TESTOWANIE OPROGRAMOWANIA

„Uważaj na ten program – ja tylko udowodniłem jego poprawność, nie testowałem go”

Donald Knuth

Plan prezentacji

1. Testowanie – wstęp
2. Refaktoryzacja
3. Pojęcia związane z testowaniem
4. Programowanie sterowane testami
5. Testy jednostkowe
   - 5.1. `JUnit`
   - 5.2. `CppUnit`
6. Testy integracyjne
7. Testy systemowe
8. Testy regresyjne
9. Pozostałe testy
10. Hierarchia testów
11. Narzędzia wspomagające testowanie
12. Źródła i przydatne linki

Testowanie – wstęp

„Testowanie może wykazać wyłącznie istnienie bugów, nigdy ich brak.”

Edsger W. Dijkstra

„Jeśli nie masz ochoty na testowanie swojego produktu, pamiętaj, że twoi klienci też nie będą tego robić”

autor nieznany

Testowanie – wstęp

- testowanie kodu na bieżąco wielokrotnie skraca czas potrzebny na usunięcie błędów w oprogramowaniu
- bez testów ukończenie programu (stworzenie działającej wersji) jest praktycznie niemożliwe

Testowanie – wstęp

Co można testować?

- poprawność – zgodność faktycznego zachowania programu z oczekiwanym
- wydajność – spełnianie wymagań czasowych
- niezawodność – odporność na działanie pod wymaganym obciążeniem
- bezpieczeństwo – odporność na ataki, ochrona danych

Testowanie – wstęp

Typy testowania ze względu na perspektywę

- black-box testing – spojrzenie na system z zewnętrznego punktu widzenia; system jako „czarna skrzynka”, która ma realizować określone zadania i tylko pod tym kątem jest testowana – np. testy odbioru
- white-box testing – testowanie obejmujące wgląd we wnętrze programu: testujący przeprowadza test jednocześnie badając kod programu. Wymagane umiejętności programistyczne – np. testy jednostkowe

Refaktoryzacja

refaktoryzacja – poprawa jakości kodu bez zmiany jego funkcjonalności

- kluczowa dla skutecznego testowania i poprawiania błędów
- przeprowadzenie testu to tylko połowa sukcesu - należy jeszcze poprawić błąd
- dobrze napisany kod ułatwia programowanie przyrostowe („małymi krokami”)

Refaktoryzacja

„Pierwszy lepszy głupiec potrafi napisać kod zrozumiały dla komputera. Dobrzy programiści piszą kod zrozumiały dla człowieka.”

Martin Fowler

„To świetna zabawa, kiedy można rozwikłać zagadkę morderstwa, ale kod nie powinien wymagać rozwikłania. Kod powinien dawać się czytać.”

Steve McConnell

Testowanie – podstawowe pojęcia

bug – każde zachowanie programu, które odbiega od oczekiwanego; zadaniem testów jest ich „wyłapywanie” i diagnozowanie

jednostka – pojedynczy element programu, podlegający testom jednostkowym, np. metoda, funkcja, klasa

przypadek testowy – zestaw danych, który służy do przetestowania danej jednostki; zawiera dane wejściowe i odpowiadający im poprawny wynik

