TDD
Test Driven Development

Dariusz Zbyrad

Czym jest testowanie?

Czemu znowu coś nie działa?
Czemu zmiany w programie tyle trwają?

Autor: Dariusz Zbyrad

Czym jest testowanie?

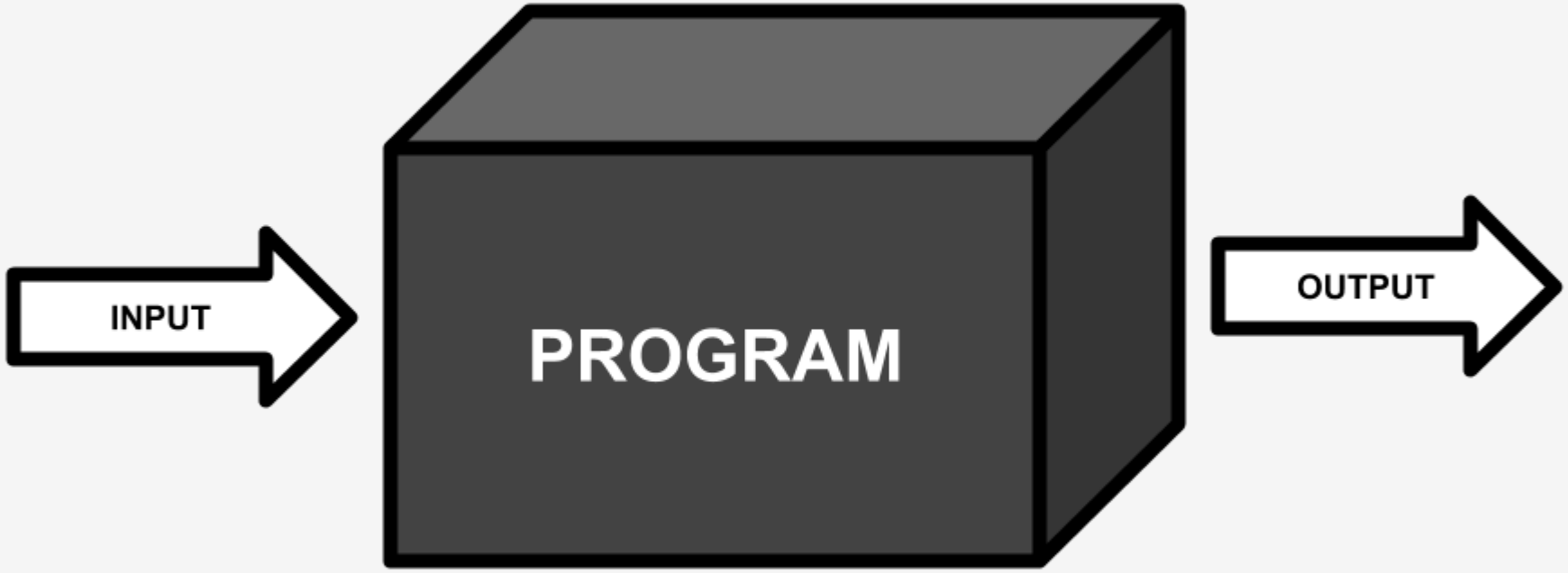
> *“**Testowanie** może ujawnić obecność błędów ale **nigdy ich braku***

Edsger Dijkstra

Rodzaje testów

- Testy jednostkowe
- Testy integracyjne
- Test end to end (E2E)

