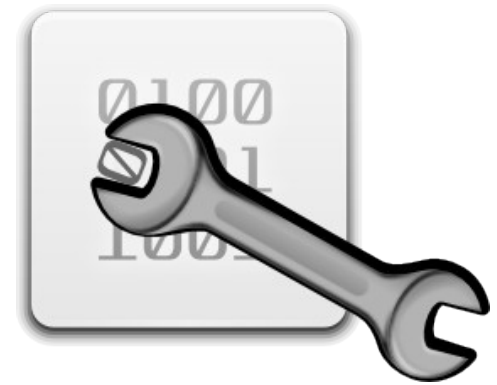


Podstawy inżynierii oprogramowania



Testowanie

Aleksander Lamża
ZKSB · Instytut Informatyki
Uniwersytet Śląski w Katowicach

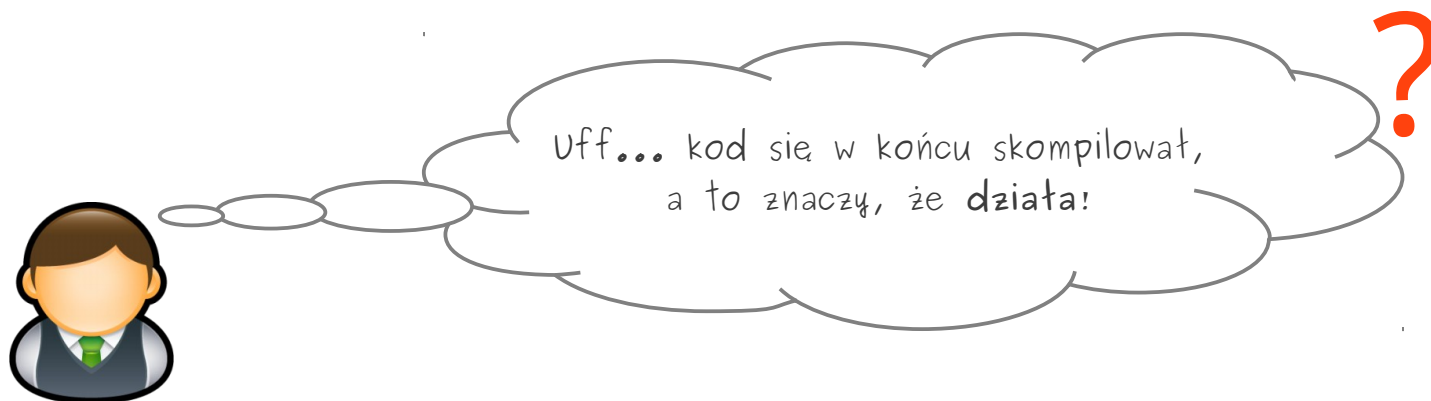
aleksander.lamza@us.edu.pl

- Cel testowania
- Ogólna klasyfikacja testów
- Testy czarnej, białej i szarej skrzynki
- Coś dla programistów – testy jednostkowe

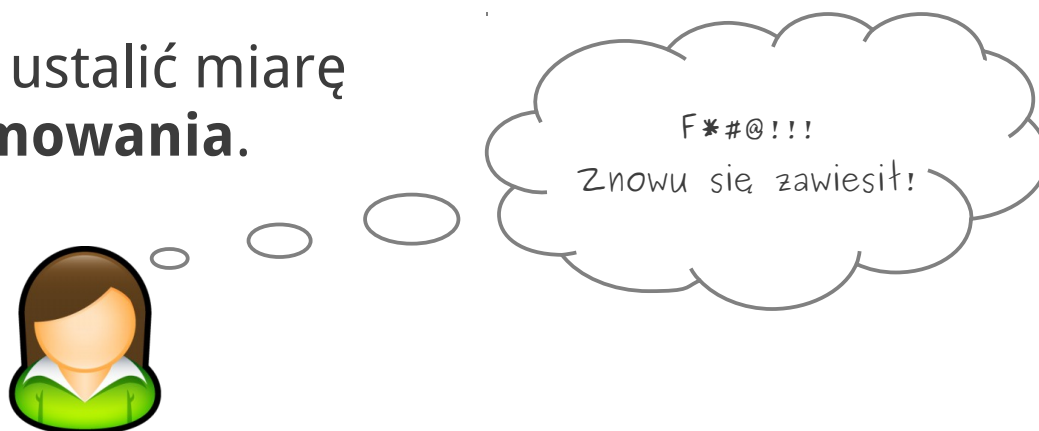
Co jest celem testowania?