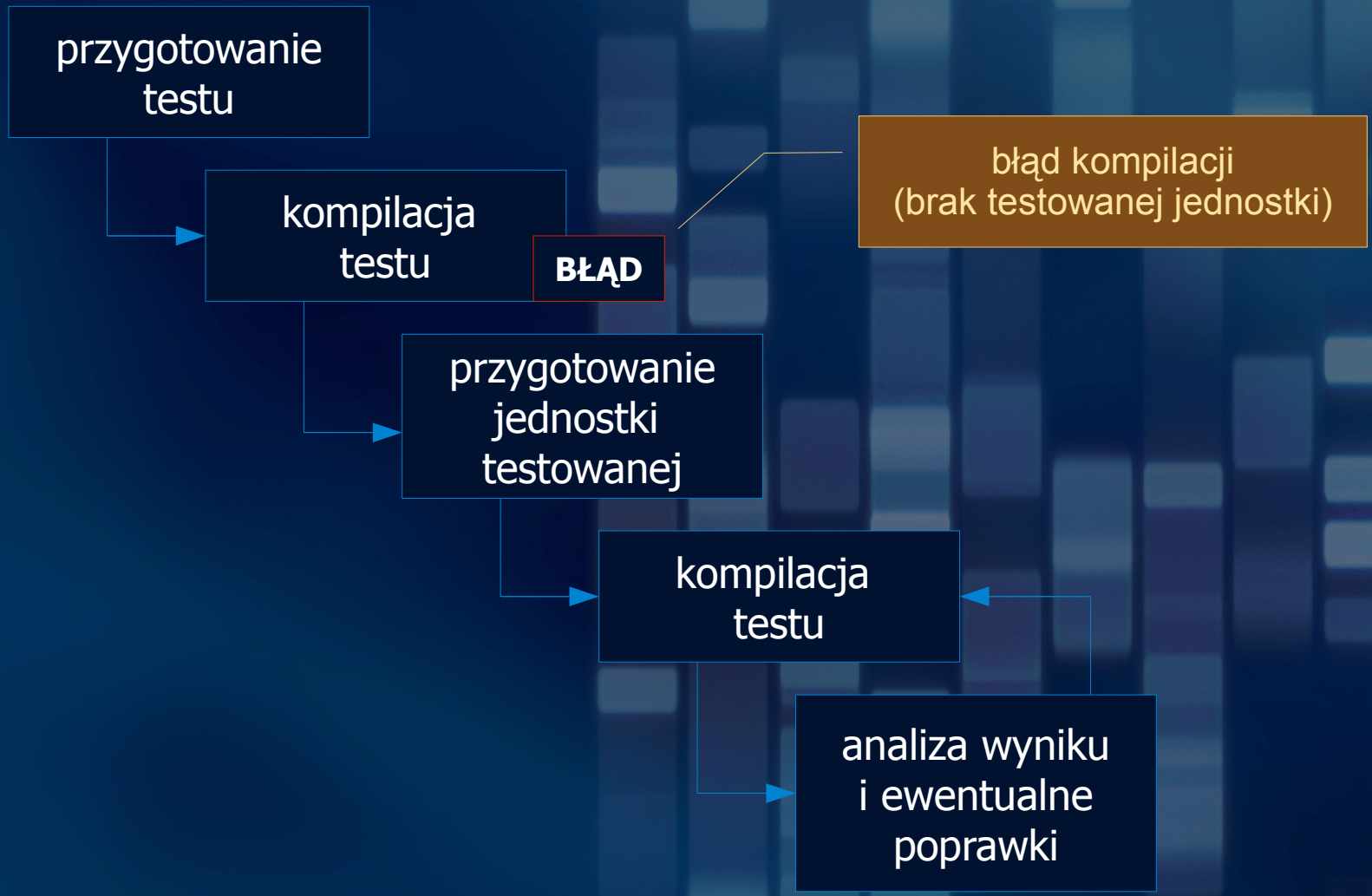
Programowanie sterowane testami

(ang. *test-driven programming*)

- testy jako sformalizowany sposób zapisu wymagań funkcjonalnych
- najpierw przygotowywany jest test (klasa testowa, przypadek testowy), a dopiero potem tworzona jest jednostka, która ma za zadanie „przejść” test
- niezaliczenie testu jest równoznaczne z błędnym kodem (niespełnieniem wymagań)

Programowanie sterowane testami

(ang. *test-driven programming*)



Programowanie sterowane testami

(ang. *test-driven development*)

ZALETY

- bardzo przejrzyście sformułowane wymagania względem systemu
- zmniejszenie ilości błędów – usterki wyłapywane „na bieżąco”
- oszczędności czasowe! (mniej debugowania)
- modularność, rozszerzalność kodu

WADY

- dodatkowe nakłady pracy bez wyczuwalnych zysków (więcej tworzonego kodu)
- żmudny proces tworzenia testów

Testy jednostkowe

- najniższy poziom testów zorganizowanych
- poprawność pojedynczych jednostek (klas, funkcji)
- fragment programu (jednostka) poddawany jest testowi, a wynik działania porównywany jest z wcześniej przygotowanym wynikiem wzorcowym

Testy jednostkowe

- bardzo elastyczne tworzenie przypadków testowych (można np. stworzyć zestaw testowy, który oczekuje na rzucenie określonego wyjątku)
- testy zautomatyzowane, wykonywane na bieżąco
- tworzą swoistą dokumentację – czytając kod testu, można rozpoznać, jaka funkcjonalność jest wymagana od danej jednostki