Metoda czarnej skrzynki – testy funkcjonalne



Autor: Dariusz Zbyrad

Metoda czarnej skrzynki – testy funkcjonalne

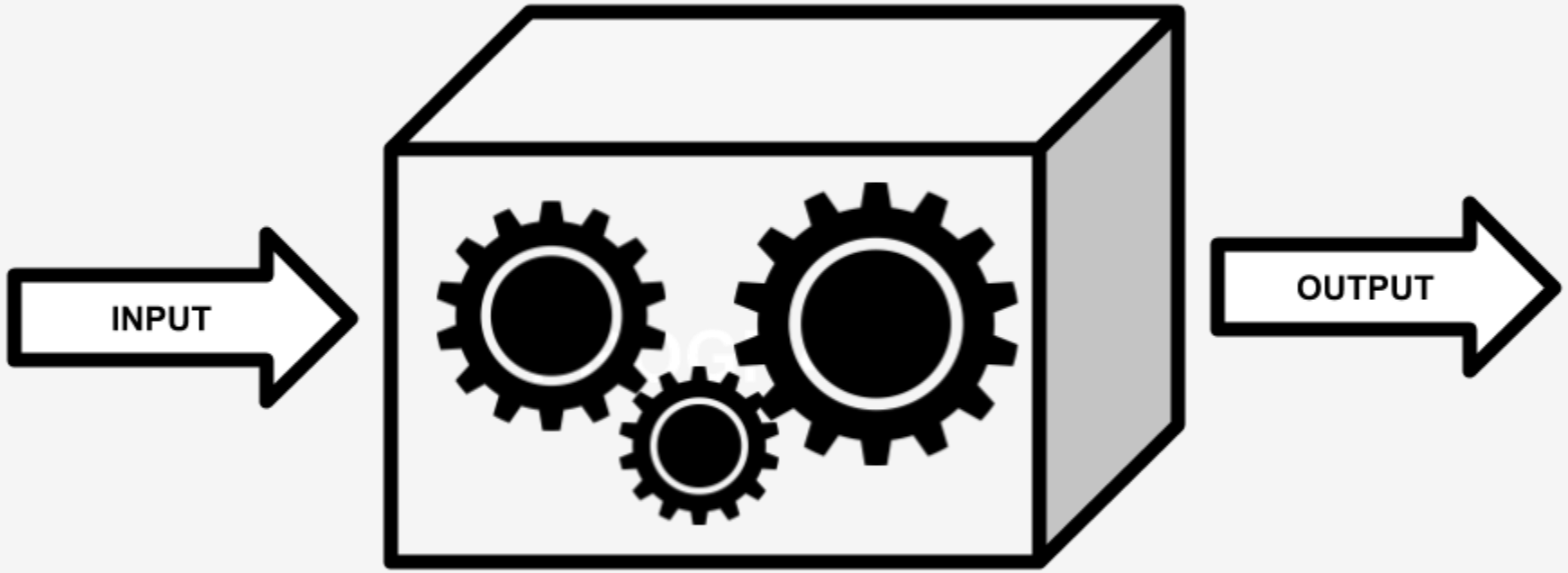
Zalety:

- Pozwalają wydajnie sprawdzać duże bloki kodu.
- Nie wymagają dostępu do kodu, zrozumienia kodu i umiejętności programowania.
- Pozwalają oddzielić perspektywę użytkownika od perspektywy programisty.

Wady:

- Nie da się dokładnie określić pokrycia kodu testami, ponieważ tester ma ograniczoną wiedzę na temat aplikacji.
- Zapewniają ograniczone pokrycie kodu testami, ponieważ wykonywana jest tylko część scenariuszy testowych.

