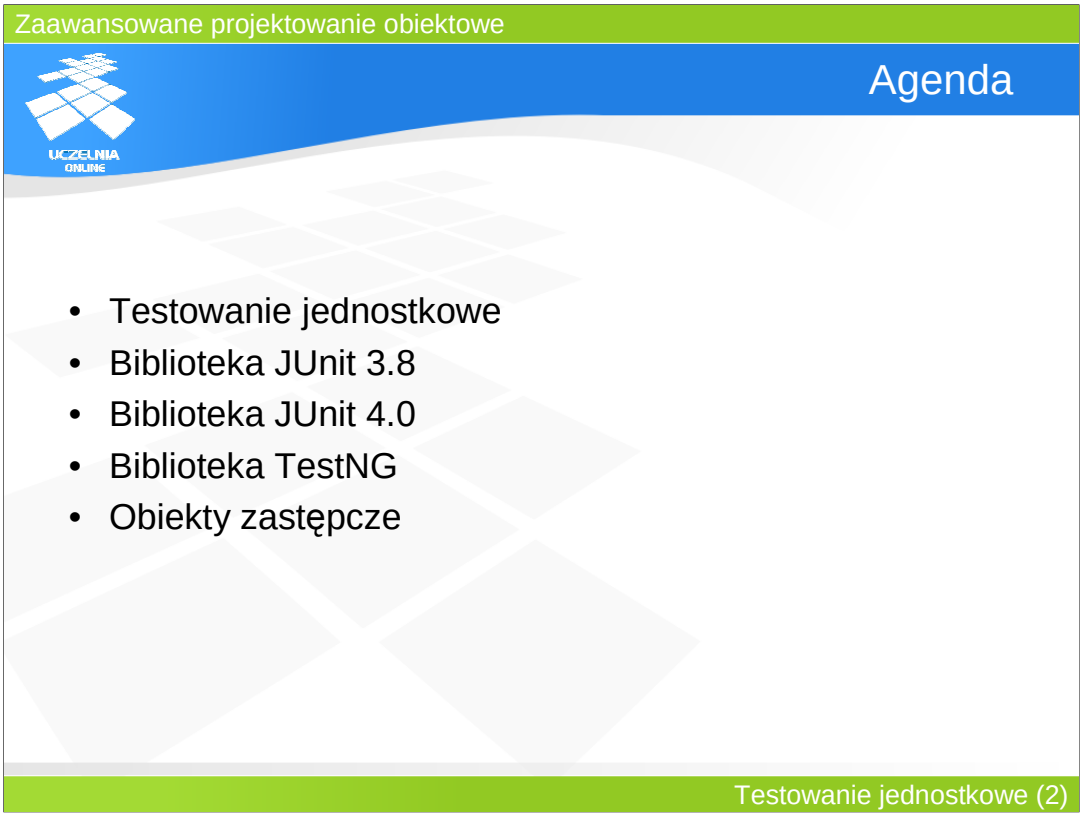

Zaawansowane projektowanie obiektowe
Testowanie jednostkowe
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE



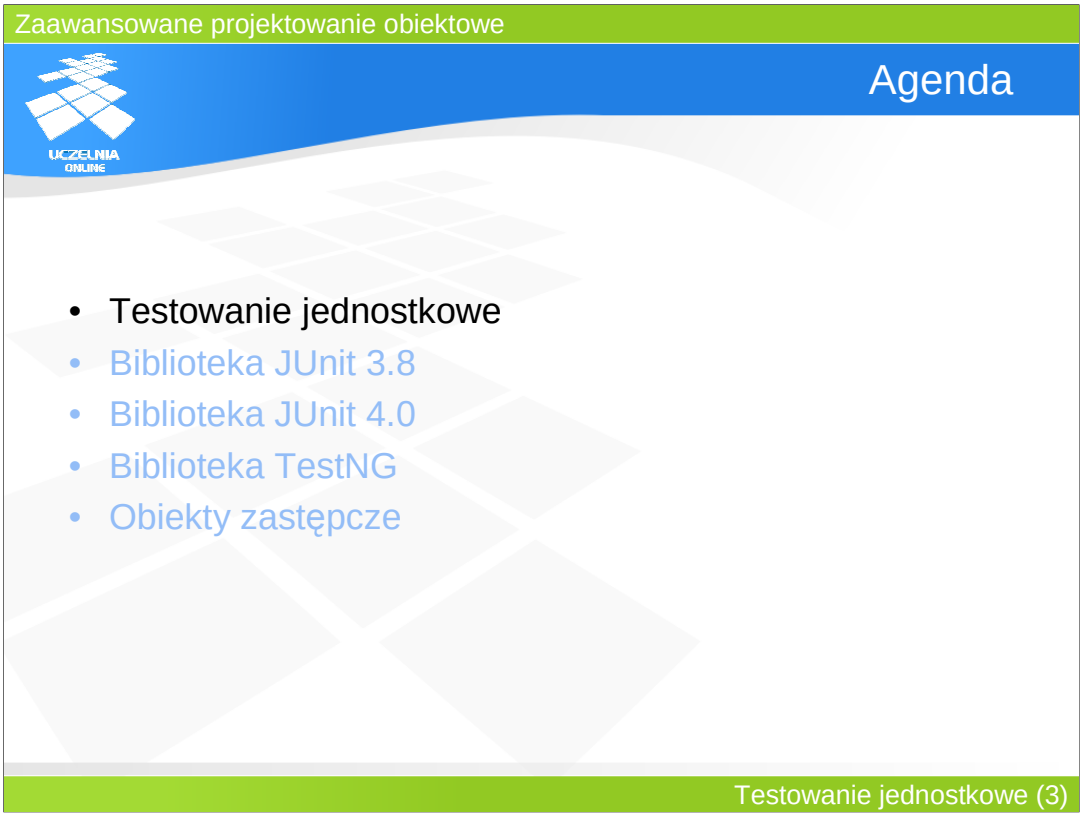
Podczas wykładu zostaną zaprezentowane następujące zagadnienia

•przypomniane będą idea i zasady związane z testowaniem jednostkowym

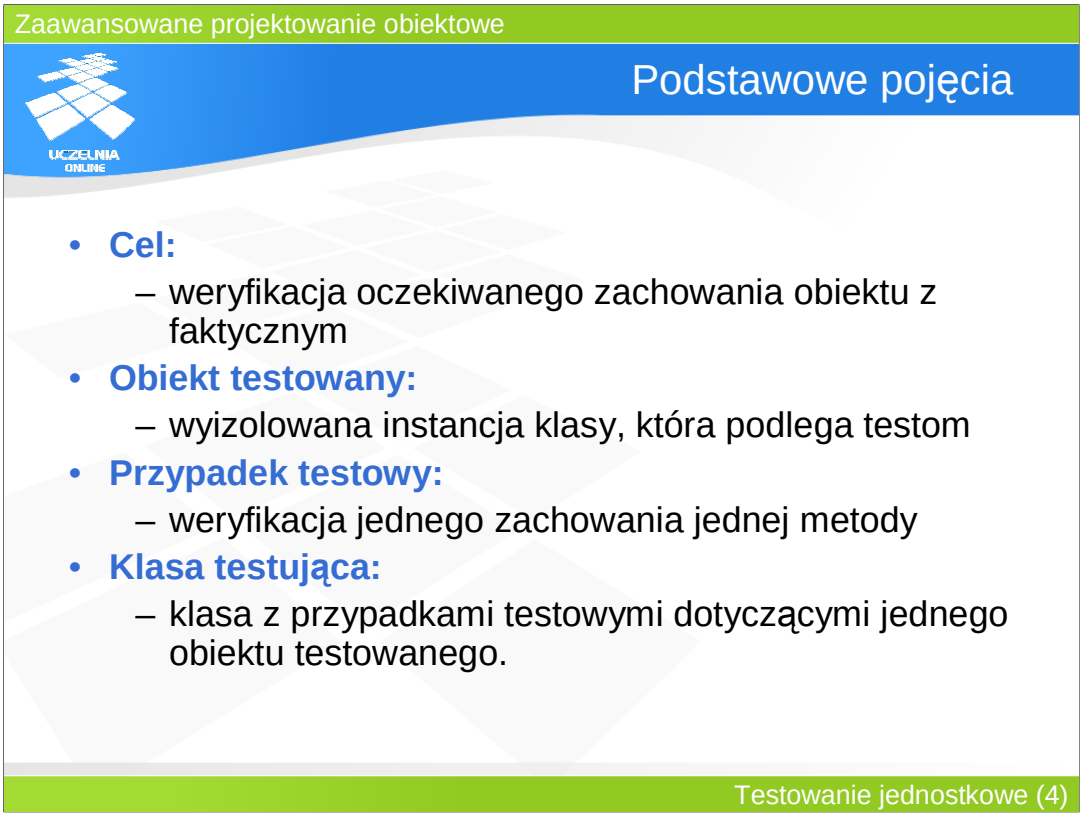
•przedstawiona zostanie klasyczna biblioteka JUnit 3.8, która zapoczątkowała wzrost znaczenia testowania jednostkowego w inżynierii oprogramowania

•zaprezentowane będą dwie biblioteki nowej generacji (JUnit 4.0 i TestNG), oparte na możliwościach oferowanych przez języka Java 5.0

•wprowadzone będzie pojęcie obiektu zastępczego (imitacji)



Pierwszą częścią jest przypomnienie zadań stawianych testowaniu jednostkowemu.

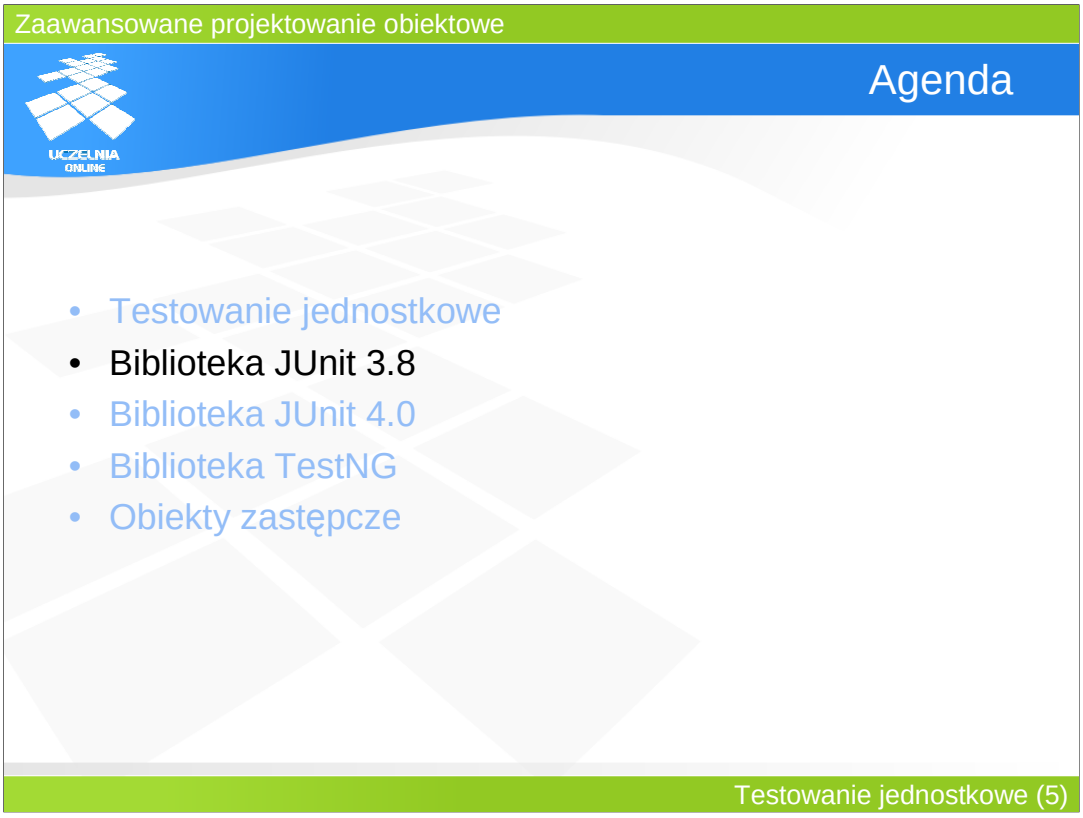


Testowanie jednostkowe jest jedną z technik weryfikacji poprawności działania programu. Dotyczy jednak bardzo niskiego poziomu – pojedynczej klasy i metody. Celem jego jest sprawdzenie, czy rzeczywisty efekt działania metody wywołanej w określonym kontekście i z określonymi parametrami jest taki, jak oczekiwano.

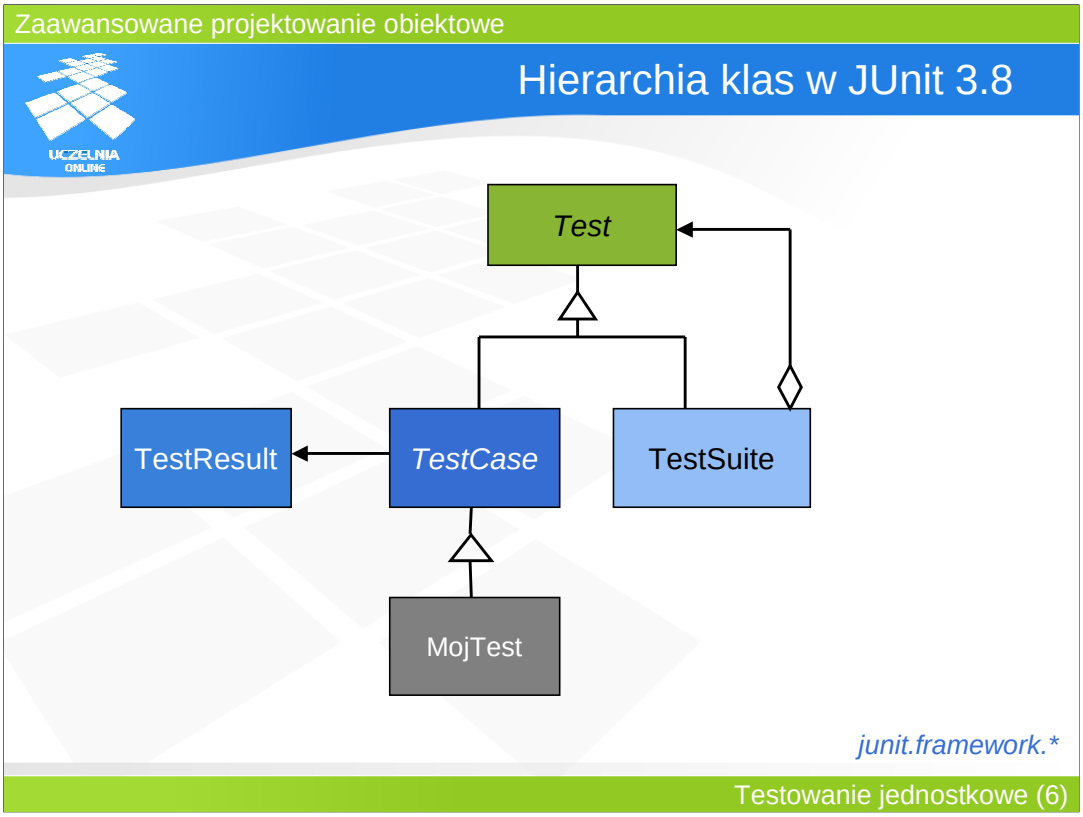
Nazwa 'testowanie jednostkowe' wywodzi się z faktu, że obiektem testowanym jest pojedyncza, wyizolowana ze środowiska instancja klasy. Zatem nie jest uruchamiany cały tworzony system, ale właśnie pojedynczy obiekt.

Przypadkiem testowym w testowaniu jednostkowym jest sprawdzenie, czy metoda w określonych warunkach zachowuje się poprawnie. Zatem dla każdej metody o nietrywialnym zachowaniu (zwykle będzie to metoda publiczna) istnieje jeden lub więcej przypadków testowych, po jednym dla każdego rodzaju zachowania testowanej metody.

Przypadki testowe dotyczące jednego obiektu testowanego są zebrane wewnątrz jednej klasy testującej. W ten sposób jednej klasie testowanej odpowiada jedna klasa testująca.



Ilustracją tych zasad jest opis biblioteki JUnit 3.8, która w znacznym stopniu przyczyniła się do wzrostu popularności testowania jednostkowego.