Testy jednostkowe

Mock Objects

(ang. *mock* - przedrzeźniać, udawać)

- obiekty symulujące działanie zewnętrznych, złożonych komponentów
- testy powinny być odizolowane od zewnętrznych czynników – testowana powinna być wyłącznie jednostka
- przykład – testując moduł wyświetlania, zamiast łączyć się z bazą przy każdym teście, tworzymy „podróbkę” połączenia, oszczędzając czas

Testy jednostkowe

Asercje

- warunki konieczne do zaliczenia testu
- jeden test – zero lub więcej asercji; niezaliczenie którejkolwiek asercji to przerwanie testu z wynikiem negatywnym

Przykłady:

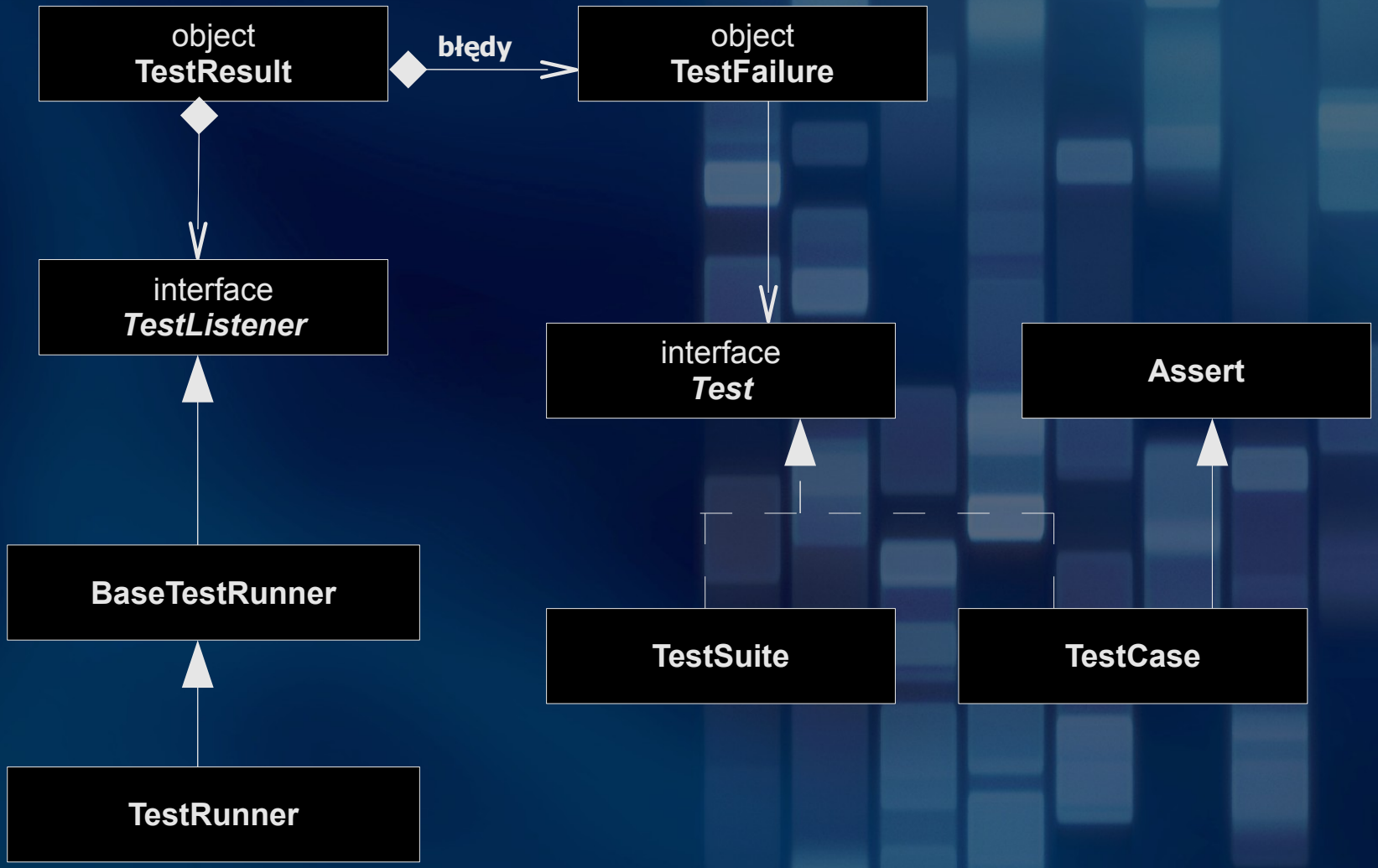
- `assertEquals(a, b)` – przechodzi, gdy `a == b`
- `assertNotNull(a)` – przechodzi, gdy `a` nie jest null