Podstawowym celem przeprowadzania testów jest wykazanie, że **kod działa i jest zgodny z wymaganiami.**

Aby tego dokonać, trzeba **wykryć błędy** i je usunąć.



Niekiedy chcielibyśmy też ustalić miarę **niezawodności oprogramowania.**



Trzy problemy, które komplikują życie

Jeden błąd może prowadzić do wielu różnych błędnych wykonaniań.

Błąd to niepoprawna konstrukcja znajdująca się w kodzie.

Błędne wykonanie to nieprawidłowe działanie oprogramowania.

Te same błędne wykonania mogą być spowodowane różnymi błędami.

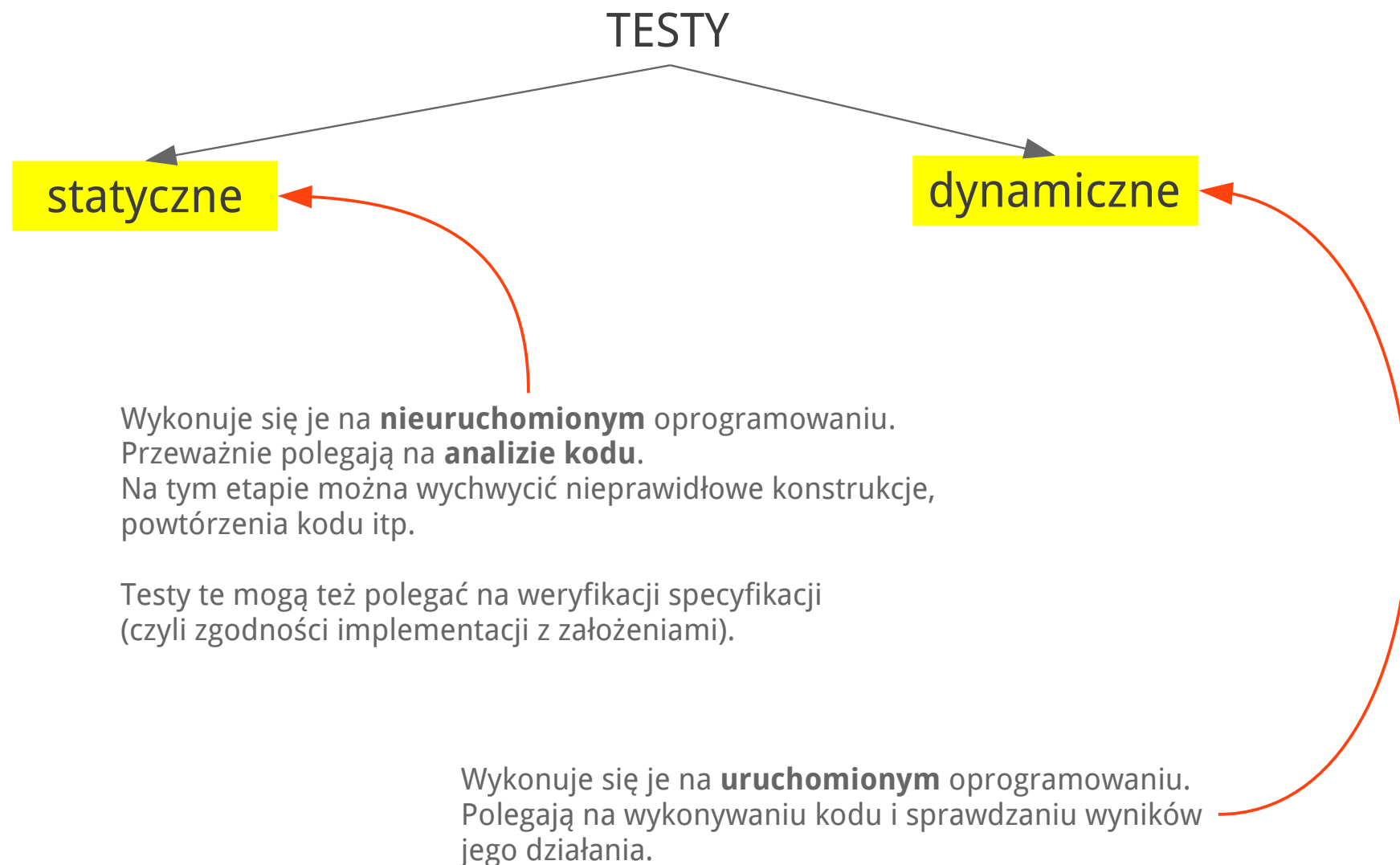
W wielu sytuacjach błąd może nie objawiać się błędnym wykonaniem.