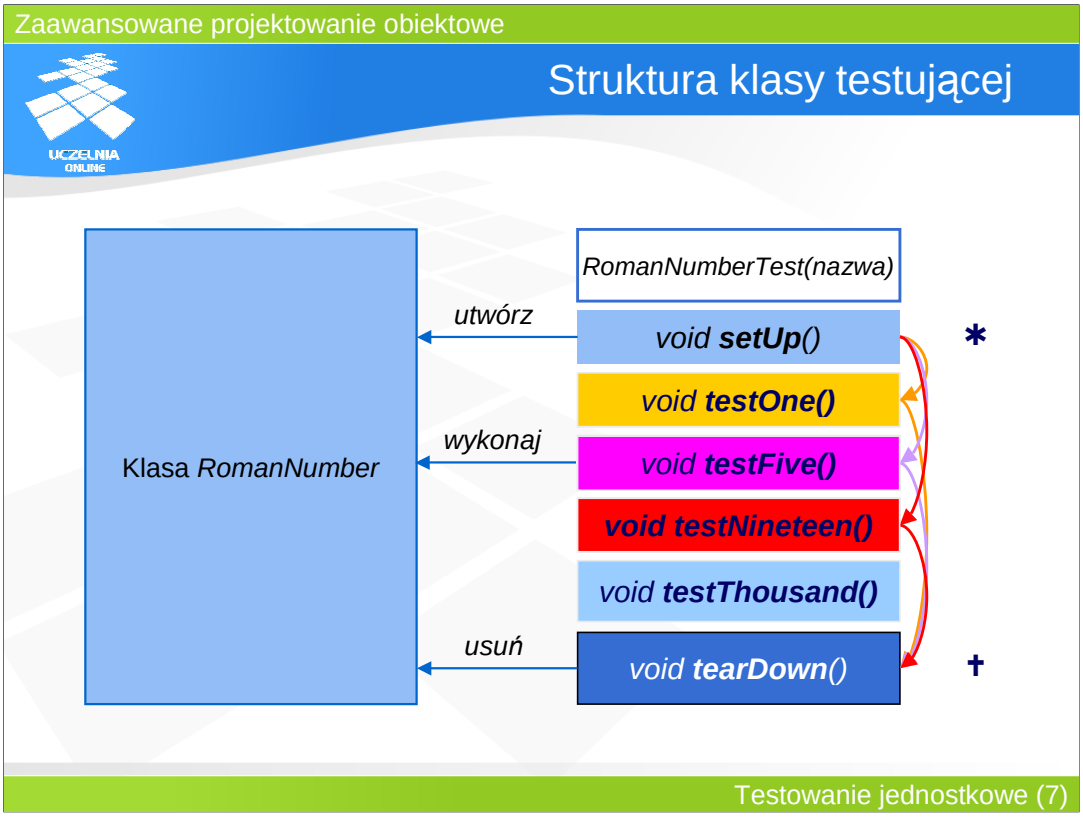


W skład biblioteki JUnit 3.8 wchodzi kilkadziesiąt klas, jednak kilka z nich ma znaczenie podstawowe.

•Klasą bazową dla wszystkich przypadków testowych jest TestCase, z uwagi na nieco nieszczęśliwą nazwę często mylony z przypadkiem testowym (tymczasem TestCase jest klasą testującą, która zawiera przypadki testowe zapisane w postaci metod; dlatego przypadek testowy to pojedyncza metoda, a nie klasa). Klasa ta udostępnia podstawowe funkcje pomocne w testowaniu, m.in. domyślne asercje.

•TestCase jest wraz z TestSuite implementacją interfejsu Test. Razem klasy te tworzą strukturę drzewiastą, w której wszystkie węzły poza liśćmi drzewa są reprezentowane przez TestSuite, a liście – przez klasę TestCase. Jest to w rzeczywistości implementacja wzorca projektowego Composite, który pozwala zarządzać całą strukturą w sposób jednorodny. Dzięki temu możliwe jest tworzenie suit (zestawów testów) i uruchamianie ich w identyczny sposób jak pojedyncze klasy TestCase.

•TestResult jest klasą przechowującą wyniki wykonanych przypadków testowych. Jest ona tworzona i wypełniana danymi przez kolejne klasy TestCase, które są wykonywane. Jest to także implementacja wzorca o nazwie Collecting Parameter



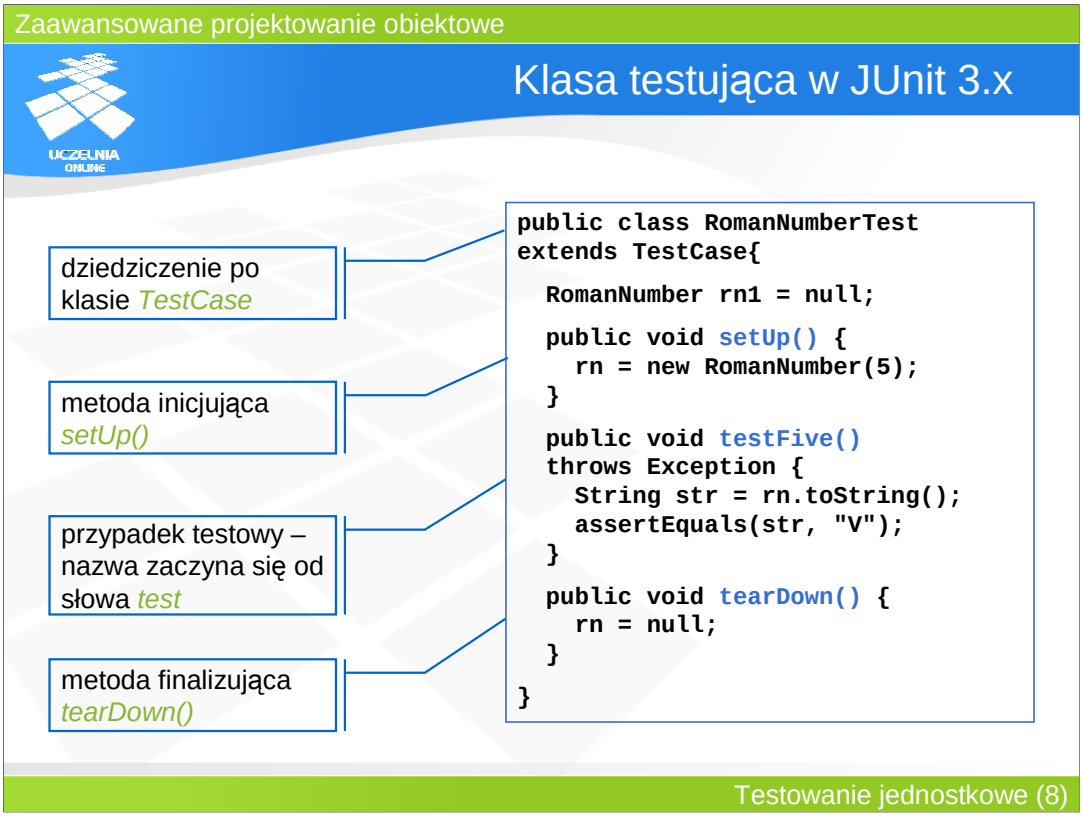
Klasa testująca w JUnit 3.x ma określoną strukturę:

•konstruktor przyjmuje parametr będący nazwą przypadku testowego; zwykle nie ma on żadnego znaczenia i można go pominąć.

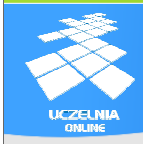
•metoda *setUp()* służy do inicjacji każdego przypadku testowego; jest wykonywana przed każdym z nich i zwykle służy do stworzenia obiektu testowanego i innych obiektów wymaganych do uruchomienia go. Metoda ta jest dziedziczona po klasie TestCase

•przypadki testowe są implementowane w pojedynczych metod zawartych w klasie testującej. Muszą one spełniać konwencję przyjętą w JUnit 3.x: nie przyjmować parametrów, nie zwracać wartości oraz mieć nazwę zaczynającą się od przedrostka 'test' (inne metody nie są rozpoznawane jako przypadki testowe). Wykonanie każdego przypadku testowego jest poprzedzone wywołaniem metody *setUp()* i zakończone wywołaniem metody *tearDown()*

•metoda *tearDown()* jest komplementarna w stosunku do metody *setUp()*, tzn. służy do przywrócenia stanu sprzed wykonania testu. Jeżeli *setUp()* jedynie tworzyła obiekty, wówczas implementacja *tearDown()* jest zbędna; ma ona znaczenie tylko jeżeli przypadek testowy zaalokował zasób, który powinien być zwolniony np. połączenie sieciowe lub do bazy danych etc. Wówczas metoda ta powinna ten zasób zwolnić



Powyżej przedstawiono klasę testującą dla klasy RomanNumber, reprezentującej liczby rzymskie. W klasie RomanNumberTest znajduje się pole przechowujące referencję do obiektu testowanego, które jest inicjowane w metodzie *setUp()*. Przypadek testowy o nazwie *testFive()* sprawdza, czy konwersja liczby arabskiej 5 na rzymską V jest dokonywana właściwie. Jakakolwiek niezgodność jest raportowana przez wyjątek. Metoda *tearDown()* została zaimplementowana jedynie dla porządku, ponieważ jej zadanie – usunięcie obiektu testowanego – i tak wykona mechanizm *garbage collector.*



Dostępne asercje w klasie *TestCase*

- **asercje porównań**
  - **assertEquals**([komunikat], oczekiwany, faktyczny)
- **asercje tożsamości**
  - **assertSame**([komunikat], oczekiwany, faktyczny)
  - **assertNotSame**([komunikat], oczekiwany, faktyczny)
- **asercje referencji *null***
  - **assertNull**([komunikat], referencja)
  - **assertNotNull**([komunikat], referencja)
- **asercje logiczne**
  - **assertTrue**([komunikat], warunek)
  - **assertFalse**([komunikat], warunek)
- **bezwarunkowe niepowodzenie**
  - **fail**([komunikat])

Klasa TestCase posiada kilka domyślnych asercji, pozwalających łatwo weryfikować relacje pomiędzy wartością oczekiwaną a wartością rzeczywistą, obliczoną przez metodę w obiekcie testowanym. Asercje są metodami przeciążonymi dla wielu typów danych języka Java, tak aby możliwie najczytelniej prezentować różnice między porównywanymi wartościami. Ponadto każda z metod ma wersję przyjmującą jako pierwszy parametr komunikat wyświetlany w momencie, gdy asercja okaże się nieprawdziwa.

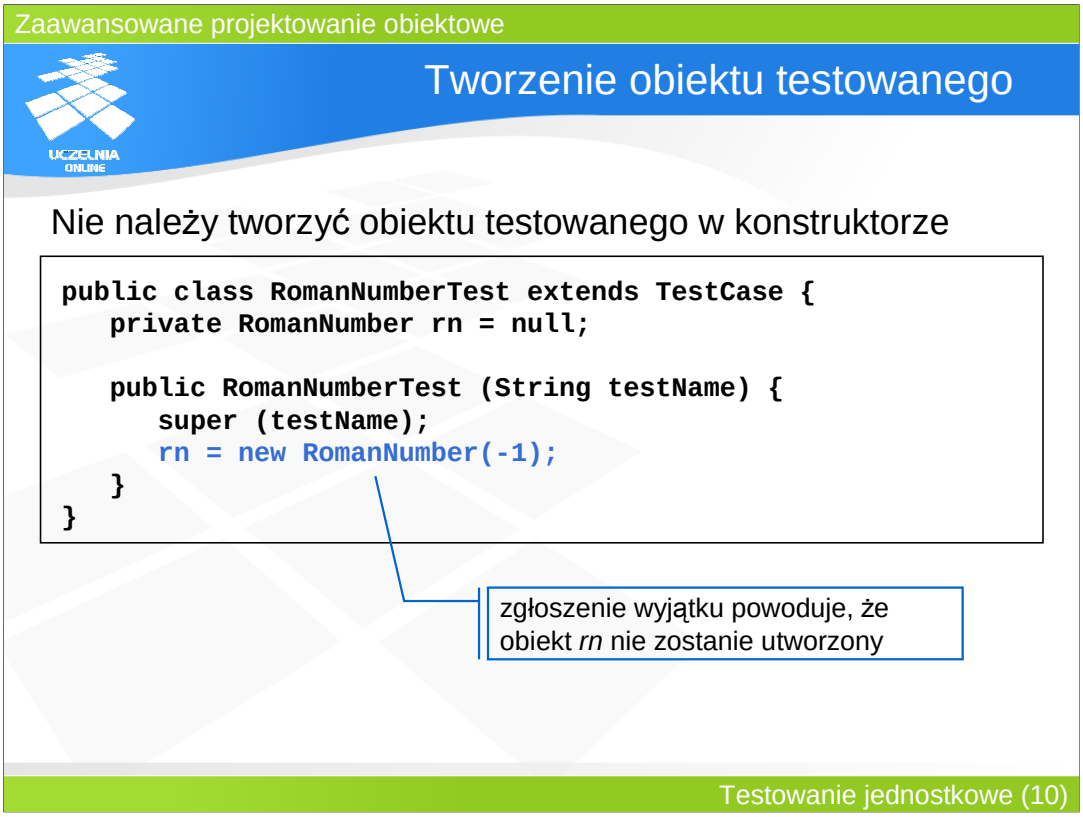
•*assertEquals()* sprawdza, czy wartości przekazane jako parametry są równe. W przypadku typów prymitywnych porównanie jest wykonywane za pomocą operatora ==, natomiast dla typów obiektowych wywoływana jest metoda *equals()*

•*assertSame()* (i analogiczna *assertNotSame()*) sprawdza identyczność obu parametrów, czyli we wszystkich przypadkach stosuje operator ==

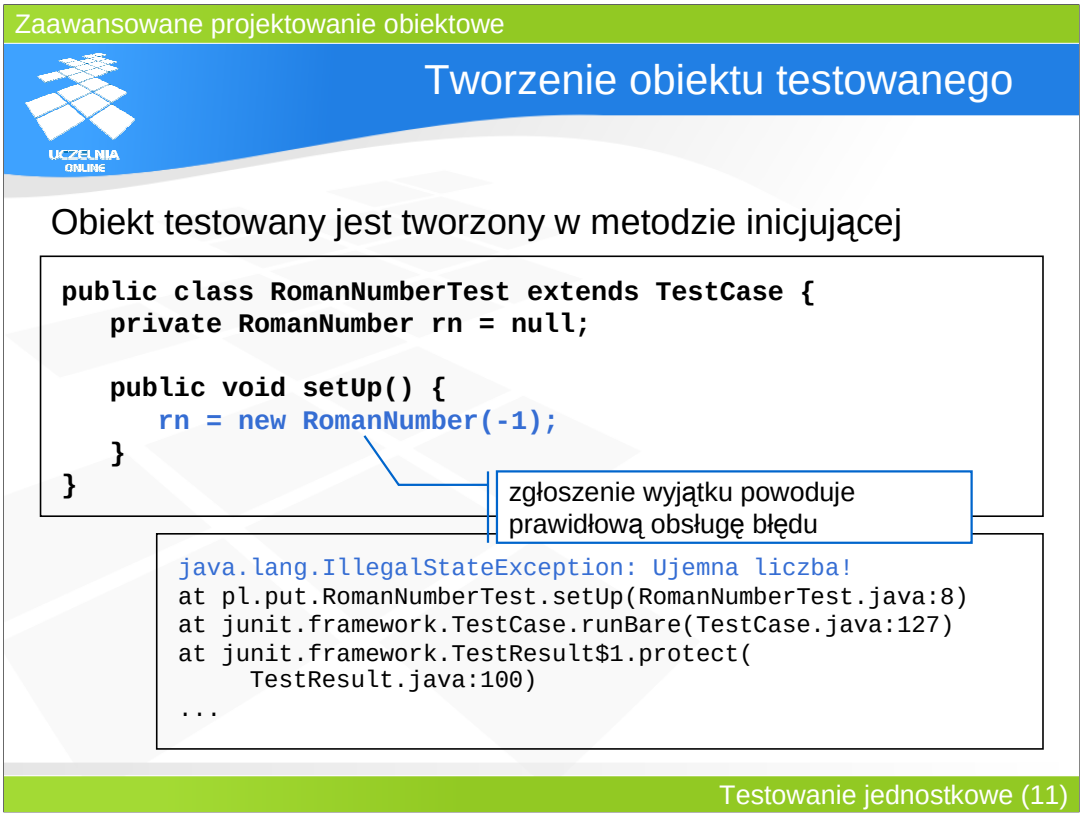
•*assertNull()* i *assertNotNull()* sprawdzają, czy podana referencja wskazuje na obiekt, czy też nie

•*assertTrue()* i *assertFalse()* badają prawdziwość podanych warunków

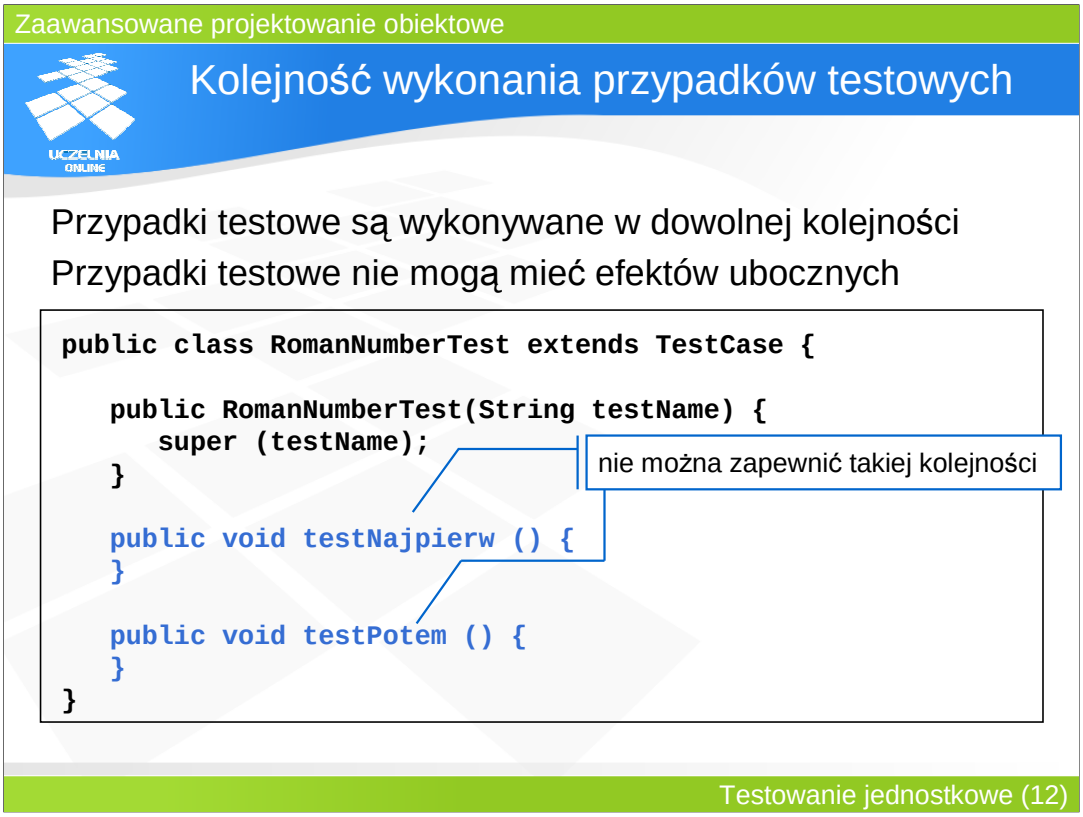
•*fail()* jest bezwarunkowym zgłoszeniem nieprawdziwości asercji. Metoda ta przydaje się w szczególnych sytuacjach, gdy określona gałąź sterowania nigdy nie powinna być wykonana



Bardzo ważną zasadą dotyczącą tworzenia testów jednostkowych jest niemal całkowite pominięcie konstruktora klasy testującej. Konstruktor jest elementem języka Java i zgodnie ze specyfikacją, jakikolwiek wyjątek zgłoszony w konstruktorze przerywa proces tworzenia obiektu. Aby uniknąć sytuacji, w której niemożność utworzenia np. obiektu testowanego spowoduje błąd konstrukcji obiektu klasy testującej, nie wolno umieszczać takiego kodu w konstruktorze. Błąd wynikający z tego zostanie przechwycony przez środowisko uruchomieniowe JUnit, ale przedstawiony on będzie w sposób niepełny i mylący, bez informacji wskazujących na przyczynę błędu.