JUnit – testy jednostkowe w Javie

http://junit.org

- wspiera zautomatyzowane przeprowadzanie testów
- klasy testowe opisywane za pomocą annotacji
- bogata kolekcja pluginów do integracji Junit różnymi środowiskami użytkownika (Eclipse, NetBeans)

JUnit – testy jednostkowe w Javie



JUnit – testy jednostkowe w Javie

Przykład

Przetestujemy metodę `max()` z klasy `java.lang.Math`

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTestClass {
    @Test
    public void testMax() {
        //kod testujący
    }

}
```

JUnit – testy jednostkowe w Javie

Przykład

Testujemy metodę `max()` z klasy `java.lang.Math`

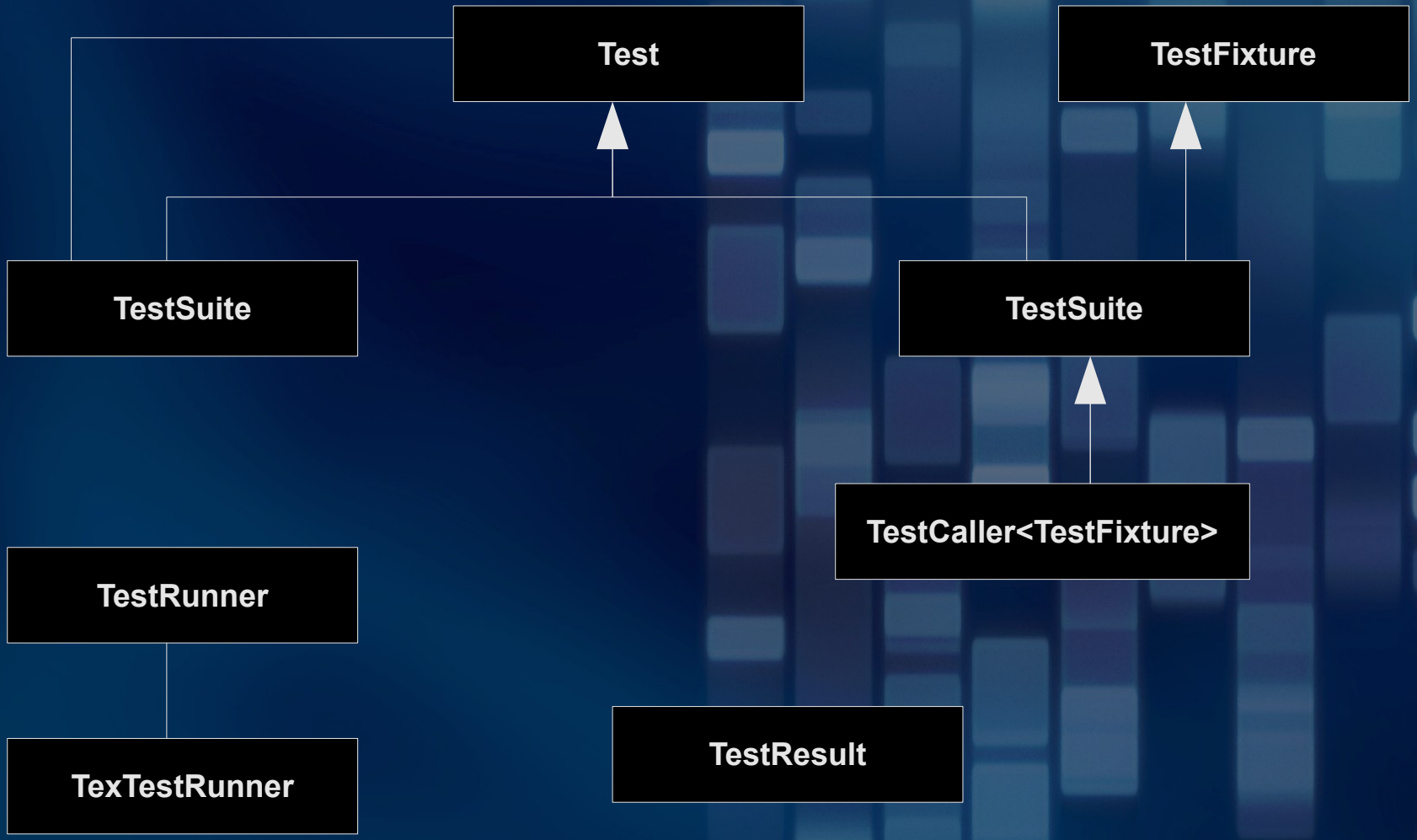
```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTestClass {
    @Test
    public void testMax() {