I co?

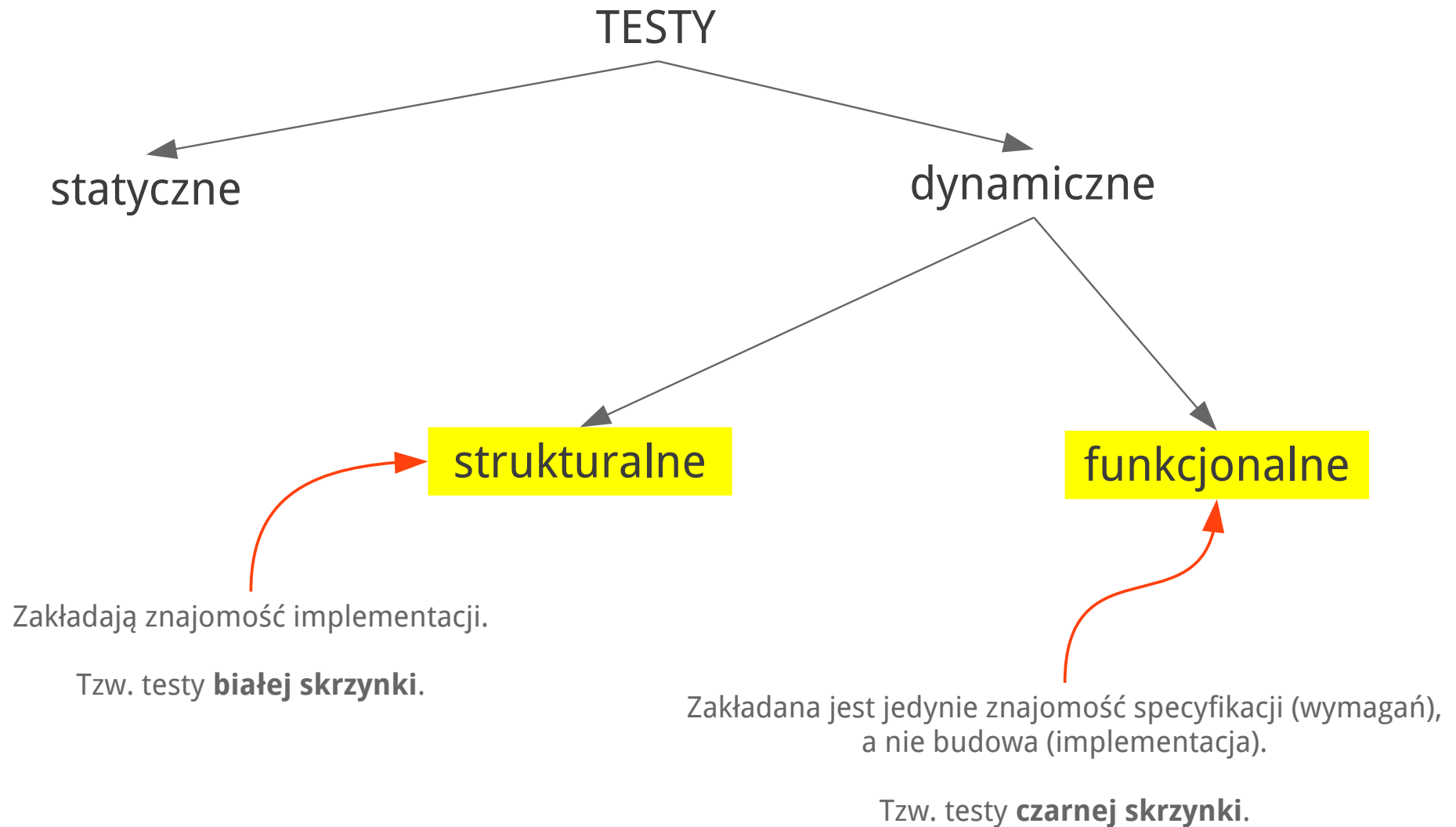
Jak w takim razie poradzić sobie z testowaniem?