Dlatego rolę logicznego konstruktora pełni w przypadku obiektów klas TestCase metoda *setUp(),* i to ona jest prawidłowym miejscem tworzenia instancji obiektu testowanego. W identycznej sytuacji, jaką przedstawiono na poprzednim slajdzie, wyjątek zostanie przechwycony i zaprezentowany we właściwy sposób, z pełną informacją o przyczynie błędu i śladzie wyjątku.



Jednym z założeń testowania jednostkowego jest niezależność poszczególnych przypadków testowych. Jednak ze względów praktycznych często jest konieczne uporządkowanie ich wg pewnej kolejności, co pozwoli wykorzystać efekty wykonania poprzedników w przypadkach testowych po nich następujących. Najprostszym przykładem jest użycie bazy danych: jeżeli każdy przypadek testowy wypełniałby bazę zestawem tych samych (lub niemal tych samych) danych, a następnie usuwał je (za pomocą metody *tearDown()*), wówczas wykonanie zbioru testów trwałoby bardzo długo.

Należy jednak pamiętać, że JUnit został stworzony z założeniem o niezależności przypadków testowych: każdy z nich jest poprzedzony metodą *setUp()* a po nim jest wywoływana metoda *tearDown()*. Dlatego zakładanie określonej kolejności ich wykonania może okazać się fałszywe. Kolejność metod w pliku nie ma żadnego znaczenia dla środowiska uruchomieniowego JUnit, więc poleganie na niej może prowadzić do błędnych wyników wykonania testów.

Kolejność wykonania przypadków testowych

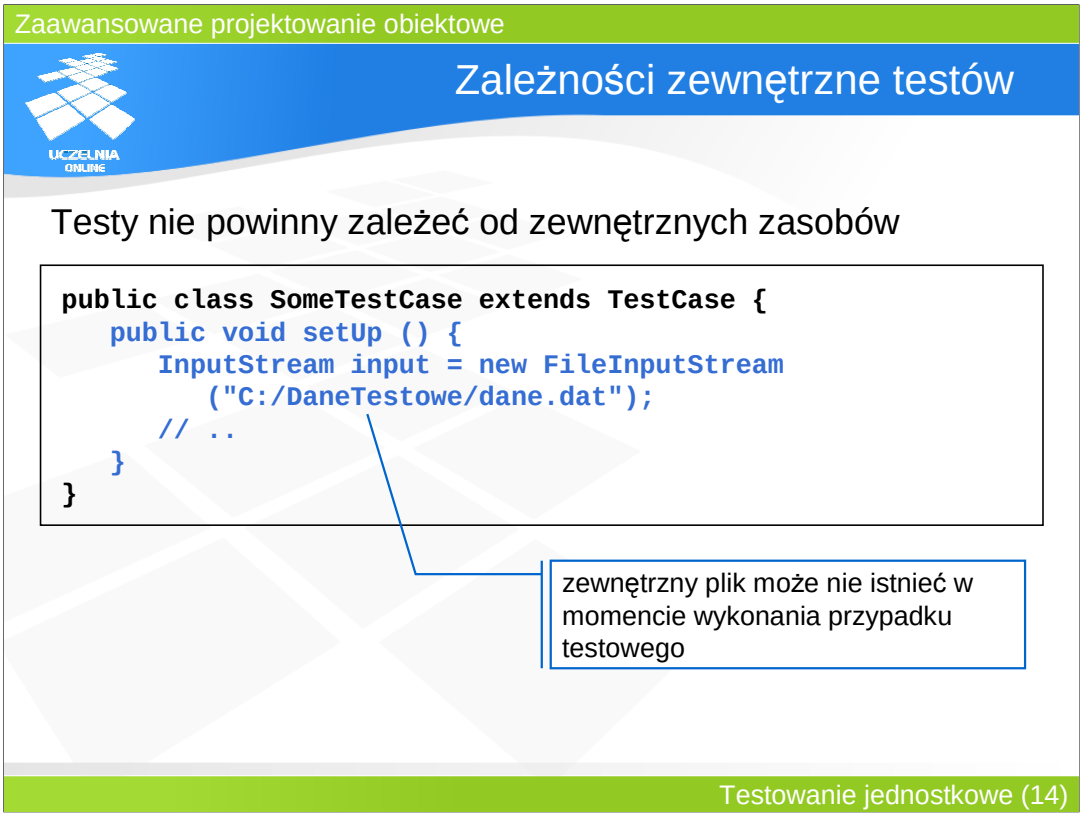
Kolejnością wykonania można sterować poprzez metodę *suite()*

```
public class RomanNumberTest extends TestCase {
   public void testNajpierw() {}
   public void testPotem() {}

   public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(new RomanNumberTest("testNajpierw"));
      suite.addTest(new RomanNumberTest("testPotem"));

      return suite;
   }
}
```

przypadki testowe są wykonywane w kolejności określonej w metodzie *suite*()

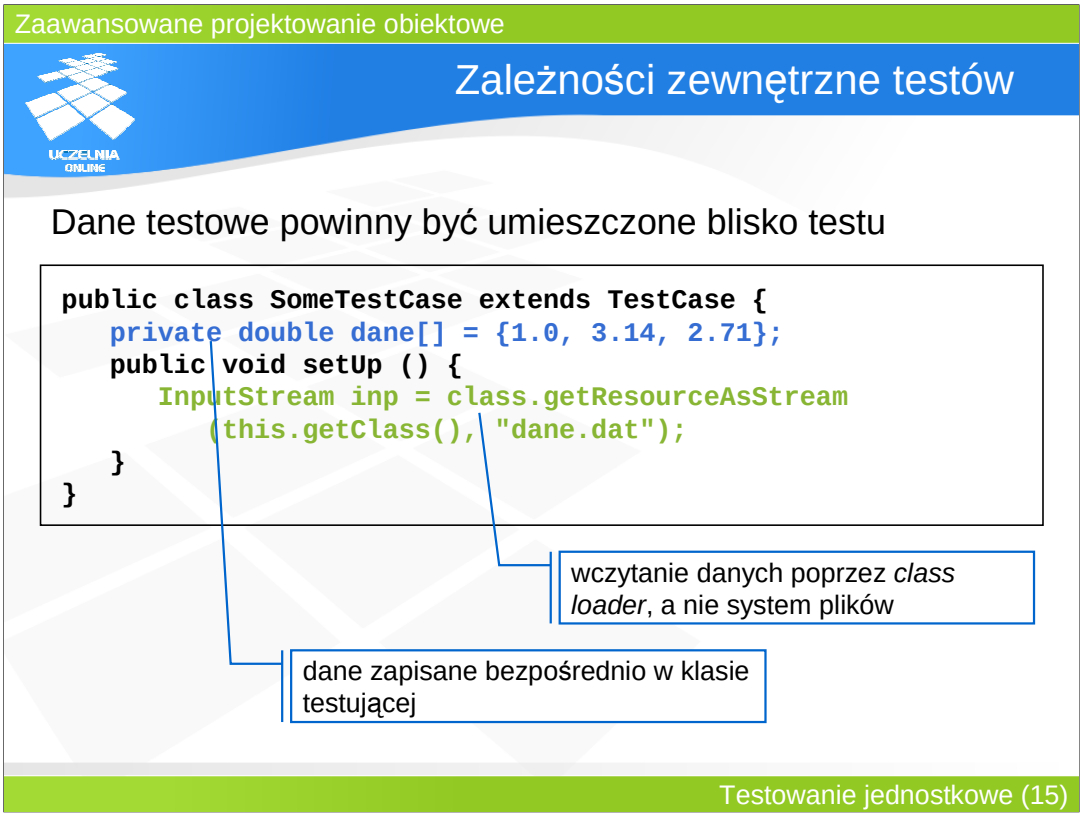
Zdarzają się jednak sytuacje, w których uporządkowanie przypadków testowych jest konieczne (a czasami tak jest – JUnit 3.x posiada kilka braków funkcjonalnych, które zostaną omówione nieco później), nawet za cenę naruszenia zasady ich niezależności od siebie. Wówczas z pomocą może przyjść znajomość wewnętrznej konstrukcji obiektów TestSuite, reprezentujących suity (zestawy testów). Przechowują one uporządkowaną kolekcję przypadków testowych, które są wykonywane w tej właśnie kolejności.

Wystarczy zatem stworzyć w klasie testującej metodę statyczną *suite()*, wykorzystywaną przez środowisko uruchomieniowe JUnit do uruchomienia przypadków testowych, która zdefiniuje kolejność przypadków testowych w suicie. Zostaną one wykonane w takiej właśnie kolejności.

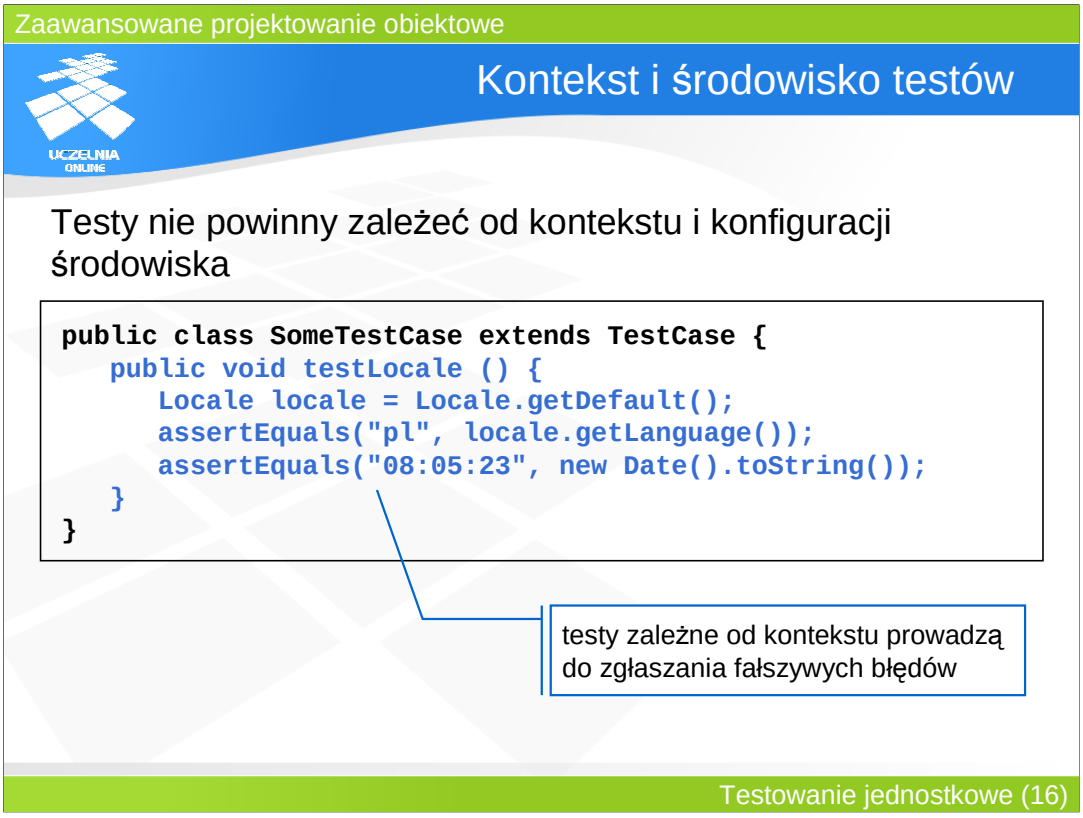


Testy zwykle są umieszczane razem z kodem produkcyjnym w repozytorium, tak aby każdy programista mógł je w każdej chwili wykonać. Dlatego umieszczanie w kodzie testów odwołań do zewnętrznych zasobów, które trzeba adresować w sposób bezwzględny, podając ich lokalizację, jest złym pomysłem. Wiąże on wszystkich użytkowników klasy z jedną strukturą katalogów, co może okazać się dużym utrudnieniem lub nawet uniemożliwić testowanie.

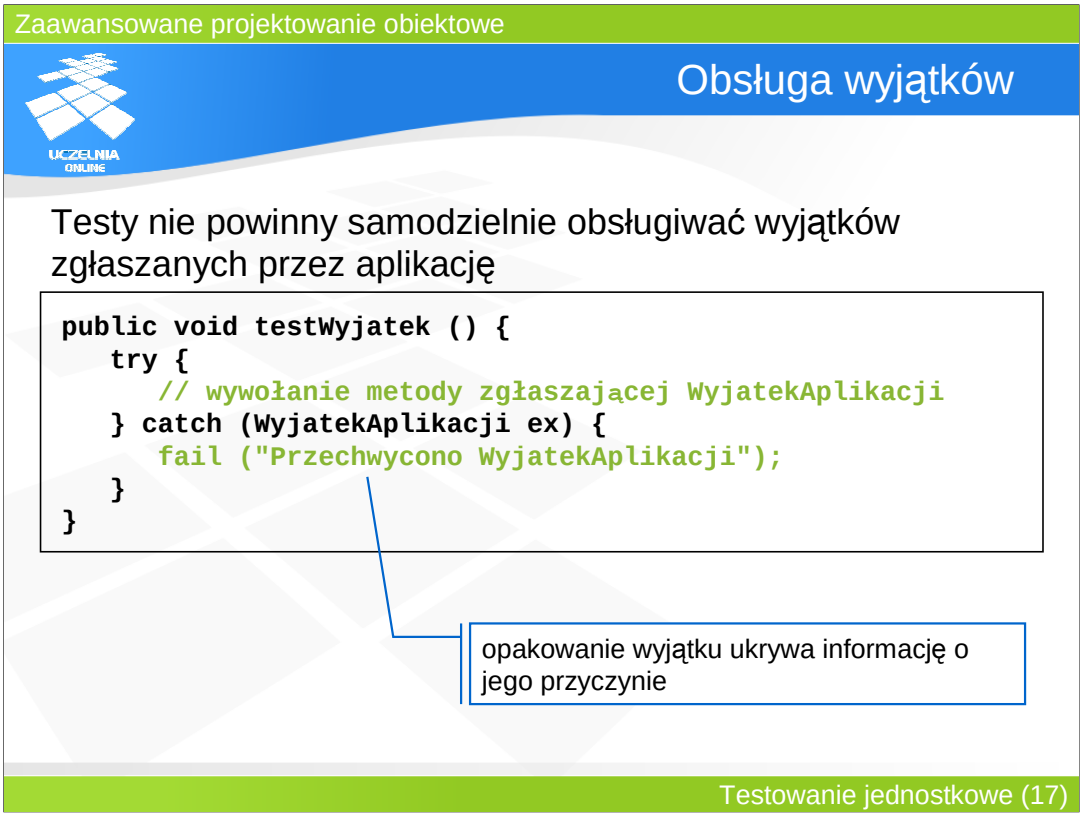
W tym przypadku odczyt pliku "C:/DaneTestowe/dane.dat" może zakończyć się niepowodzeniem mylnie wskazującym na błąd samego przypadku testowego.



Lepszym rozwiązaniem jest umieszczenie danych nawet bezpośrednio w klasie testującej jako pole lub w postaci oddzielnej klasy (np. wewnętrznej). Zwykle nie jest to najlepsze rozwiązanie, ponieważ wymaga rekompilacji programu przy zmianie danych, jednak z drugiej strony daje ono pewność, że dane testowe będą zawsze dostępne. Kiedy takie rozwiązanie jest niemożliwe (np. z uwagi na rozmiar danych lub ich skomplikowaną strukturę), wówczas można np. plik z danymi dołączyć do pliku JAR z klasami i następnie odczytać go korzystając z mechanizmu *class loader*, pomijając system plików. Pozwala to na adresowanie plików wewnątrz pliku JAR bez znajomości jego położenia.