Metoda białej skrzynki – testy strukturalne



Autor: Dariusz Zbyrad

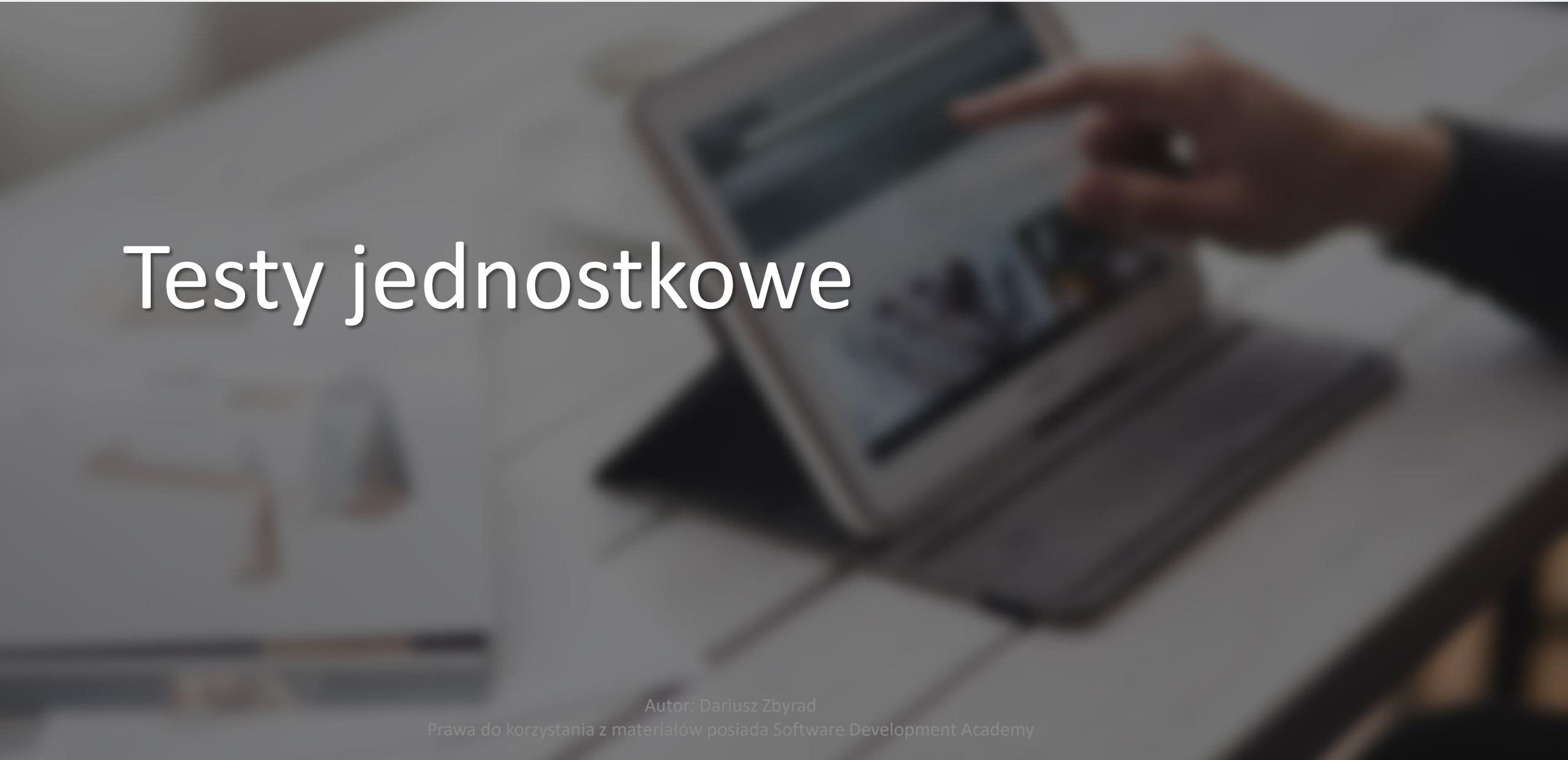
Metoda białej skrzynki – testy strukturalne

Zalety:

- Są wydajne, jeśli chodzi o wyszukiwanie błędów i problemów.
- Umożliwiają wykrywanie ukrytych błędów.
- Umożliwiają zrozumienie kodu programistom.
- Pomagają optymalizować kod.

Wady:

- Mogą nie wykryć niezaimplementowanych, brakujących funkcji.
- Wymagają dostępu do kodu.