Zacznijmy od sprawdzenia, co mamy do dyspozycji...

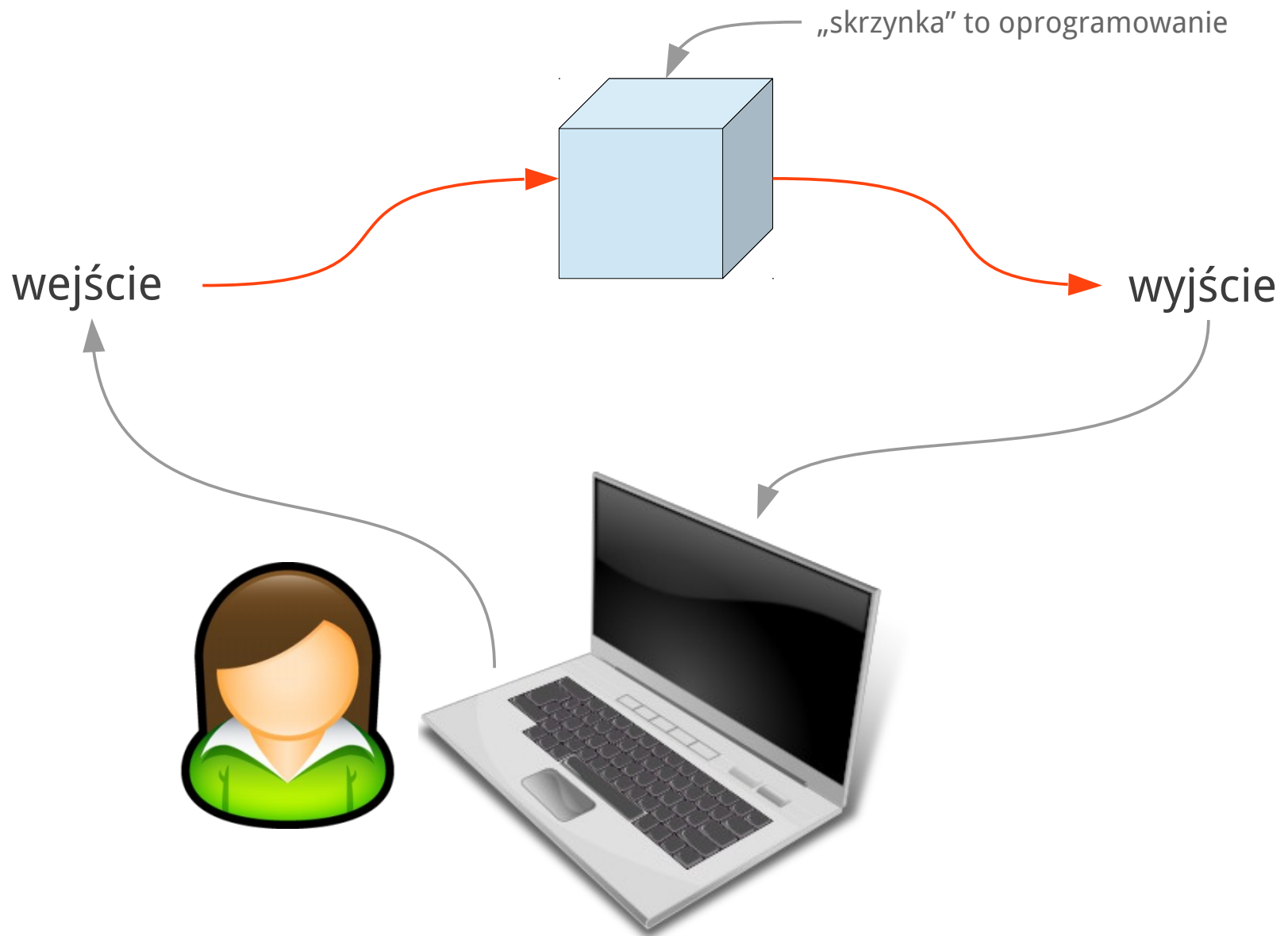
Ogólna klasyfikacja testów



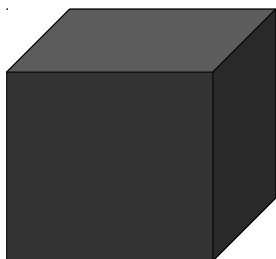
Ogólna klasyfikacja testów



O co chodzi ze skrzynkami?