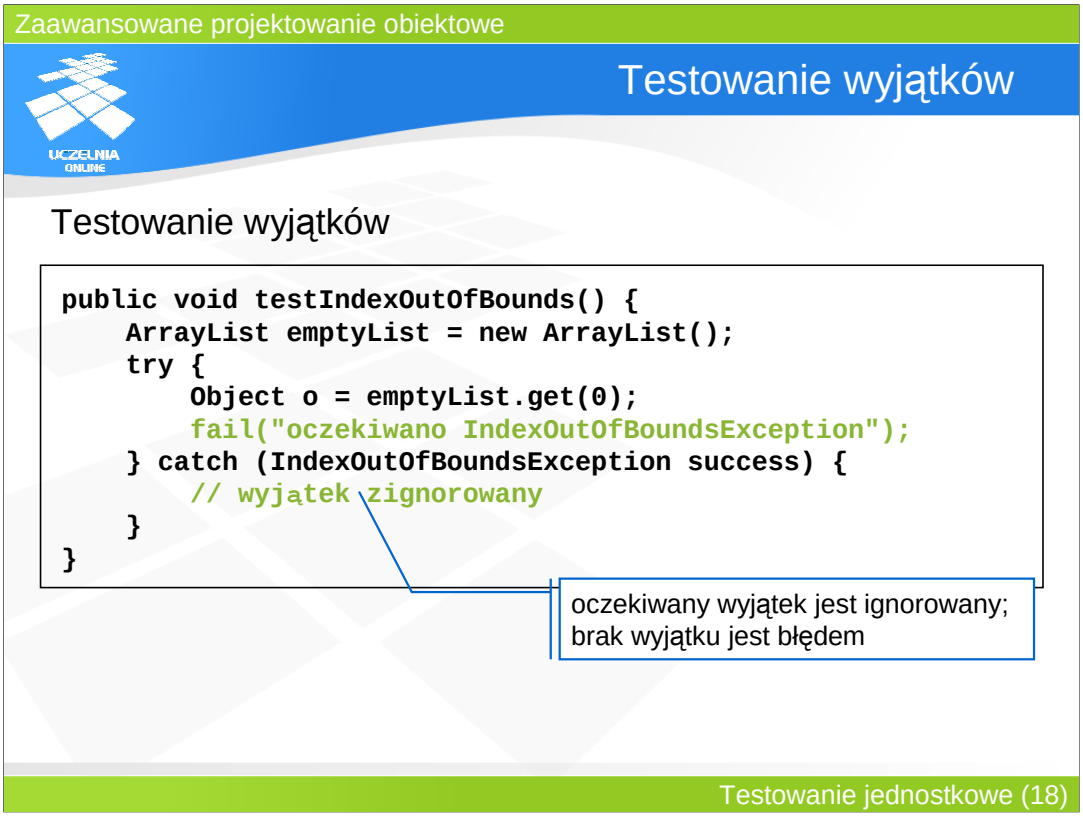


Podobną naturę posiada problem związany z konfiguracją środowiska (w przypadku Javy może to być np. format prezentacji czasu i daty, bieżąca godzina czy też strefa czasowa). Przyjęcie założenia o konkretnym formacie stosowanym domyślnie przez maszynę wirtualną Javy może spowodować, że uruchomienie testów na komputerze o innej konfiguracji będzie powodować trudne do zlokalizowania błędy. Dlatego testy te powinny być wykonywane nie względem ustawień określonych na stałe w treści przypadku testowego, ale względem bieżącej konfiguracji komputera.



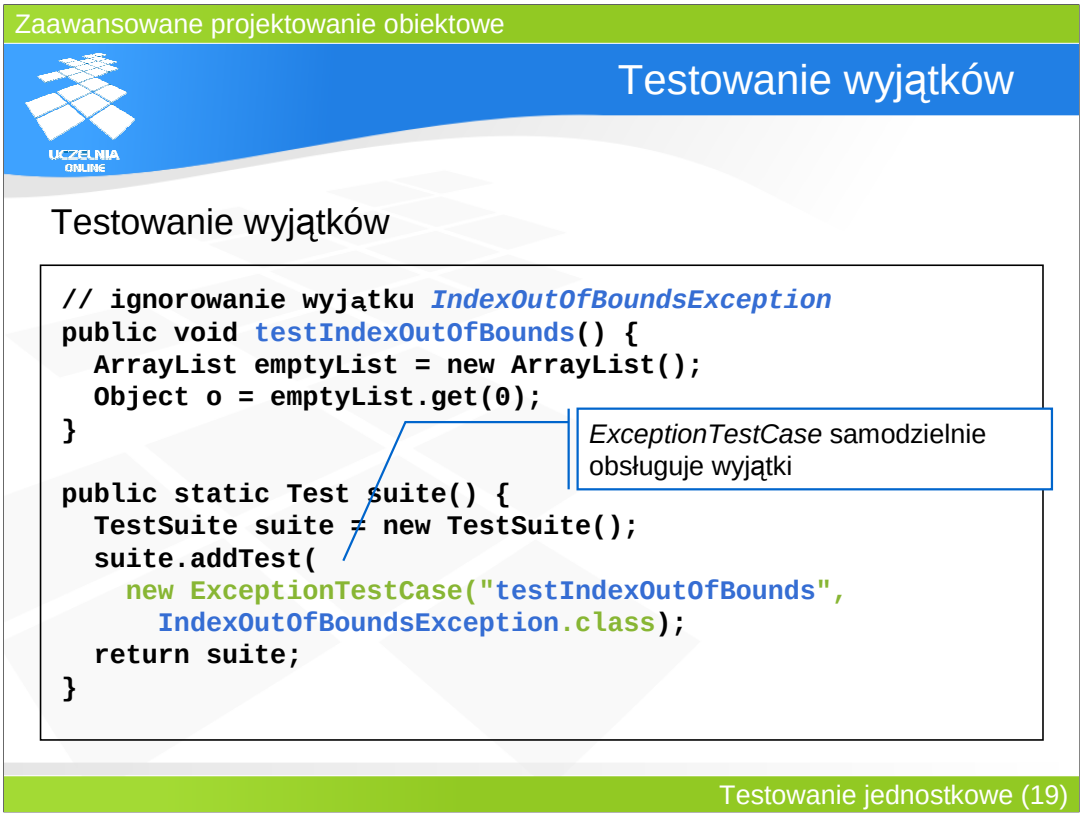
Przypadki testowe, wywołując metodę w obiekcie testowanym, mogą spowodować zgłoszenie wyjątku. Wyjątki sprawdzane wymagają obsługi lub przekazania do metody wywołującej. W niektórych przypadkach zdarza się, że programista próbuje przechwycić taki wyjątek i np. przetłumaczyć go na błąd informujący o niepowodzeniu asercji poprzez wywołanie metody *fail().* Poza rzadkimi przypadkami jest to zupełnie niepotrzebne i błędne: środowisko uruchomieniowe JUnit samodzielnie przechwytuje takie wyjątki i prezentuje ich przyczynę, a przechwycenie ich przez przypadek testowy i przetłumaczenie na inny rodzaj wyjątku (metoda *fail()* zastępuje wówczas jeden wyjątek innym) jedynie utrudnia identyfikację przyczyny błędu.

Prawidłowym rozwiązaniem jest zadeklarowanie zgłaszanych wyjątków w sygnaturze przypadku testowego, co jest całkowicie akceptowalne z punktu widzenia konwencji narzucanych przez JUnit.



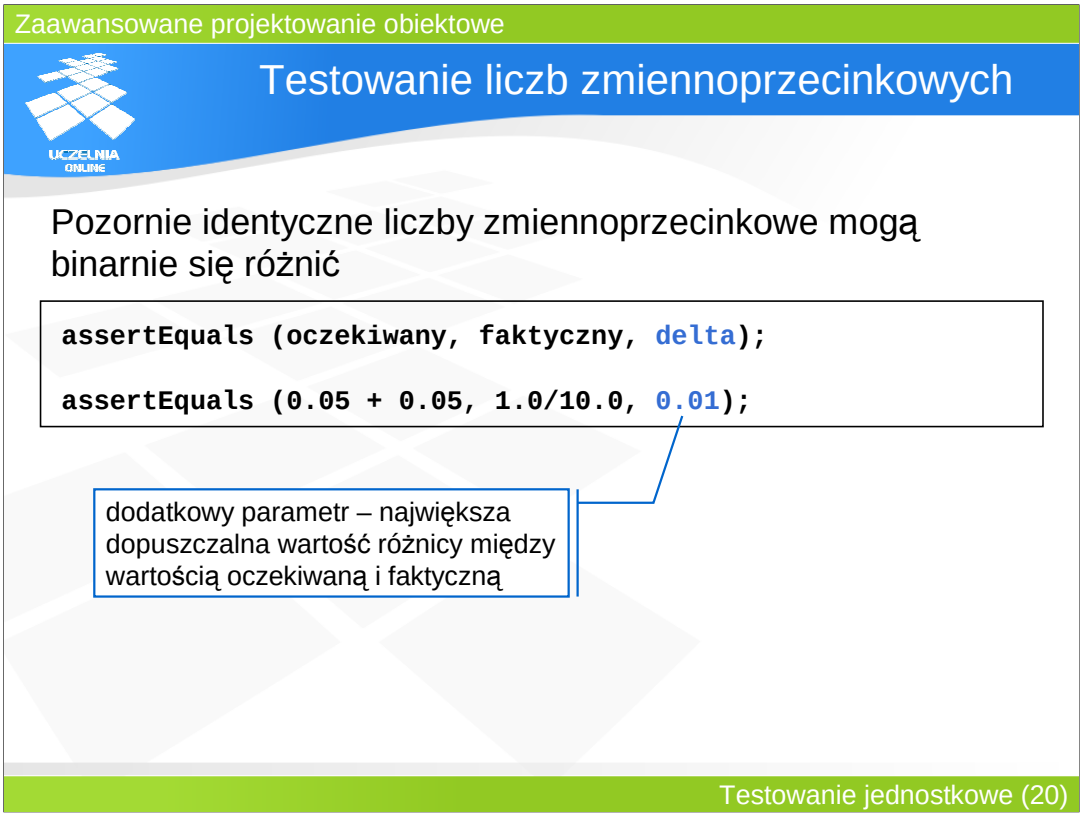
Innym zagadnieniem jest testowanie wyjątków. Wówczas sytuacja ulega odwróceniu: przypadek testowy sprawdza, czy w odpowiedzi na pewne wywołanie metody w obiekcie testowanym zostanie zgłoszony wyjątek. Obecność wyjątku jest pożądana i powinna zostać potraktowana jako sytuacja normalna, natomiast brak wyjątku jest błędem.

W przypadku JUnit 3.8 najprostszym rozwiązaniem jest umieszczenie tej metody wewnątrz właściwie skonfigurowanej klauzuli *try..catch*. Wewnątrz klauzuli *try* wywoływana jest metoda, która powinna zgłosić wyjątek, a zaraz po niej – metoda *fail()*. W klauzuli *catch* przechwytywany jest wyjątek zgłaszany przez testowaną metodę, a następnie jest on ignorowany. W ten sposób, jeżeli wyjątek się pojawi, wówczas zamiast metody *fail()* wykonana zostanie klauzula *catch*, która zignoruje wyjątek. Natomiast jego brak spowoduje zgłoszenie błędu za pomocą metody *fail()*.



Innym rozwiązaniem jest użycie specjalnej klasy ExceptionTestCase, która opakowuje obiekty typu TestCase, a następnie samodzielnie realizuje opisaną wcześniej logikę "odwracania wyjątku". Rozwiązanie to jest elegantsze z punktu widzenia strukturalizacji kodu.

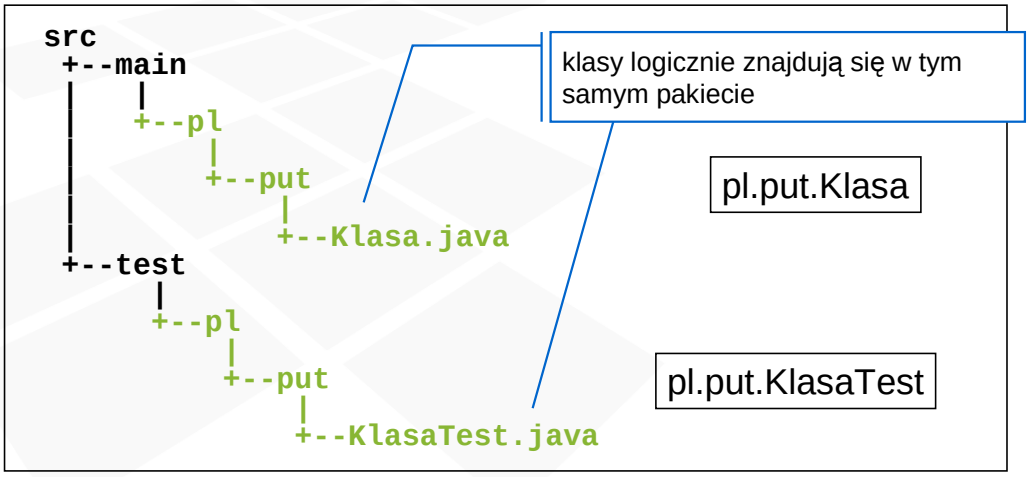
Konstruktor tej klasy przyjmuje dwa argumenty: nazwę przypadku testowego oraz klasę wyjątku, który należy przechwycić.



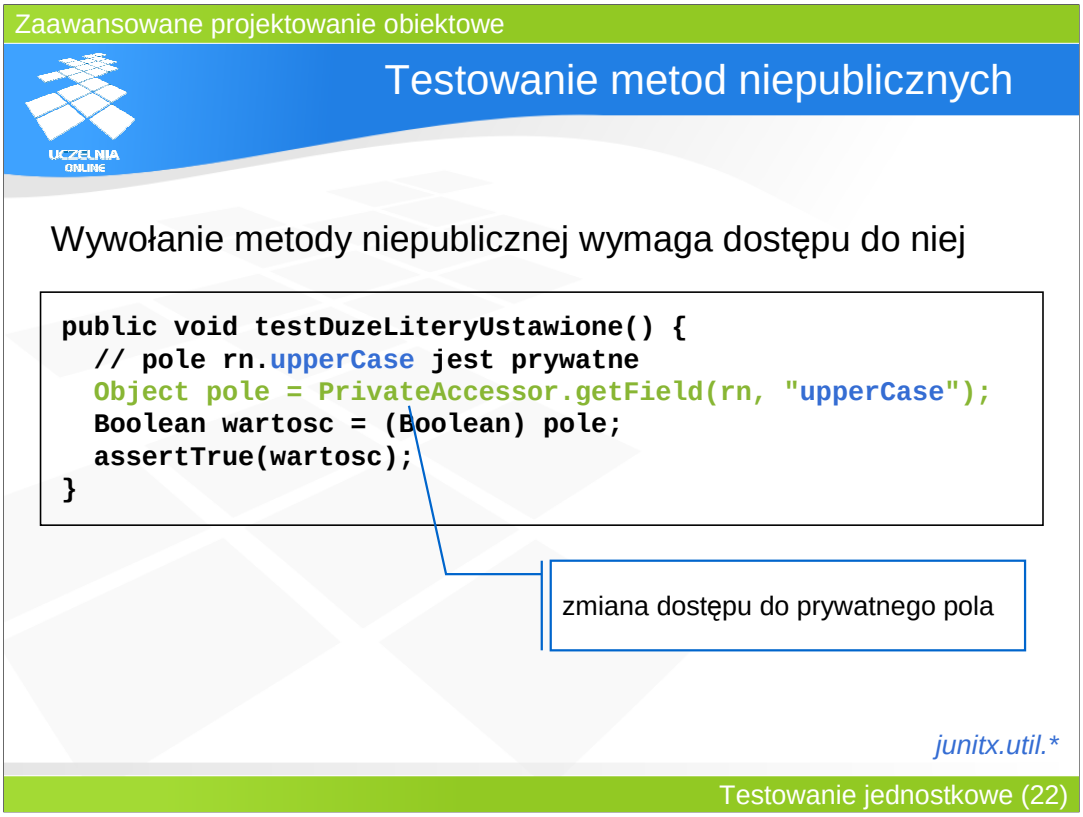
Podczas testowania wartości zmiennoprzecinkowych, warto pamiętać, że wiele z nich jest reprezentowanych w komputerach niedokładnie. Dlatego asercje dla tych typów zostały przeciążone i posiadają dodatkowy parametr, będący największą dopuszczalną różnicą pomiędzy wartościami oczekiwaną i faktyczną. Asercja wówczas nie porównuje tych wartości bezpośrednio, a jedynie sprawdza, czy są one dostatecznie bliskie siebie. Parametr ten powinien przyjmować niewielkie wartości dodatnie, np. 0.01 lub 0.001.