        assertEquals( 3, (int)Math.max(3,6) ); //oblany
        assertEquals( 6, (int)Math.max(3,6) ); //zaliczony

    }

}
```

CppUnit – testy jednostkowe w C++

http://cppunit.sourceforge.net

- Podobnie jak Junit wspiera automatyczne przeprowadzanie testów
- klasy testujące dziedziczą po klasie CppUnit::TestCase i przykrywają metodę wirtualną runTest()
- rezultat jest obiektem (referencją do obiektu) klasy CppUnit::TestResult

CppUnit – testy jednostkowe w C++



CppUnit – testy jednostkowe w C++

Przykład

Przetestujemy następującą funkcję `max()`:

```
int max (int a, int b) {
    if ( a > b ) {
        return a;
    }
    else {
        return b;
    }
}
```

Tworzymy klasę testującą:

```
#include <cppunit/TestResult.h>
#include <cppunit/TestCase.h>

class Max_test : public CppUnit::TestCase {
public:
  void runTest() {
    assert(max(1,2)==1); //sztucznie wprowadzony błąd!
    assert(max(2,1)==2);
};
```

CppUnit – testy jednostkowe w C++

Przykład

Uruchamiamy test z klasy testującej:

```
void main() {
    CppUnit::TestResult result;
    Max_test max;