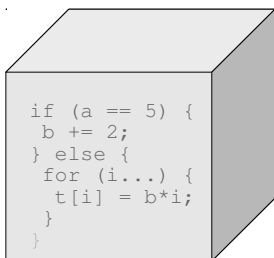


„Skrzynkowa” klasyfikacja testów



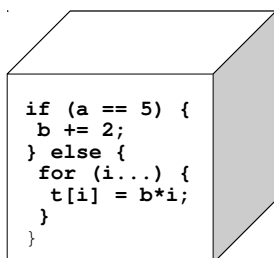
Testy czarnej skrzynki

Testujący nie znają (i najczęściej nie chcą znać) budowy programu.
Dla nich liczy się tylko działanie (funkcjonalność).



Testy szarej skrzynki

Testujący skupiają się na funkcjonalności, ale mają dostęp do wewnętrznych mechanizmów testowanego oprogramowania.

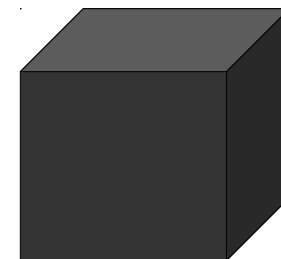


Testy białej skrzynki

Testujący (najczęściej programiści) znają budowę programu. Mają pełny wgląd w kod i to na nim się skupiają.
Dla nich również liczy się działanie, ale patrzą na to z innej perspektywy.

Testy czarnej skrzynki

Na co zwracać uwagę podczas testów czarnej skrzynki?






Czy program robi to, co zostało opisane w opowieściach użytkownika?

Czy po wprowadzeniu niepoprawnych danych program reaguje prawidłowo?

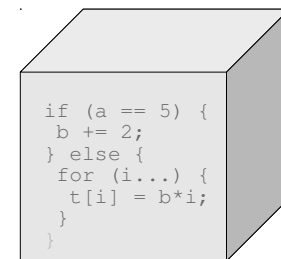
Czy wyniki operacji są zgodne z założeniami?

Przykłady testów czarnej skrzynki:

- testy akceptacyjne  Przeprowadzane np. na zakończenie każdej iteracji.
- testy integracyjne  Wykonuje się je po połączeniu modułów oprogramowania.
- testy systemu  Służą do sprawdzenia kompletnego oprogramowania.

Testy szarej skrzynki

Na co zwracać uwagę podczas testów szarej skrzynki?



Czy proces przetwarzania danych przebiega prawidłowo?

Niektóre dane nie są dostępne z poziomu interfejsu użytkownika i potrzebny jest dostęp np. do logów.

Czy dane przesyłane do innych systemów są prawidłowe?

Jeżeli oprogramowanie komunikuje się z innymi systemami, konieczne jest sprawdzenie poprawności przesyłanych danych (a to wymaga wglądu w wewnętrzne mechanizmy).

Czy oprogramowanie nie ma luk w zabezpieczeniach?

Z powodu błędów w implementacji może dochodzić do powstawania luk w zabezpieczeniach, które nie mogą być wykryte na wcześniejszych etapach testowania.

Przykładowe testy

Aktualizacja awataru w profilu użytkownika

Po kliknięciu awataru lub łącza „Zmień awatar” ma się pojawić okno wyboru pliku graficznego (JPG, PNG) o maksymalnym rozmiarze 200 kB. Po wybraniu pliku i potwierdzeniu wyboru plik ma zostać przesłany na serwer. Nowy awatar ma zostać wyświetlony na stronie profilu.

Przykładowa opowieść użytkownika dotycząca aktualizacji awataru na stronie profilu użytkownika.

Po **kliknięciu awataru lub łącza** „Zmień awatar” ma się pojawić okno wyboru pliku graficznego (**JPG, PNG**) o maksymalnym rozmiarze **200 kB**. Po wybraniu pliku i potwierdzeniu wyboru plik ma zostać **przesłany na serwer**. Nowy awatar ma zostać **wyświetlony na stronie profilu**.

Przykładowe przypadki testowe

Przykładowe scenariusze testów dla tej opowieści

Test 1. Wybranie małego pliku PNG

Na stronie profilu kliknij awatar. Kiedy pojawi się okno wyboru pliku, wybierz plik **mały.png** i kliknij przycisk **OK**.