Struktura katalogów z testami

Układ katalogów pozwala oddzielić testy od kodu

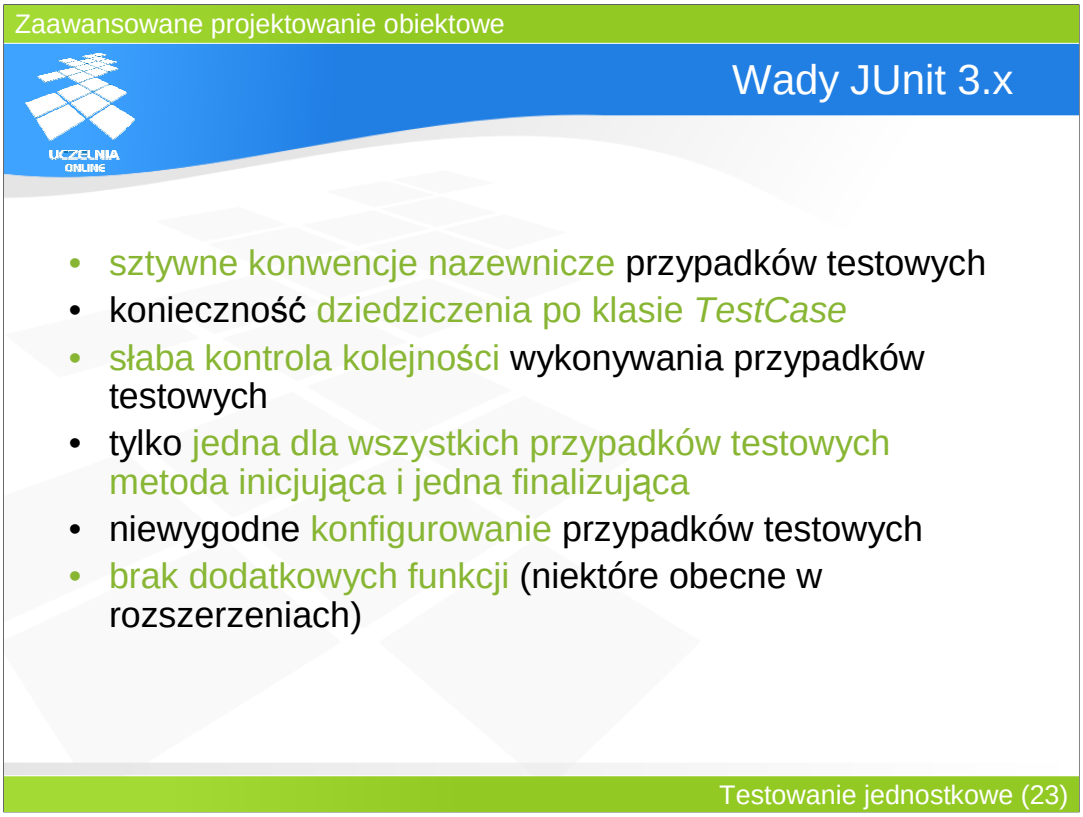


Jak już wspominaliśmy wcześniej, zgodnie z konwencją przyjętą w JUnit, każdej klasie produkcyjnej odpowiada klasa testująca. Powstaje jednak pytanie, jak logicznie powinny one wobec siebie być umieszczone. Z uwagi na konieczność dostępu do niektórych składowych o pakietowym (ang. *package private*) zakresie widoczności wskazane jest, aby znajdowały się one w tych samych pakietach. Jednak fizyczne umieszczanie klas w tych samych katalogach utrudnia zarządzanie konfiguracją testów i kodu produkcyjnego: niewskazane jest przecież dostarczanie odbiorcy programu testów wymieszanych z innymi klasami. Dlatego często stosuje się rozwiązanie z dwiema równoległymi strukturami pakietów, osobnej dla klas produkcyjnych, i osobnej dla klas testujących. Pozwala to osiągnąć wszystkie postawione przed chwilą cele.



Często pojawiające się pytanie dotyczy testowania metod niepublicznych. Jedna z odpowiedzi mówi, że testować należy tylko metody publiczne, ponieważ jedynie one uczestniczą w interakcji z innymi obiektami, zatem stanowią one jedyny dostępny na zewnątrz interfejs klasy. Inna szkoła dowodzi, że znacznie bardziej uzasadnione jest testowanie wszystkich metod o nietrywialnym zachowaniu, bez względu na zakres ich widoczności. Wówczas czasami konieczne staje się przetestowanie także metod prywatnych i chronionych.

W Javie jest to możliwe jedynie poprzez mechanizm Java Reflection. Rozwiązanie to na potrzeby testowania zostało zaimplementowane w dodatku do JUnita pod nazwą JUnitX w klasie PrivateAccessor. Pozwala ona zmienić poziom dostępu do dowolnej składowej (pola lub metody) i w razie potrzeby odwołać się do niej.



JUnit w wersji 3.x, mimo prostoty wewnętrznej i spójnego, logicznego projektu obiektowego, posiada szereg wad, które z czasem w coraz większym stopniu utrudniają jego stosowanie w praktyce:

•przyjęcie konwencji nazewniczych dotyczących nazw przypadków testowych jest dość nieporęczne;

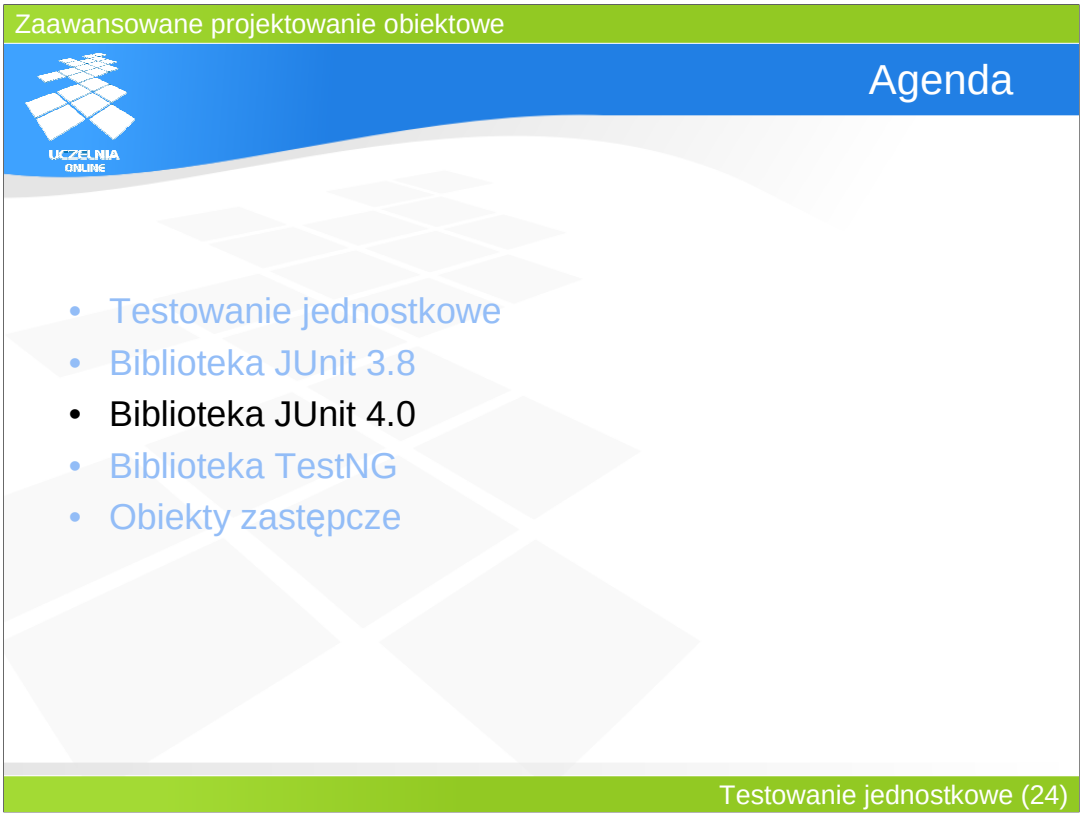
•ścisła integracja (poprzez dziedziczenie) z klasą TestCase stwarza dodatkową zależność, której można uniknąć;

•kontrola kolejności wykonywanych przypadków testowych jest utrudniona, ponieważ pierwotnie założono całkowitą ich niezależność od siebie;

•istnieje tylko jedna para metod inicjalizująco-finalizujących, i to wyłącznie na poziomie przypadku testowego. Powoduje to zbędną duplikację kodu w bardziej złożonych przypadkach;

•brak wielu istotnych funkcji (np. obsługi limitów czasowych, wbudowanej obsługi testowania wyjątków, braku mechanizmu konfiguracji przypadków testowych), spośród których część jest obecna w rozszerzeniach tworzonych przez niezależnych autorów.

Wady te oraz pojawienie się możliwości związanych z nowymi elementami języka Java w wersji 5.0 spowodowały powstanie oraz szybki rozwój dwóch bibliotek: JUnit 4.0, będącej napisaną od nowa wersją JUnit 3.8, wyposażoną w nowe możliwości, oraz TestNG – niezależnej implementacji częściowo zgodnej z JUnit 3.x, jednak opartej na nieco zmodyfikowanych założeniach.



Druga część wykładu jest poświęcone nowej wersji biblioteki JUnit w wersji 4.0

Założenia dla JUnit 4.x

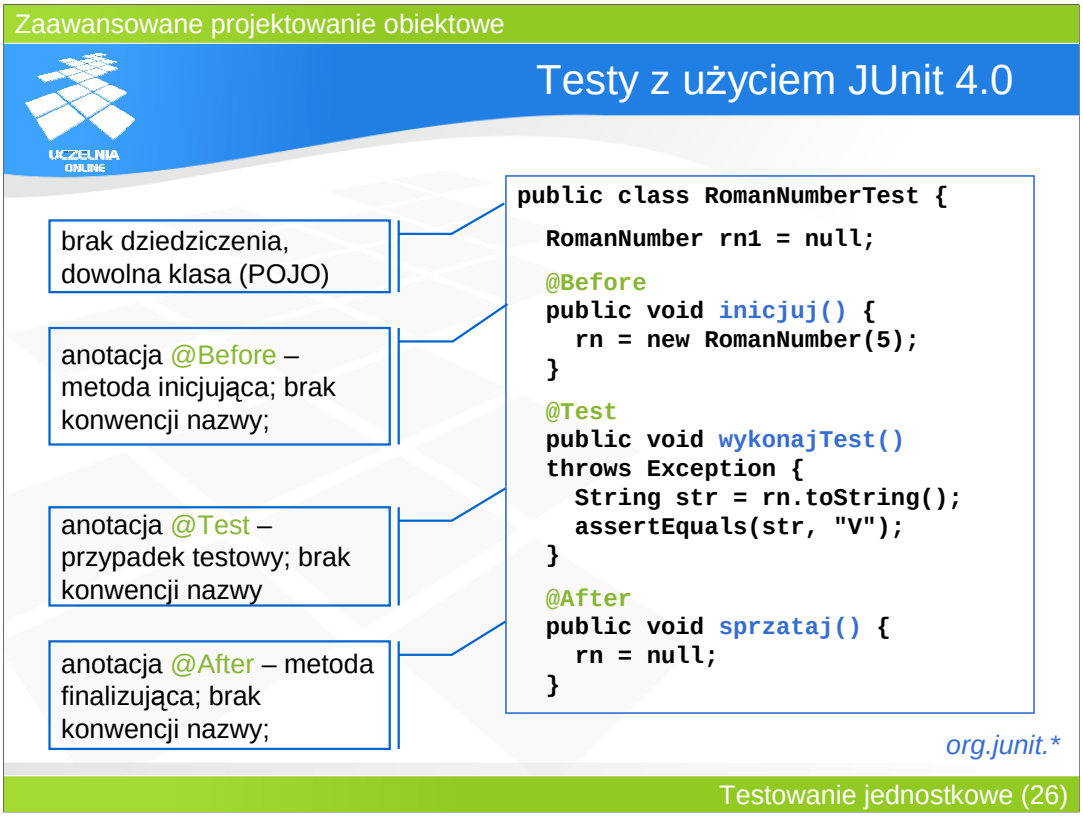
- brak konieczności dziedziczenia przez klasy testujące
- wykorzystanie anotacji Java 5.0 zamiast konwencji nazw
- metody inicjujące i finalizujące na poziomie metody i klasy
- wiele metod inicjujących i finalizujących
- wbudowana obsługa spodziewanych wyjątków
- pomijanie wybranych przypadków testowych
- obsługa limitów czasowych
- kompatybilność z JUnit 3.x

http://www.junit.org

W JUnit 4.0, tworzonym podobnie jak wersja 3.x przez K. Becka i E. Gammę, część z wymienionych wcześniej problemów została rozwiązana. W miejsce dziedziczenia po klasie TestCase oraz zamiast konwencji nazewniczej przypadków testowych zastosowano wprowadzone w JDK 5.0+ anotacje, które określają znaczenie poszczególnych metod. Rozwiązanie to jest znacznie bardziej eleganckie i zrozumiałe od narzuconego sposobu nazywania metod.

Ponadto rozszerzono możliwości inicjacji i finalizacji metod – obecnie można w klasie umieścić dowolną liczbę takich metod, oraz stworzono dodatkową parę metod inicjująco-finalizującą na poziomie klasy, a nie tylko przypadku testowego. Zostały też zaimplementowane mechanizmy służące do testowania wyjątków, ignorowania wybranych przypadków testowych i obsługi limitów czasowych.

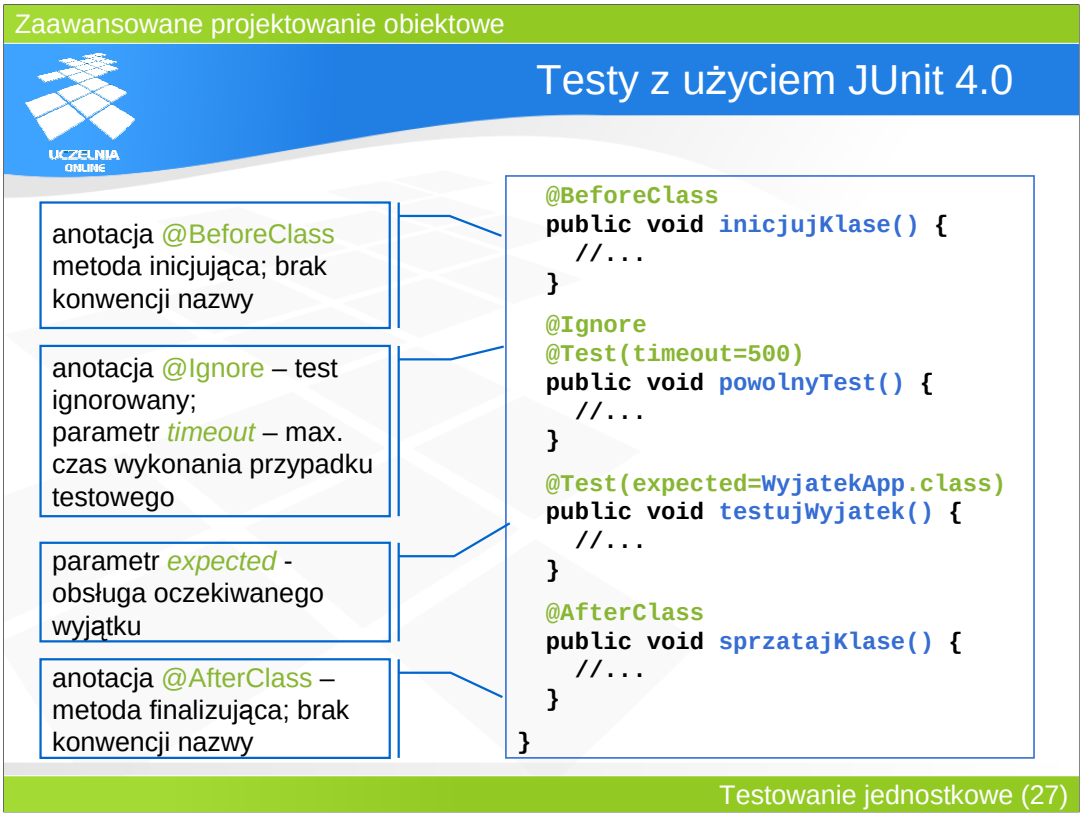
Co bardzo ważne, przyjęto zasadę dwustronnej kompatybilności z JUnit 3.x: dotychczasowe testy mogą być uruchamiane przez silnik JUnit 4.0, a nowe testy można opakować w adapter umożliwiający ich wykonywanie w starym środowisku uruchomieniowym.



Na slajdzie przedstawiono przykład klasy testującej RomanNumberTest, która wcześniej została napisana pod kątem użycia z biblioteką JUnit 3.8.

Metoda *inicjuj()* oznaczona anotacją @Before jest wykonywana analogicznie do metody *setUp()* znanej z JUnit 3.8, czyli przed wykonaniem każdego przypadku testowego. Jej odpowiednikiem finalizującym jest metoda *sprzataj()* oznaczona anotacją @After, która zostanie wykonana po każdym przypadku testowym. Liczba metod inicjujących i finalizujących jest dowolna, jednak między nimi nie są określone zależności kolejnościowe.

Przypadki testowe nie są już związane konwencją nazewniczą znaną z JUnit 3.8: do ich oznaczania wystarcza anotacja @Test.

