second Edition, The Economics of Testing, page 9, Word Association, 2004

[8] Marnie L. Hutcheson, Software *Testing Fundamentals: Methods and Metrics*, Maintaining Quality Assurance in Today's Software Testing Environment, John Wiley & Sons, 2003

[9] Tony Morris; *JTiger 2.1 User Documentation*; 2006; http://jtiger.org/userdoc/userdoc.html

[10] Cédric Beust, Hani Suleiman; *Next generation Java testing: TestNG and advanced concepts*; Addison-Wesley; 2008

[11] Martin Fowler; *Inversion of Control Containers and the Dependency Injection pattern*; 2004. http://www.martinfowler.com/articles/injection.html

Spis ilustracji

Spis listingów

Załącznik A: Słownik terminów

Opracowanie własne na podstawie słownika stworzonego przez International Software Testing Qualifications Board, w przekładzie wykonanym przez Stowarzyszenie Jakości Systemów Informatycznych (http://www.istqb.org/ ; http://www.sjsi.org/).

Analiza statyczna (Static analysis) **—** analiza programu bez jego wykonywania.

Awaria (Failure) **—** odmienny od spodziewanych wynik działania oprogramowania.

Błąd, Usterka, Defekt (Bug, Fault, Defect) **—** efekt popełnionej pomyłki w czasie projektowania bądź implementacji systemu informatycznego, może on doprowadzić do powstania awarii.

Integracja (Integration) **—** czynność łączenia modułów oprogramowania w całość.

Komparator (Comparator) **—** narzędzie ułatwiające porównywanie wyników zebranych w czasie testów z oczekiwanymi.

Luźne wiązanie (Loose Coupling) **—** dążenie do tworzenia modułów oprogramowania zależnych od siebie tylko z pośrednictwem jasno określonych interfejsów a niezależnych od ich implementacji.

Moduł, Komponent (Module, Component) **—** jednostka wchodząca w skład systemu określona osobną specyfikacją.

Odwrócenie sterowania (Inverse of Control) **—** wzorzec projektowy polegający na oddaniu kontroli nad wykonaniem programu centralnemu modułowi tego oprogramowania.

Pokrycie (Coverage) **—** jest miarą mówiącą o tym w jakim stopniu testy jednostkowe w czasie swojego działania wymusiły wykonanie całego kodu testowanego oprogramowania.

Pomyłka (Error) **—** (niezamierzone) działanie człowieka skutkujące wprowadzeniem błędu do kodu programu.

Retesty (Re-testing) **—** ponowne wykonanie testów po naprawieniu wykrytych błędów, w celu sprawdzenia czy zostały one usunięte skutecznie.

TestCase — definicja ciągu akcji mających na celu sprawdzenie poprawności działania systemu (lub jego elementu) wraz ze zbiorem ich oczekiwanych rezultatów.

Testowanie (Testing) **—** proces sprawdzający czy oprogramowanie spełnia wymagania, jednocześnie służąc wykrywaniu błędów w nim zawartych.

Testowanie integracyjne (Integration testing) **—** testy znajdujące błędy w interfejsach łączonych moduły.

Testowanie obciążeniowe (Volume testing) **—** testy systemu przetwarzającego duże ilości danych.

Testowanie pielęgnacyjne (Maintance testing) **—** testy sprawdzające wpływ wprowadzania zmian (lub zmian środowiska otaczającego) na wdrożony system.

Testowanie regresywne (Regression testing) **—** ponowne testy systemu po wprowadzonych zmianach, wykrywające błędy ujawniające się po jego modyfikacji.

Testowanie systemowe (System testing) **—** testy kompletnego systemu, potwierdzające jego zgodność z założeniami.

Testowanie wydajnościowe (Performance testing) **—** testy sprawdzające czy system spełnia założenia dotyczące wydajności.

TestSuite — zbiór definicji TestCase.

Testy akceptacyjne (Acceptance testing) **—** formalne testy umożliwiające odbiorcy systemu ustalenie czy system może zostać odebrany (czy spełnia wszystkie wymagania).

Testy dymne (Smoke testing) **—** przypadki testowe pokrywające najistotniejszą funkcjonalność systemu.

Testy jednostkowe (Unit testing) **—** testowanie pojedynczych modułów oprogramowania, lub ich części składowych.

Testy potwierdzające (Confidence test) **—** rodzaj testów dymnych, mający na celu sprawdzenie czy usunięto wcześniej wykryte błędy.

Testy wstępne (Intake test) **—** rodzaj testów dymnych, mający na celu sprawdzenie czy system jest gotowy do testów.

Wstrzykiwanie zależności (Dependency Injection) — wzorzec projektowy polegający na automatycznym wstawianiu do obiektu referencji do komponentów zależnych.

Wynik oczekiwany (Expected outcome) — zgodne z zachozachowanie modułu w określonych warunkach.

Wysoka zwartość (High Cohesion) — dążenie do tworzenia modułów wysoce wyspecjalizowanych (zorientowanych na pojedynczy problem).

Wyszukiwanie zależności (Dependency Lookup) — wzorzec projektowy polegający na pobieraniu przez obiekt referencji do komponentów zależnych z repozytorium Dependency Lookup.