1. Sprawdź, czy na stronie profilu pojawił się nowy awatar.
2. Sprawdź, czy na serwerze został zapisany nowy awatar (przeładuj stronę profilu).

Test 3. Anulowanie wyboru pliku

Na stronie profilu kliknij awatar. Kiedy pojawi się okno wyboru pliku, wybierz plik **mały.png** i kliknij przycisk **Anuluj**.

1. Sprawdź, czy na stronie profilu jest wyświetlany poprzedni awatar.
2. Sprawdź, czy na serwerze jest zapisany poprzedni awatar (przeładuj stronę profilu).

Test 2. Wybranie za dużego pliku PNG

Na stronie profilu kliknij awatar. Kiedy pojawi się okno wyboru pliku, wybierz plik **za-duży.png** i kliknij przycisk **OK**.

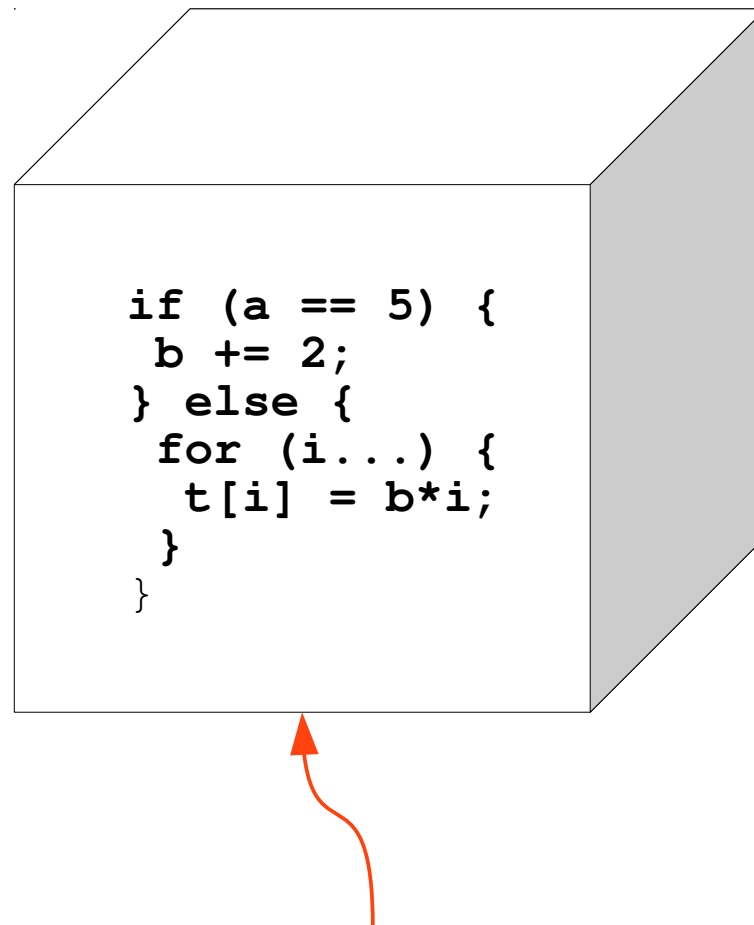
1. Sprawdź, czy pojawił się komunikat informujący o za dużym pliku.
2. Sprawdź, czy na stronie profilu jest wyświetlany poprzedni awatar.
3. Sprawdź, czy na serwerze jest zapisany poprzedni awatar (przeładuj stronę profilu).

Test 4. Wywołanie okna wyboru pliku za pomocą łącza „Zmień awatar”

Na stronie profilu kliknij **łącze „Zmień awatar”**.

1. Sprawdź, czy pojawiło się okno wyboru pliku.

Wracamy do skrzynek

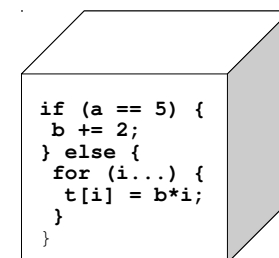


Została jeszcze **biała skrzynka**, zwana też **szklaną**.

Testy białej skrzynki

Co jest istotne w testach białej skrzynki?

Trzeba zadbać o jak najwyższe



pokrycie instrukcji

Instrukcja jest uznana za pokrytą, jeżeli jest wykonana co najmniej raz.

pokrycie rozgałęzień

Chodzi o odwiedzenie wszystkich gałęzi programu (if, switch).

pokrycie ścieżek

Brane są pod uwagę wszystkie możliwe ścieżki wykonania kodu.

pokrycie warunków

Dotyczy złożonych warunków logicznych, np. `if (a && b)`.