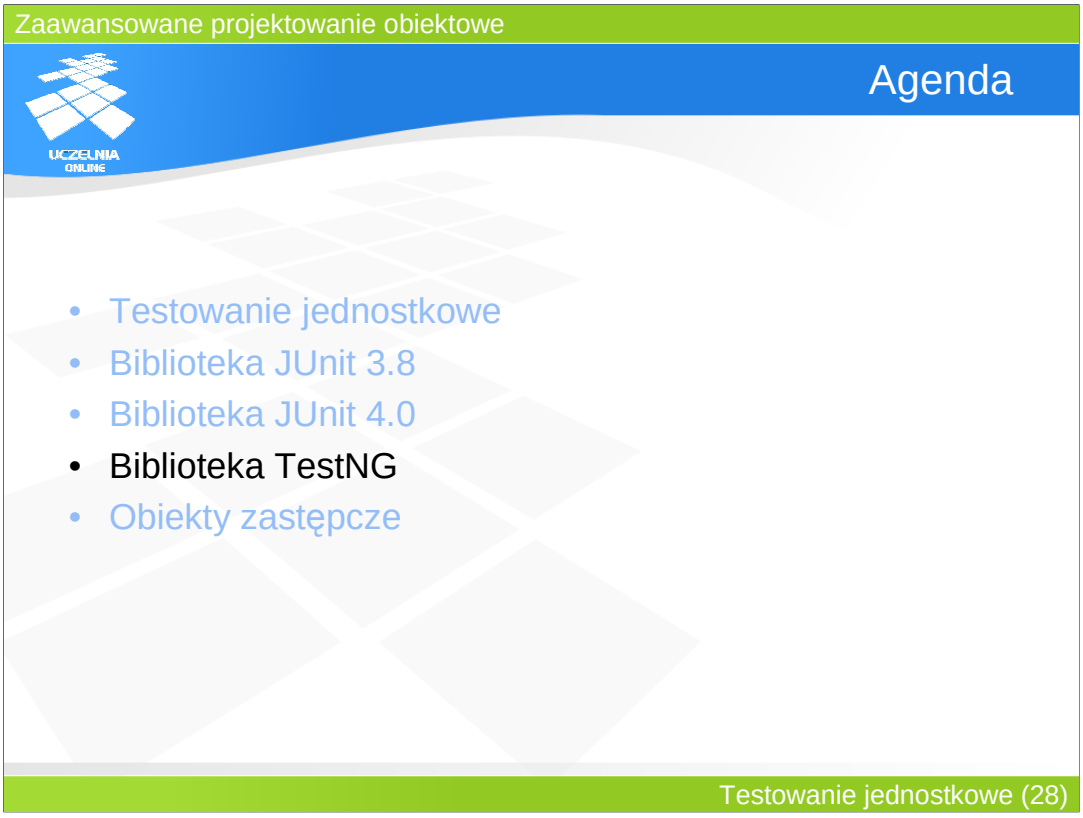


JUnit 4.0 pozwala na definiowanie dodatkowej pary metod inicjalizującej i finalizującej, tym razem na poziomie klasy. Metody te zostaną wykonane przez wszystkimi i po wszystkich przypadkach testowych należących do danej klasy testującej. Do ich oznaczania służą anotacje @BeforeClass i @AfterClass.

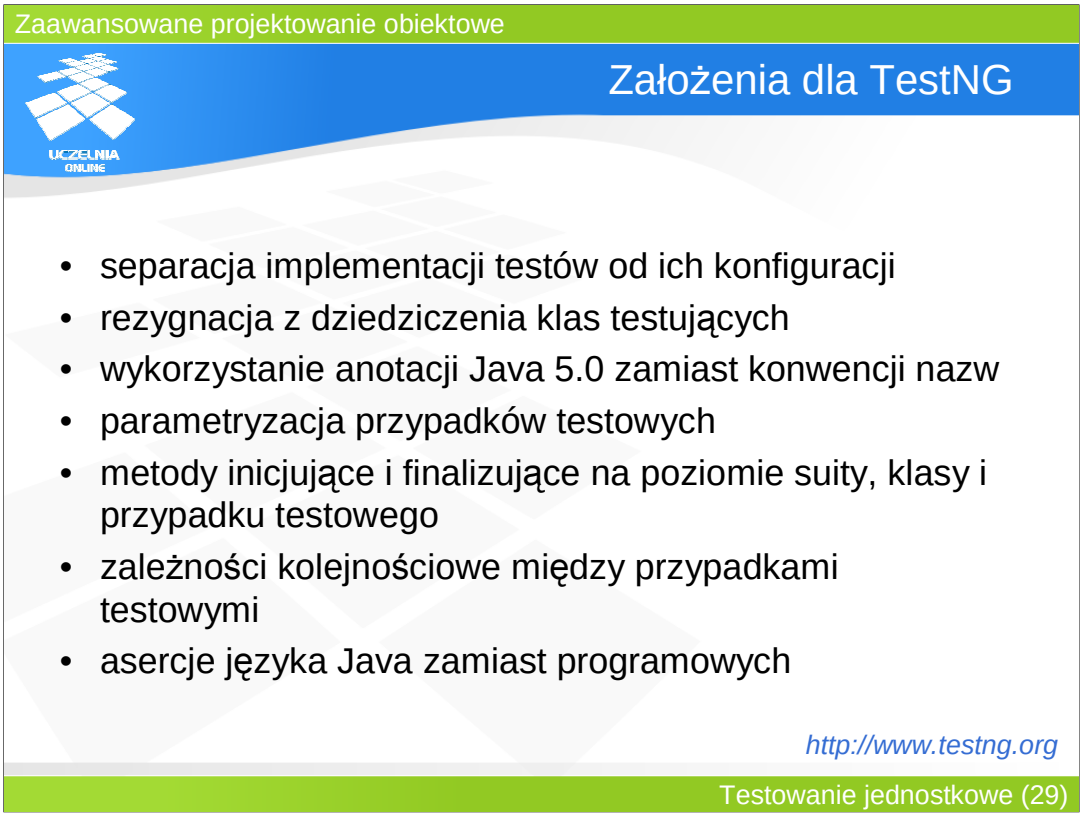
Anotacja @Ignore pozwala czasowo usunąć przypadek testowy z grona przypadków aktywnych: nie będzie on wykonywany w momencie wykonania testów.

Parametr *timeout* w anotacji @Test służy natomiast do określenia maksymalnego czasu wykonywania przypadku testowego. Jeżeli zostanie on przekroczony, jest przerywany, a odpowiednia informacja trafia do programisty.

Zaimplementowano także oparty o anotację mechanizm testowania wyjątków. Parametr *expected* w anotacji @Test pozwala określić, jaki typ wyjątku jest oczekiwany, i którego brak będzie błędem. Mechanizm ten zastępuje dotychczasowe prowizoryczne rozwiązania stosowane w JUnit 3.x



Alternatywą dla JUnit 4.0 jest biblioteka TestNG. Dzięki brakowi wstecznej zgodności z poprzednimi wersjami, jej możliwości znacznie wykraczają poza to co oferuje JUnit.



Najważniejsze cechy TestNG to:

•**Separacja implementacji testów od ich konfiguracji**. Testy jednostkowe są wykonywane wielokrotnie, jednak nie zawsze konieczne jest wykonanie wszystkich przypadków testowych. TestNG pozwala specyfikować zakres testów niezależnie od kodu tych testów w postaci pliku XML. Dzięki temu można zdefiniować różne zestawy testów (za pomocą osobnych plików XML) do różnych zastosowań.

•**Rezygnacja z mechanizmu dziedziczenia z wyróżnionej klasy testującej**. Klasa testująca zapisana w TestNG nie musi być potomkiem konkretnej klasy należącej do biblioteki – może nią być dowolna zwykła klasa (tzw. POJO – *Plain Old Java Object*).

•**Koniec z konwencjami nazewniczymi**. Podobnie, nazwy metod-przypadków testowych nie muszą odpowiadać jakimkolwiek konwencjom. Wystarczy, że są oznaczone anotacją @Test.

•**Parametryzacja przypadków testowych**. Przypadki testowe mogą przyjmować parametry typu String, których liczba i wartości są konfigurowane niezależnie od kodu testu.

•**Trójpoziomowa ziarnistość inicjalizacji i finalizacji**. Istnieje możliwość precyzyjnego ustalenia ziarnistości i zakresu inicjalizacji oraz finalizacji środowiska testowego: za pomocą adnotacji można oznaczyć metody wykonywane przed i po wszystkich przypadkach testowych w grupie (*@beforeSuite* i *@afterSuite*), w klasie testującej (*@beforeTestClass* i *@afterTestClass*) oraz – podobnie jak w JUnit'cie 3.x – pojedynczym przypadku testowym (*@beforeTestMethod* i *@afterTestMethod*)

Założenia dla TestNG (cd.)

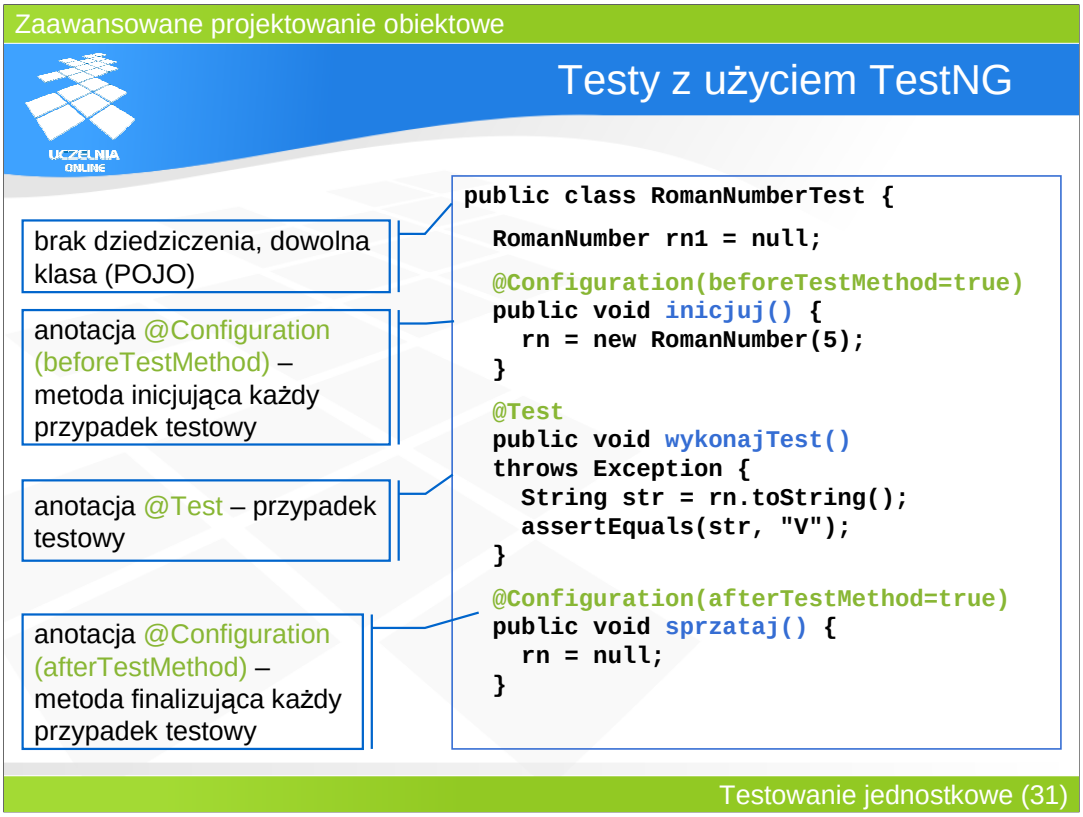
- separacja implementacji testów od ich konfiguracji
- rezygnacja z dziedziczenia klas testujących
- wykorzystanie anotacji Java 5.0 zamiast konwencji nazw
- parametryzacja przypadków testowych
- metody inicjujące i finalizujące na poziomie suity, klasy i przypadku testowego
- zależności kolejnościowe między przypadkami testowymi
- asercje języka Java zamiast programowych

http://www.testng.org

•**Wykorzystanie asercji wbudowanych w język w miejsce asercji programowych**.

•**Specyfikowanie zależności między grupami przypadków testowych i określanie kolejności ich wykonywania**. Cecha ta, nieobecna w JUnit, pozwala m.in. na automatyczne pomijanie tych testów, których zależności wcześniej nie zostały spełnione (tzn. zgłosiły wyjątki). Pozwala także ręcznie oznaczać niektóre testy do pominięcia w aktualnym wykonaniu.

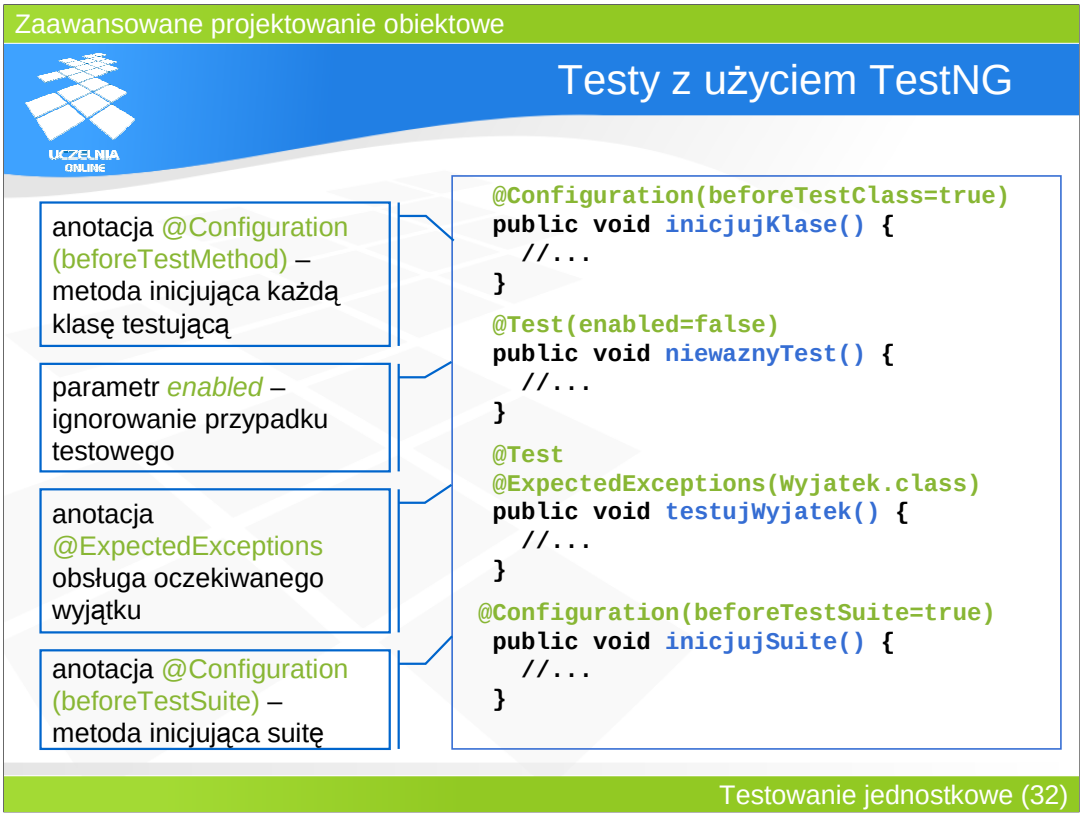
•**Proste wskazywanie oczekiwanych wyjątków**. Przypadek testowy sprawdzający pojawienie się oczekiwanego wyjątku zaimplementowany z wykorzystaniem JUnit 3.x musiał przechwycić ten wyjątek, a następnie go zignorować, natomiast brak wyjątku był sygnalizowany bezwarunkowym zgłoszeniem błędu. TestNG pozwala wskazać oczekiwany wyjątek za pomocą anotacji



Ponownie, przykładem jest klasa RomanNumberTest.

Metoda inicjująca *inicjuj()* jest oznaczona anotacją @Configuration z parametrem *beforeTestMethod*, który wskazuje, że inicjuje ona każdy przypadek testowy Jej odpowiednikiem finalizującym jest metoda *sprzataj()* z identyczną anotacją i parametrem *afterTestMethod*.

Podobnie jak w przypadku implementacji dla JUnit 4.0, metoda *wykonajTest()* jest przypadkiem testowym wykrywanym automatycznie przez środowisko TestNG dzięki zastosowaniu anotacji @Test.

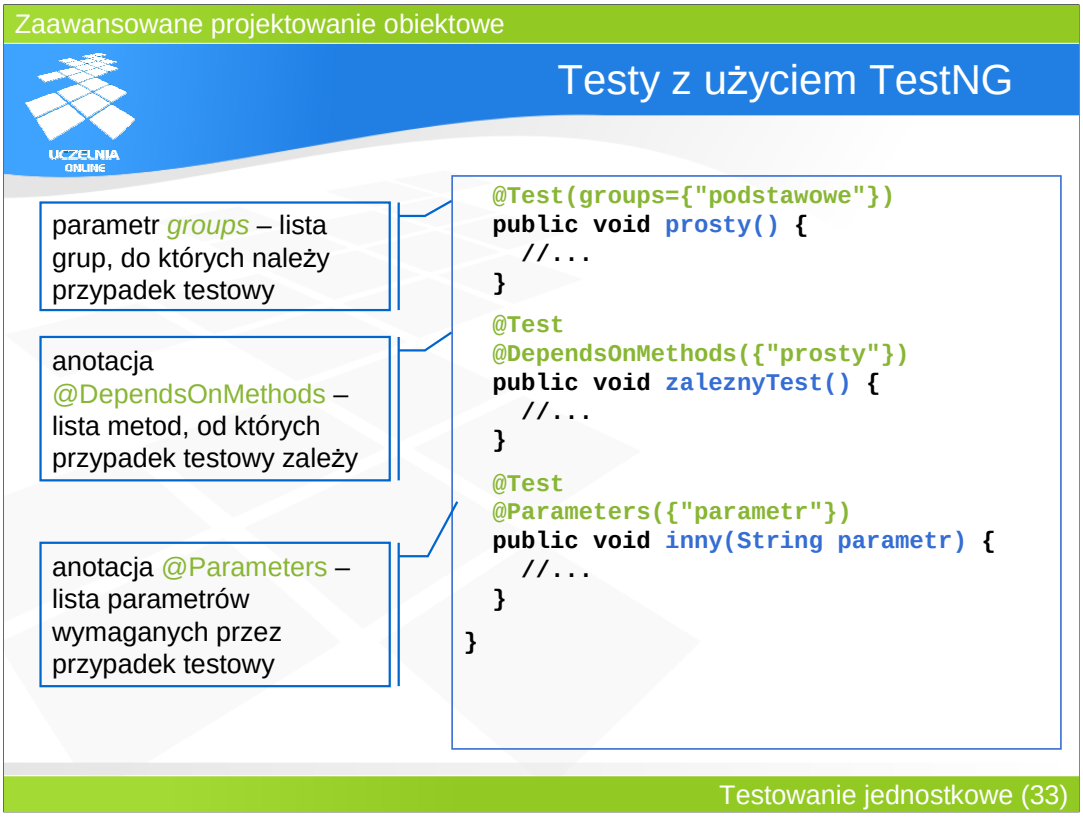


Metoda *inicjujKlase()* zostanie wykonana jeden raz przed wykonaniem wszystkich przypadków testowych zawartych w tej klasie.