    max.run( &result );
}
```

Testy integracyjne

- sprawdzanie poprawności współpracy pomiędzy komponentami systemu
- coraz większe grupy komponentów są integrowane i testowane pod kątem wzajemnej komunikacji (met. iteracyjna), do momentu przetestowania systemu jako całości
- wykrywa usterki w interfejsach

Testy systemowe

- testowanie systemu z perspektywy użytkownika końcowego
- przeprowadzane na kompletnym (w pełni zintegrowanym) systemie
- badanie pod kątem spełnienia wymagań funkcjonalnych
- testerzy umyślnie usiłują wywołać błędy, robiąc wszystko, czego tylko można się spodziewać po userze (ang. *destructive attitude*)

Testy regresyjne

„Patch to poprawka do programu, usuwająca stare błędy i dodająca nowe.”

Prawo Murphy'ego

„Błędy pojawią się w jednej części działającego programu, kiedy zmienisz drugą, zupełnie z nią niepowiązaną”

Prawo Murphy'ego

Testy regresyjne

- polegają na ponownym testowaniu fragmentu programu po wprowadzeniu modyfikacji
- mają na celu upewnienie się, że wprowadzona poprawka/modyfikacja nie stworzyła nowych bugów w innych częściach programu
- częstą metodą przeprowadzania jest ponowne uruchomienie wcześniej zaliczonych testów – sprawdzenie, czy nie pojawiają się nowe błędy

Pozostałe testy

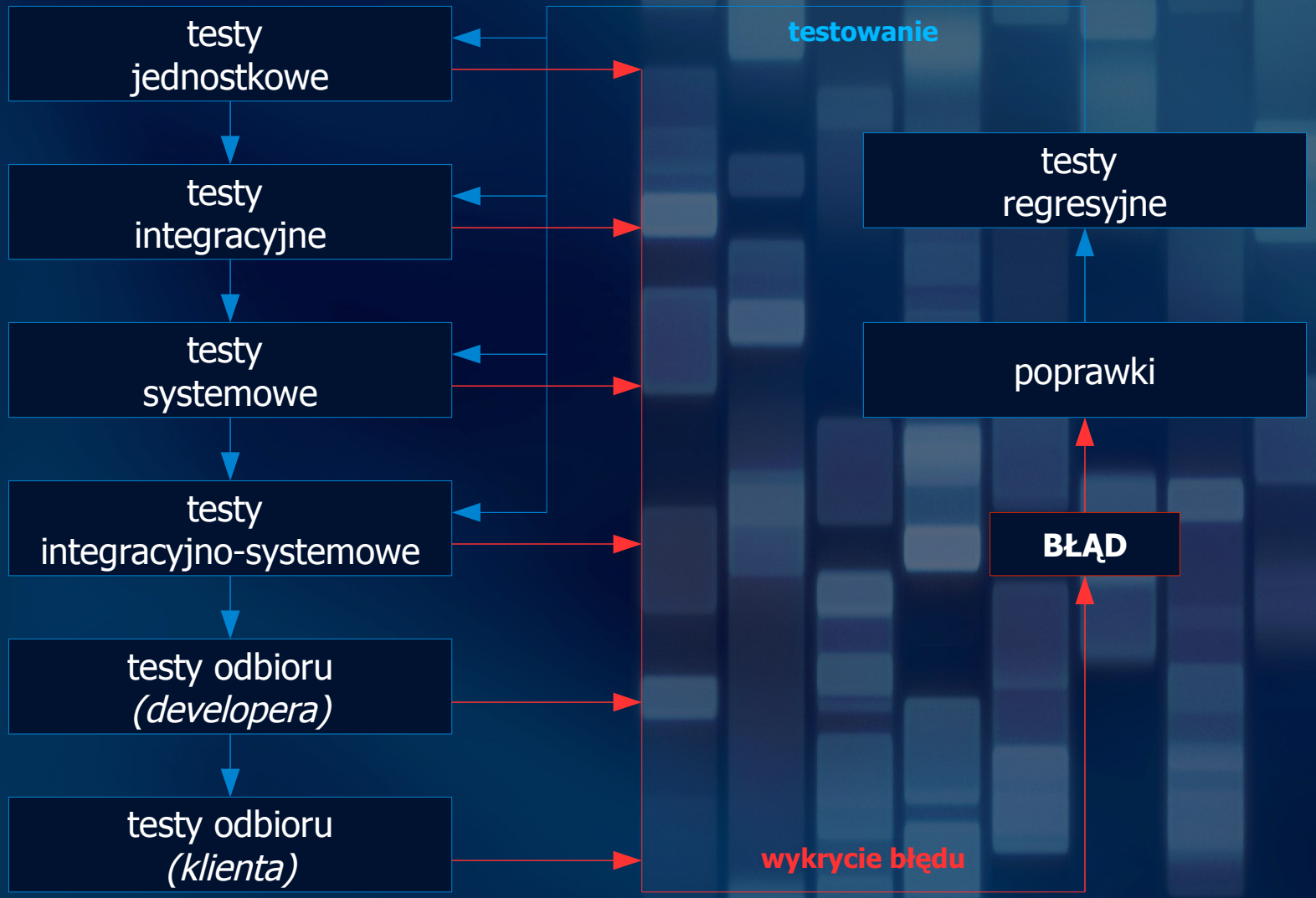
Testy integracyjno-systemowe

- testowanie stworzonego systemu pod kątem współpracy z innymi (system jako komponent)

Testy odbioru

- ostateczne testowanie gotowej aplikacji, gotowej do przekazania klientowi
- klient może dokonać testów odbioru na własną rękę (UAT, ang. *User Acceptance Testing*)

Hierarchia testów



Narzędzia wspomagające testowanie

xUnit

umowna nazwa bibliotek/frameworków do testów jednostkowych
dostępny dla wielu języków

JIRA, trac, Bugzilla

oprogramowanie wspomagające zarządzanie projektem,
raportowanie błędów i usterek

Canoo WebTest, WatiN

otwarte narzędzia do efektywnego testowania aplikacji webowych

JSystem

narzędzie do zautomatyzowanego tworzenia testów systemowych

więcej:

http://www.opensourcetesting.org/

Źródła, dalsze informacje

http://wazniak.mimuw.edu.pl/
[PL] (kurs *Zaawansowane CPP*, wykład 4 - „Testowanie”)

http://en.wikipedia.org/wiki/Software_testing
[EN] (bardzo rozbudowany artykuł)

http://www.testinggeek.com/
[EN] (portal dla testerów)

http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/
[EN] (artykuł na temat testowania oprogramowania)

Książki:

„Software Testing: A Craftsman's Approach” *Paul Jorgensen*

„Sztuka testowania oprogramowania” *Glenford J. Myers*