Pokrycie – przykład

```
delta = b*b - 4*a*c;  
if (delta < 0)  
    System.out.println("Delta ujemna");  
else  
    if (delta == 0)  
        System.out.println("X0=" + (-b / (2*a)));  
    else  
        System.out.println("X1=", + ((-b - sqrt(delta)) / (2*a)) ...);
```

Jakie sytuacje powinniśmy sprawdzić?

Testy białej skrzynki w praktyce – testy jednostkowe

Testy jednostkowe (ang. unit testing) służą do testowania elementarnych części składowych kodu.



(np. metod, klas, modułów)

Sposób testowania:

kod sprawdza kod



Kod testów jest rozwijany równolegle z kodem produkcyjnym.

Testy jednostkowe – narzędzia

Do przeprowadzania testów jednostkowych potrzebne jest odpowiednie narzędzie, tzw.

framework testów jednostkowych

Java – **JUnit**

C++ – **CppUnit**

C# – **NUnit**

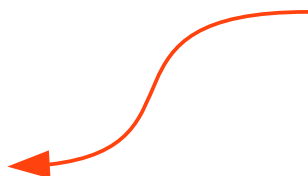
PHP – **PHPUnit**

Python – **PyUnit**

...

Przykłady frameworków dla różnych języków.

Obszerną listę dostępnych narzędzi można znaleźć np. w Wiki:
http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)



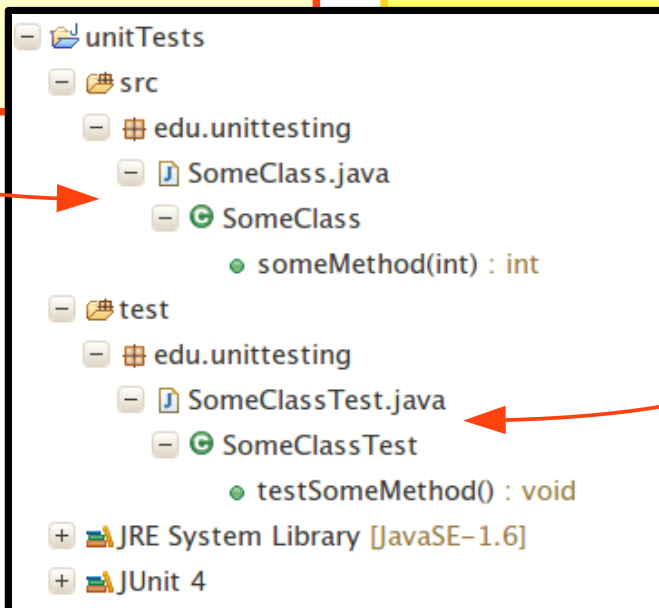
Prosty przykład testu jednostkowego

Kod produkcyjny (folder źródłowy: *src*)

```
class SomeClass {  
  
    ...  
  
    public int someMethod(int a) {  
        return ++a;  
    }  
  
    ...  
  
}
```

Kod testów (folder źródłowy: *test*)

```
class SomeClassTest {  
  
    ...  
  
    @Test  
    public void testSomeMethod() {  
        SomeClass o = new SomeClass();  
        assertEquals(13, o.someMethod(12));  
    }  
  
}
```



Składnia testu jednostkowego

```
@Test  
public void testSomeMethod() {  
    ...  
}
```

Adnotacja wskazująca metodę testującą.

Najzwyczajniejsza w świecie metoda.

Kod metody testującej.
Głównym jej elementem są **asercje**.

Adnotacje

@Test

@Test (expected = SomeException.class)

@Test (timeout = 200)

Umieszczenie tej adnotacji oznacza, że metoda ma być traktowana jako przypadek testowy.

@Before
@BeforeClass

Te adnotacje przydają się wtedy,
gdy przed uruchomieniem zestawu testów
(lub po jego uruchomieniu)
chcemy wykonać jakiś wspólny kod
(np. musimy utworzyć obiekty, pobrać dane itp.).

@After
@AfterClass

@Ignore

Adnotacja ta przydaje się wtedy, gdy
tymczasowo chcemy zignorować jakiś test.