Metoda *niewaznyTest()* zostanie pominięta, ponieważ parametr *enabled* anotacji @Test powoduje zignorowanie jej przez TestNG.

Metoda *testujWyjatek()* oczekuje pojawienia się wyjątku typu Wyjatek. Lista oczekiwanych wyjątków jest przekazywana za pomocą anotacji @ExpectedExceptions.

Metoda *inicjujSuite()* zostanie wykonana dokładnie raz dla całej suity (czyli jednego uruchomienia dowolnego zbioru przypadków testowych)

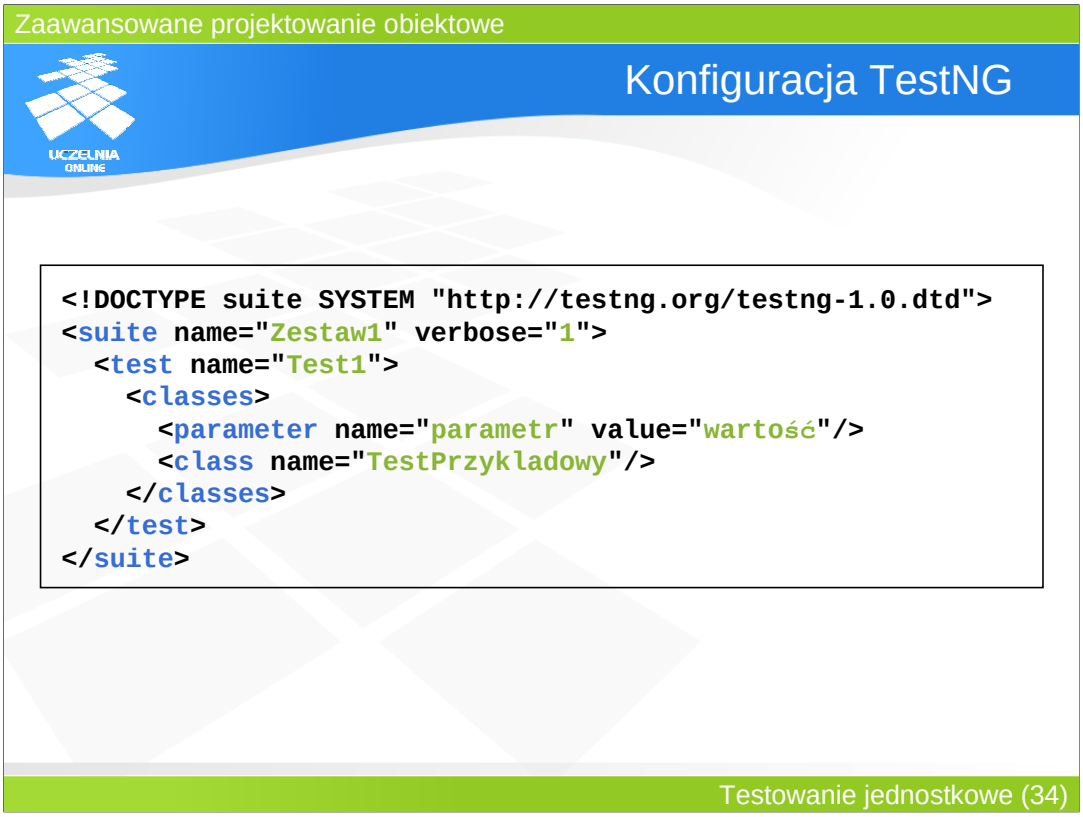


Na tym slajdzie przedstawiono bardziej zaawansowane cechy TestNG.

Anotacja @Test może posiadać parametr *groups*, który definiuje grupy, do jakich dany przypadek testowy należy. Uruchamiając testy, można podać grupy, jakie powinny zostać wykonane. Pozostałe (niewymienione grupy) zostaną zignorowane.

Metoda *zaleznyTest()* oznaczona anotacją @DependsOnMethods zostanie wykonana po poprawnym (tzn. bez zgłoszenia wyjątku) metody od której zależy, czyli metody *prosty().* Mechanizm ten pozwala m.in. tworzyć metody zawierające wspólne fragmenty kodu dla wybranych grup przypadków testowych oraz uniknąć wykonywania zależnych przypadków testowych, jeżeli wykonywane przed nimi metody (np. inne przypadki testowe) nie powiodły się. Pozwala to uniknąć fałszywych komunikatów o błędach, które w rzeczywistości są spowodowane błędnym wynikiem przypadków zależnych

Anotacja @Parameters zastosowana do przypadku testowego pozwala na przekazanie mu dowolnych parametrów, których ten przypadek wymaga. Parametry te (zawsze typu String!) są przekazywane poprzez plik konfiguracyjny.

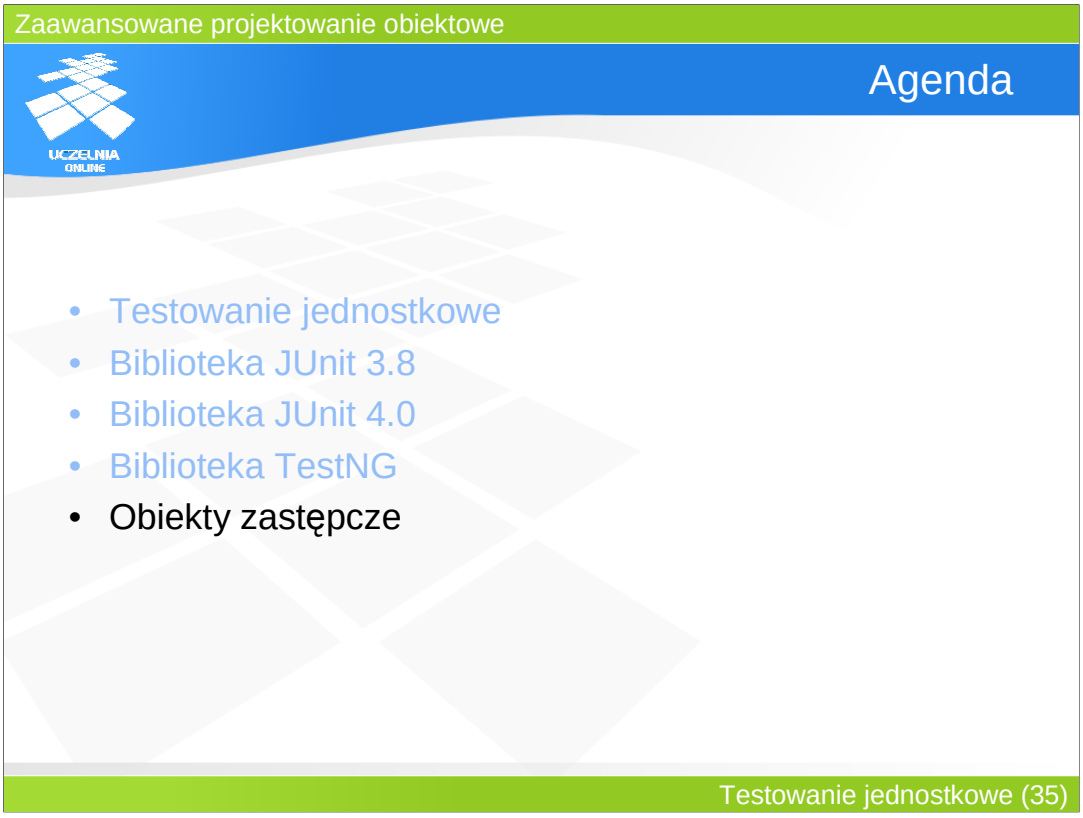


Pliki konfiguracyjne TestNG są zapisywane w postaci XML. Najwyższym poziomem konfiguracji jest suita o podanej nazwie (w typ przypadku Zestaw1). Wewnątrz niej znajdują się pojedyncze testy, zbudowane z klas.

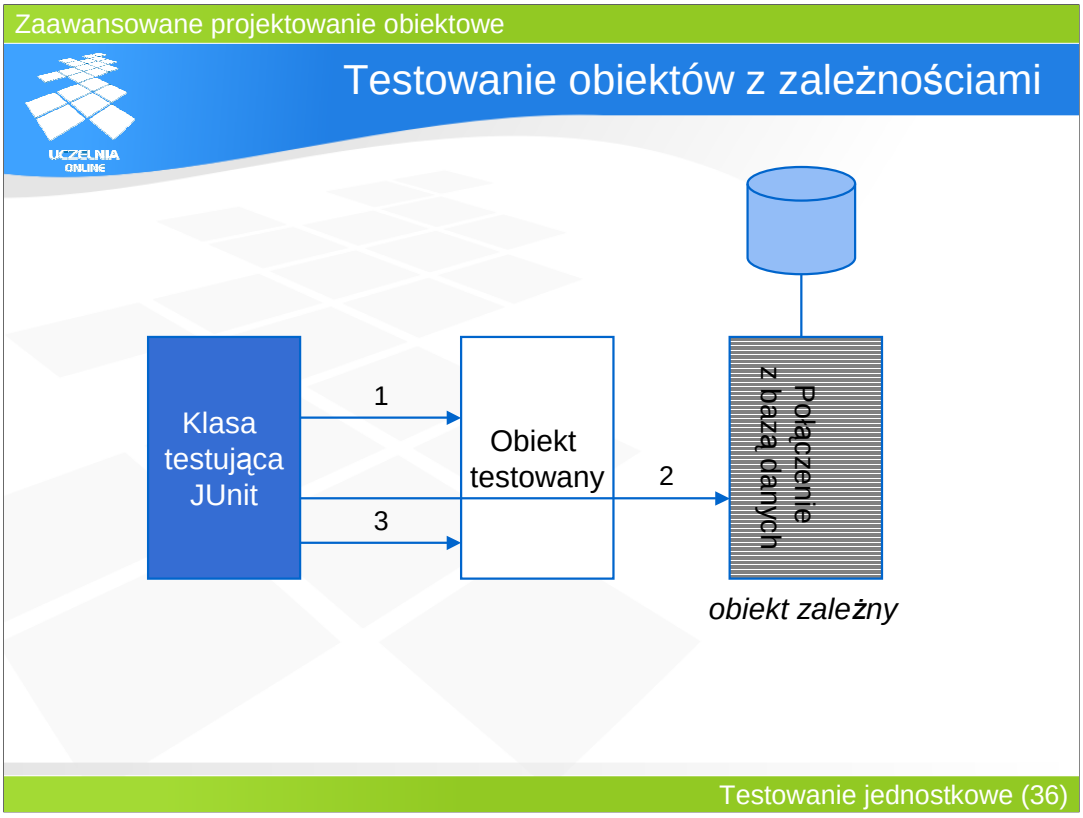
Za pomocą pliku można przekazać wartości parametrów do przypadków testowych, które tego wymagają, dzięki czemu ten sam przypadek testowy może zostać uruchomiony wielokrotnie z różnymi wartościami parametrów.

Ponadto można zignorować wybrane przypadki testowe lub połączyć je w grupy niezależnie od anotacji zapisanych bezpośrednio w kodzie.

Ponieważ uruchomienie testów TestNG wymaga przekazania także nazwy pliku konfiguracyjnego, można łatwo zarządzać konfiguracją testów, tworząc kilka różnych plików XML. Wówczas np. można wykonywać testy na różnych poziomach szczegółowości, koncentrując się na aktualnie istotnych elementach.



Ostatnia część wykładu jest poświęcona obiektom zastępczym, zwanym także imitacjami obiektów. Technika ta umożliwia rozszerzenie zakresu stosowania testowania jednostkowego na nowe obszary.



Typowy proces testowania jednostkowego, w którym obiekt testowany wymaga pewnego środowiska, polega na wykonaniu następujących kroków

1.Utworzenie instancji obiektu testowanego przez klasę testującą

2.Uzyskanie referencji do obiektów zależnych niezbędnych do utworzenia/użycia obiektu testowanego. Może to polegać na bezpośrednim utworzeniu tego obiektu bądź jego wyszukaniu (np. w katalogu JNDI), lub też uruchomieniu oprogramowania dostarczającego tego środowiska (np. serwera aplikacji)

3.Wywołanie przypadku testowego, który wywołuje metody w obiekcie testowanym

Jednak taka procedura jest niezupełnie zgodna z ideą testowania jednostkowego. W tym przypadku testom poddawany jest nie tylko sam obiekt testowany, lecz także wszystkie jego zależności. Powoduje to, że błąd znaleziony w teście może w rzeczywistości wynikać z obiektu testowanego lub jego zależności, które przecież aktualnie nie są testowane. Niezależnie od tego, nie do zaakceptowania jest sytuacja, w której dla potrzeb każdego przypadku testowego będzie uruchamiany serwer aplikacji, którego rozruch zajmuje dłuższy czas.

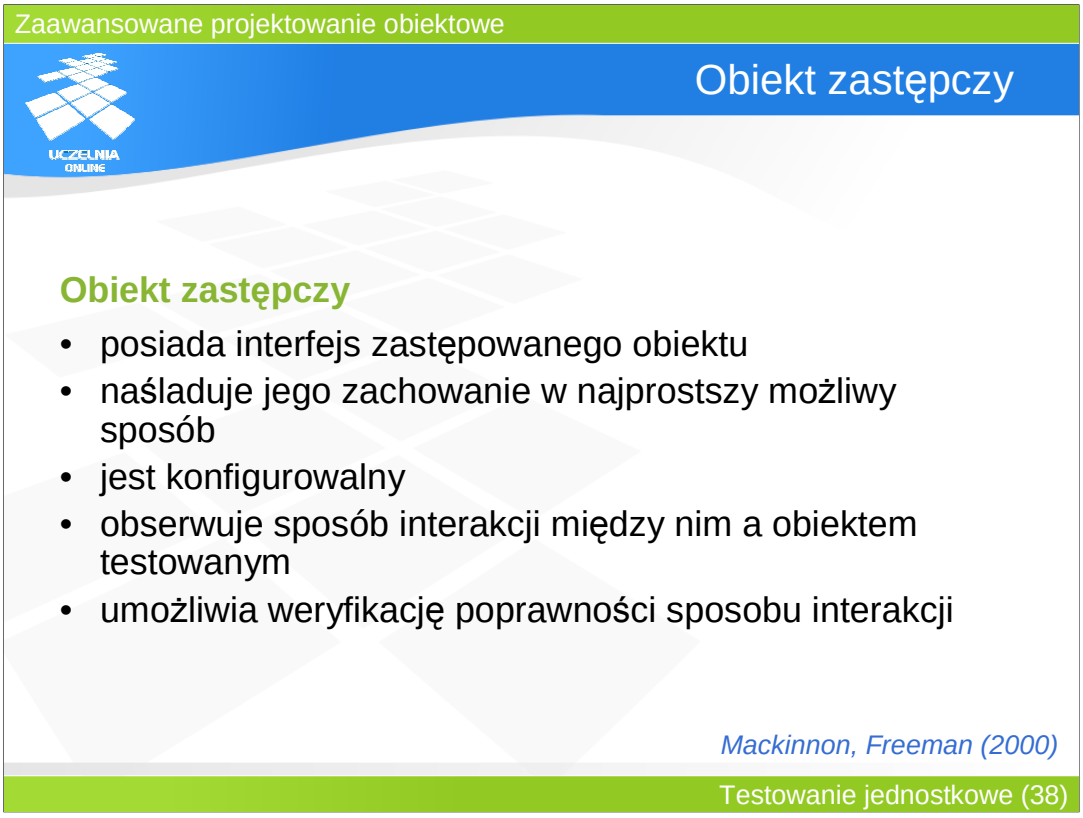
Celem testowania jednostkowego jest wyizolowanie obiektu testowanego od środowiska. Jednak jak osiągnąć ten cel, gdy obiekt ten wymaga spełnienia zależności?

Rozwiązaniem jest zastąpienie dla potrzeb przypadku testowego obiektu zależnego przez obiekt zastępczy (ang. *mock object*). Obiekty zastępcze są pod względem typu identyczne jak obiekty zależne i są zamiennikami wystarczającymi dla potrzeb uruchomienia testu.

W przypadku pokazanym na rysunku, obiekt połączenia z bazą danych jest zastąpiony przez obiekt zastępczy. Obiekt ten nie wykonuje żadnej operacji związanej z bazą danych, nie łączy się z nią ani nie wykonuje zapytań. Jego rola ogranicza się do naśladowania obiektu zależnego w takim tylko zakresie, jaki jest niezbędny do wykonania przypadku testowego.

W stosunku do tradycyjnej procedury testowania, zmieniony jest krok 2. Zależności są spełniane nie przez obiekt testowany, ale przez klasę testującą, która przekazuje obiektowi testowanemu instancje obiektów zastępczych.