Asercje

```
@Test
public void testSomeMethod() {
    SomeClass o = new SomeClass();
    assertEquals(13, o.someMethod(12));
}
```

Zakładamy, że wywołanie metody `someMethod()` obiektu `o` z argumentem `12` zwróci **wynik 13**.

assertEquals(expected, actual)
assertNull(object)
assertNotNull(object)
assertTrue(cond)
assertFalse(cond)
assertSame(expected, actual)
assertNotSame(expected, actual)
fail()

Każda asercja jako pierwszy (opcjonalny) argument może przyjąć komunikat.

Zasady tworzenia testów jednostkowych

O kod testów jednostkowych **należy dbać** tak samo jak o kod produkcyjny.

Kod testów musi być **zwięzły i czytelny**.



Jedna asercja na test

Jedna koncepcja na test

Jedna asercja na test

```
@Test
public void testSomeMethod() {
    SomeClass o = new SomeClass();
    assertEquals(32, o.someMethod(2, 5));
    assertEquals(1, o.someMethod(5, 0));
    assertEquals(0.5, o.someMethod(2, -1));
}
```



```
private SomeClass o;

@Before
public void setUp() {
    o = new SomeClass();
}

@Test
public void testSomeMethod() {
    assertEquals(32, o.someMethod(2, 5));
}

@Test
public void testSomeMethodZero() {
    assertEquals(1, o.someMethod(5, 0));
}

@Test
public void testSomeMethodNegative() {
    assertEquals(0.25, o.someMethod(4, -1));
}
```


Jedna koncepcja na test

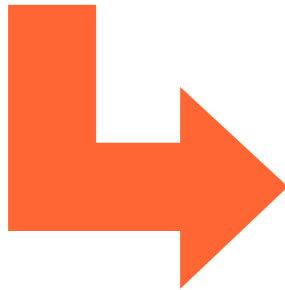
```
@Test
public void testAddMinutes() {
    Time t1 = new Time(10, 30);

    assertNotNull(t1);

    Time t2 = t1.addMinutes(15);
    assertEquals(10, t2.getHours());
    assertEquals(45, t2.getMinutes());

    Time t3 = t1.addMinutes(50);
    assertEquals(11, t3.getHours());
    assertEquals(20, t3.getMinutes());

    Time t4 = t1.addMinutes(-50);
    assertEquals(9, t4.getHours());
    assertEquals(40, t4.getMinutes());
}
```



```
private Time t;

@Before
public void setUp() {
    t = new Time(10, 30);
}

@Test
public void testTimeExists() {
    assertNotNull(t);
}

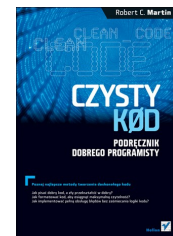
@Test
public void testAddMinutesSameHour() {
    Time tt = t.addMinutes(15);
    assertEquals(10, tt.getHours());
    assertEquals(45, tt.getMinutes());
}

@Test
public void testAddMinutesNextHour() {
    Time tt = t.addMinutes(50);
    assertEquals(11, tt.getHours());
    assertEquals(20, tt.getMinutes());
}

@Test
public void testAddMinutesPrevHour() {
    Time tt = t.addMinutes(-50);
    assertEquals(9, tt.getHours());
    assertEquals(40, tt.getMinutes());
}
```

Pięć zasad dobrych testów jednostkowych

Czysty kod. Podręcznik
dobrego programisty
R.C. Martin
Helion 2010
s. 151



F

jak **szybkie** (ang. fast).
Jeżeli przeprowadzanie testów trwa zbyt długo,
nie chce się tego robić.

I

jak **niezależne** (ang. independent).
Powinno być możliwe przeprowadzanie testów w dowolnej kolejności i konfiguracji.
Wprowadzenie zależności między testami ukrywa problemy i utrudnia ich diagnozę.

R

jak **powtarzalne** (ang. repeatable).
Testy powinny dać się przeprowadzić w każdym środowisku
i powinny dawać te same efekty.

S

jak **samosprawdzające** (ang. self-validating).
Testy powinny dawać jeden rezultat „tak-nie”.

T

jak **na czas** (ang. timely).
Testy powinny być pisane w odpowiednim momencie.

Jaki to jest „odpowiedni” moment?
Pełna odpowiedź innym razem.
Teraz powiem tylko, że chodzi o pisanie
testów **przed** napisaniem kodu
produkcyjnego.

Jakie wnioski?

TESTOWAĆ

TESTOWAĆ

I JESZCZE RAZ TESTOWAĆ

!