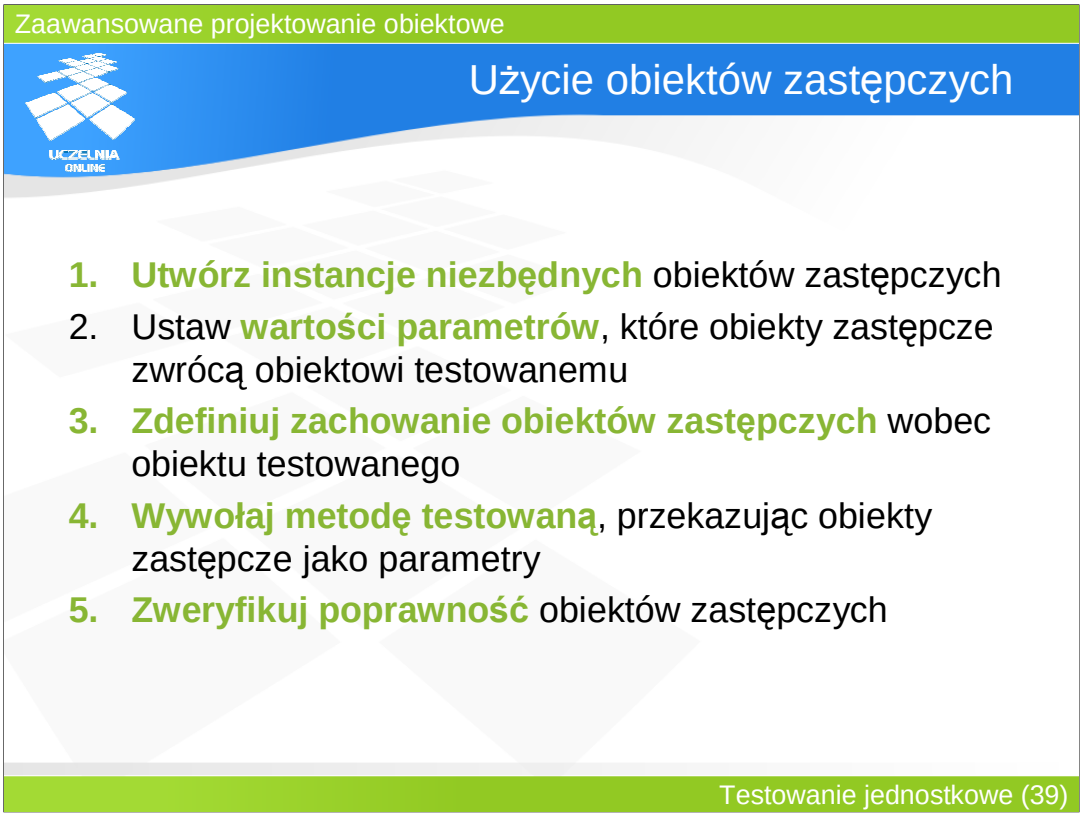
Co ważne, obiekt zastępczy NIGDY nie zastępuje obiektu testowanego, a jedynie jego zależności. Celem jest przecież sprawdzenie, czy obiekt testowany zachowuje się poprawnie, więc użycie zamiast niego obiektu zastępczego nie ma sensu.



Koncepcja obiektów zastępczych, zaproponowana przez Mackinnona i Freemana w 2000 roku, ma na celu rozwiązanie przedstawionych przed chwilą problemów związanych z testowaniem jednostkowym. Obiekt zastępczy posiada interfejs zastępowanego przez niego obiektu zależnego, a zatem może zastąpić go w interakcji z obiektem testowanym bez jakichkolwiek zmian w kodzie. Jest najprostszą możliwą implementacją obiektu zależnego, co oznacza, że nie wykonuje żadnych operacji, które nie są niezbędne z punktu widzenia obiektu testowanego. Zwykle jego działanie sprowadza się do zwracania poprzez jego metody określonych "na sztywno" wartości. Jest zatem prymitywnym naśladowcą obiektu zależnego.

Aby zdefiniować jego zachowanie, musi być konfigurowalny – można w nim określić wartości zwracane przez jego metody.

Wreszcie ostatnią własnością jest możliwość śledzenia interakcji z obiektem testowanym. Dzięki temu powstaje dodatkowa metoda weryfikacji zachowania obiektu testowanego: jeżeli wywoła on metody obiektu zastępczego niewłaściwą liczbę razy, nie zapyta o parametr, o który powinien zapytać – wówczas obiekt zastępczy może także zgłosić błąd, który będzie tożsamy z niespełnieniem asercji przypadku testowego.



Pełna procedura przebiega w następujących krokach:

1.Utworzenie instancji obiektów zastępczych koniecznych do utworzenia/użycia obiektu testowanego. W ten sposób przygotowane zostaje środowisko wykonania przypadku testowego

2.Obiekty zastępcze są konfigurowane wartościami parametrów, które zostaną przekazane obiektowi testowanemu, gdy on wywoła ich metody.

3.Obiekty zastępcze otrzymują dodatkową informację na temat spodziewanej interakcji z obiektem testowanym, np. minimalnej lub maksymalnej liczbie wywołań określonej metody. Informacja ta posłuży następnie do zweryfikowania poprawności stanu obiektów zastępczych po zakończeniu testu. W ten sposób pełni rolę dodatkowego mechanizmu weryfikacyjnego, niezależnego od asercji umieszczonych w przypadku testowym.

4.Metoda testowana jest wywoływana przez przypadek testowy. Jeżeli to konieczne, metoda testowana otrzymuje parametry będące obiektami zastępczymi.

5.Wyniki wywołania metody są weryfikowane przez obiekty zastępcze (patrz pkt. 3). Jeżeli jakakolwiek założona zależność nie została spełniona, zgłaszany jest wyjątek, który wskazuje na niepowodzenie przypadku testowego.

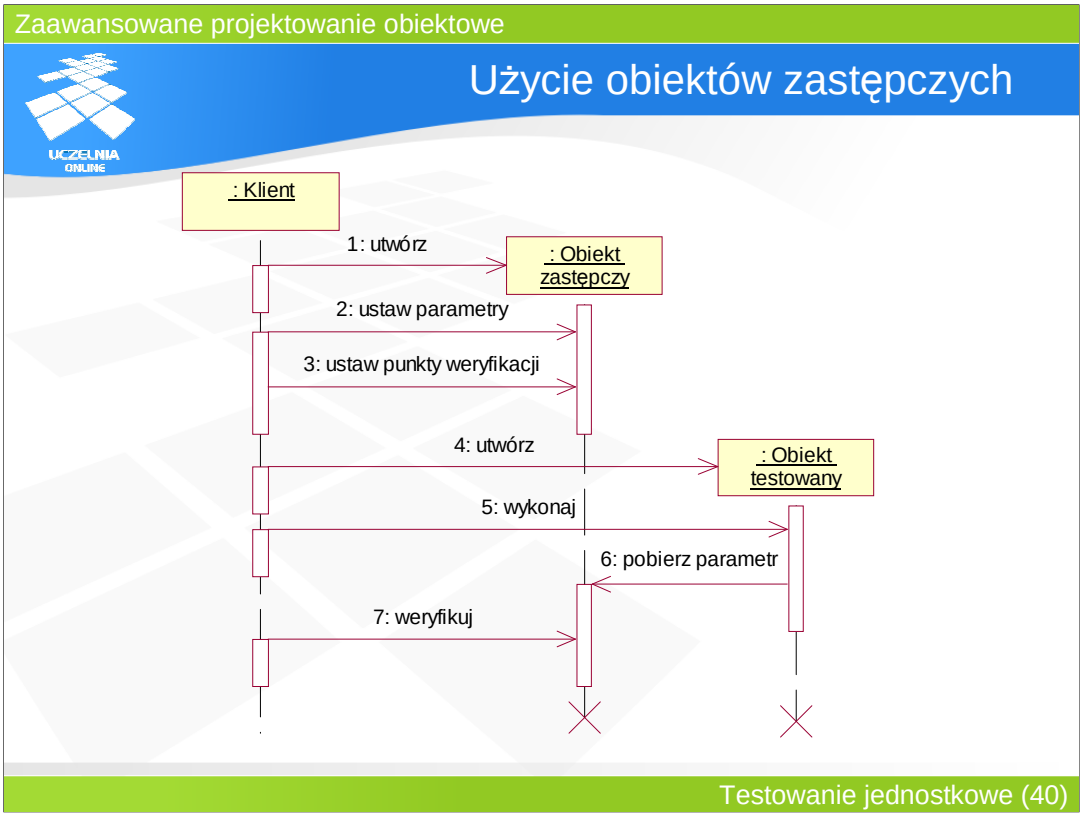


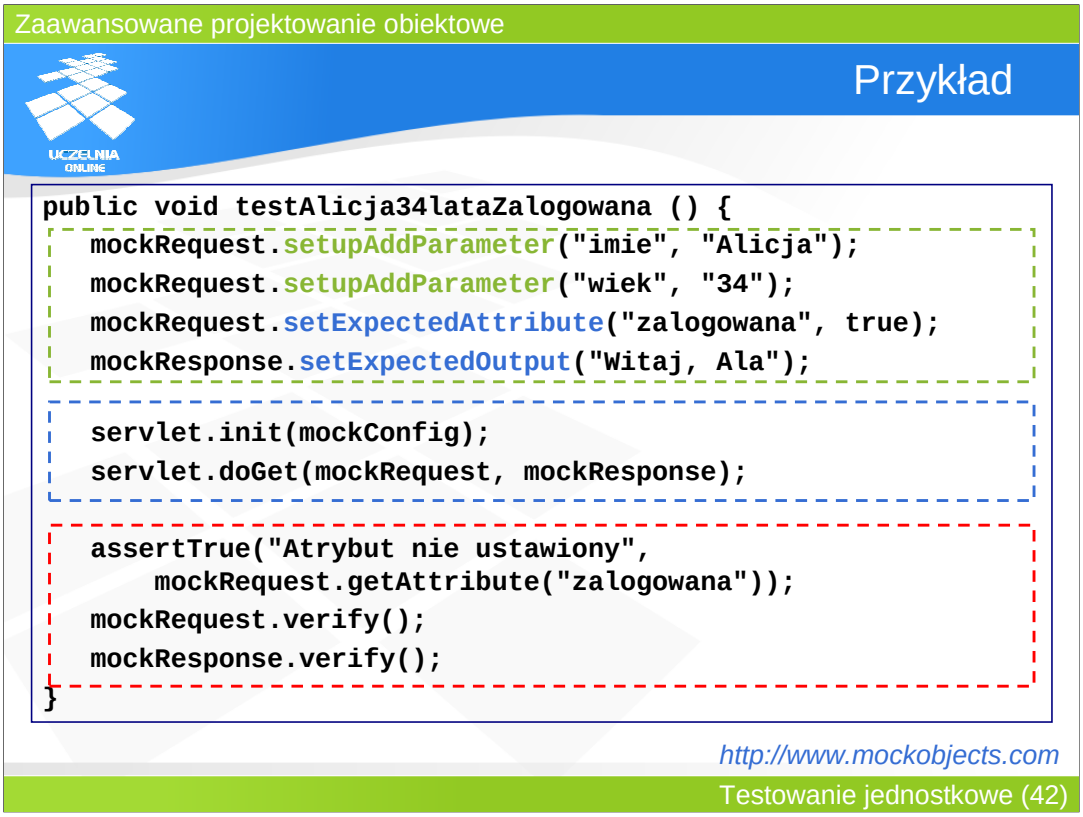
Diagram przedstawia przykładową wymianę komunikatów pomiędzy przypadkiem testowym, obiektem testowanym i obiektem zastępczym.

1. Klient tworzy instancję Obiektu zastępczego
2. Klient ustawia parametry Obiektu zastępczego, które następnie zostaną przekazane Obiektowi testowanemu
3. Klient ustawia punkty weryfikacji w Obiekcie zastępczym, sprawdzające poprawność interakcji między nim a Obiektem testowanym
4. Klient tworzy instancję Obiektu testowanego
5. Klient wykonuje metodę *wykonaj()* Obiektu testowanego
6. Obiekt testowany w odpowiedzi pobiera parametry z Obiektu zastępczego
7. Klient kontroluje osiągnięcie punktów weryfikacji w Obiekcie zastępczym

Zastosowania obiektów zastępczych

- trudno utworzyć i skonfigurować obiekt rzeczywisty
- rzeczywisty obiekt zachowuje się niedeterministycznie
- trudno wywołać określone zachowanie rzeczywistego obiektu (np. błąd sieciowy)
- znany jest tylko interfejs rzeczywistego obiektu
- rzeczywisty obiekt jest mało wydajny na potrzeby testu
- rzeczywisty obiekt posiada interfejs użytkownika
- stan obiektu wymaga weryfikacji po wykonaniu przypadku testowego

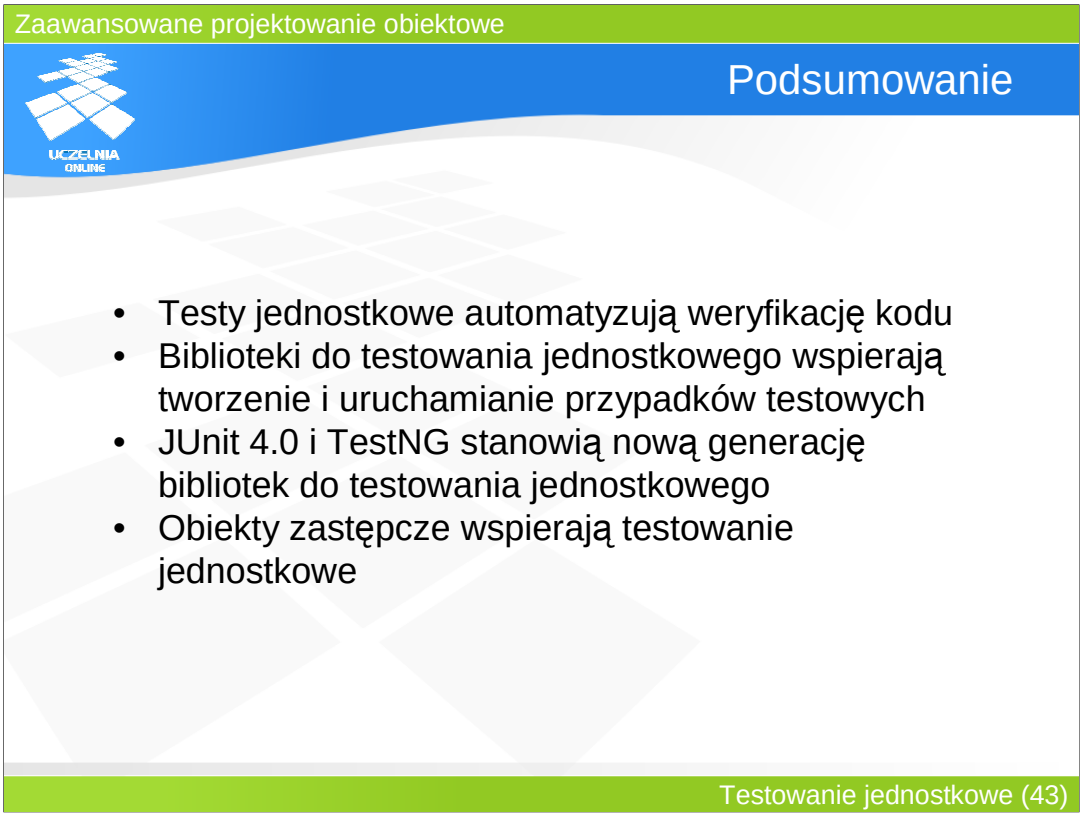
Obiekty zastępcze okazują się skutecznym rozwiązaniem w wielu sytuacjach, w których użycie rzeczywistego obiektu zależności jest trudne, niewygodne lub niemożliwe.



Przykład przedstawia zastosowanie idei obiektów zastępczych do testowania serwletów. Serwlety służą do obsługi protokołów opartych o model klient-serwer, a ich najpopularniejszym zastosowaniem jest tworzenie dynamicznych aplikacji WWW. Serwlety są w znacznym stopniu zintegrowane z kontenerem – obiektem, który je tworzy i zarządza ich cyklem życia. Testowanie serwletów poza kontenerem jest zatem bardzo trudne. Na przykład metoda *doGet()*, która służy do obsługi żądania GET protokołu HTTP (jak zresztą niemal wszystkie metody serwleta), wymaga przekazania dwóch paramterów: HttpServletRequest, reprezentującego nadesłane przez klienta żądanie HTTP, i HttpServletResponse, które jest tworzoną przez serwlet odpowiedzią HTTP. Obiekty te są tworzone i konfigurowane przez kontener, zatem nie można wywołać tej metody poza kontenerem.

Z pomocą przychodzą obiekty zastępcze, które zastępują te obiekty. Są one najpierw konfigurowane parametrami wejściowymi, które zwykle przesłałby klient w żądaniu, oraz oczekiwanymi parametrami, które powinny pojawić się w tych obiektach po wywołaniu metody serwleta. Następnie serwlet (utworzony za pomocą konstruktora, poza kontenerem), jest konfigurowany (metoda *init()* także przyjmuje obiekt zastępczy jako parametr) i wywoływana jest metoda *doGet()* z obiektami zastępczymi jako parametrami.

Po wywołaniu weryfikowane są asercje oraz spełnienie oczekiwanych zależności zdefiniowanych wewnątrz obiektów zastępczych na początku.



Podczas wykładu przypomniano koncepcję testów jednostkowych, ich przeznaczenie oraz specyfikę działania. Przedstawiono także trzy biblioteki wspomagające tworzenie i uruchamianie takich testów: JUnit 3.8, JUnit 4.0 i TestNG. Dwie ostatnie biblioteki stanowią odpowiedź na braki funkcjonalne JUnita 3.8 i, wykorzystując nowe możliwości języka Java, oferują znacznie prostsze metody tworzenia, konfiguracji i wykonania przypadków testowych. Ostatnim punktem była koncepcja obiektów zastępczych, które pozwalają na wykonywanie testów jednostkowych także w sytuacji, gdy niemożliwe jest spełnienie zależności obiektów testowanych.