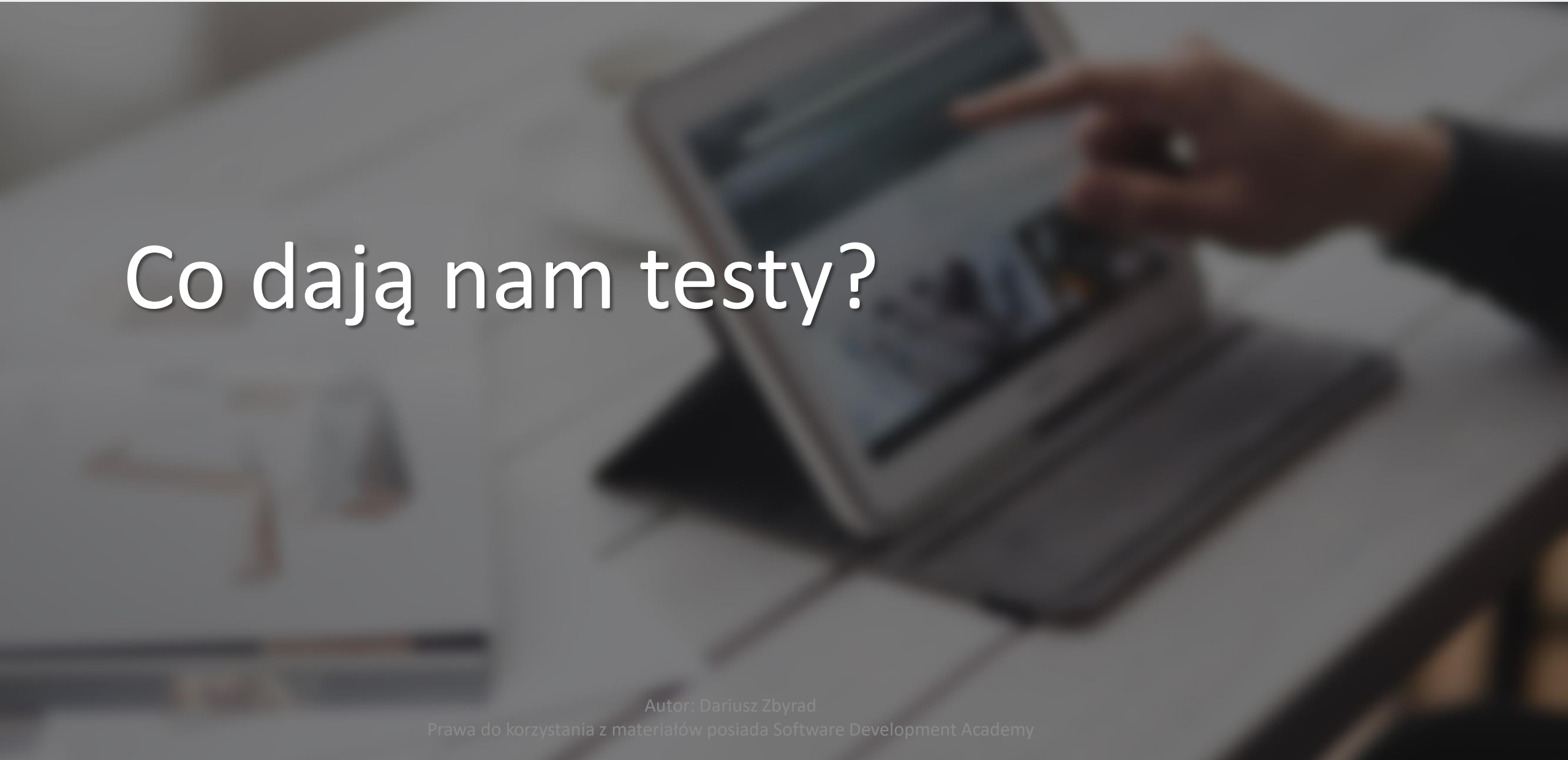


Testy jednostkowe

Autor: Dariusz Zbyrad

Testy jednostkowe

- Testują elementarne części aplikacji: klasy, metody
- Pozwalają szybko wykryć błędy implementacyjne
- Generują raport pozwalający szybko znaleźć miejsce błędu
- Szybkie, automatyczne i niezawodne



Co dają nam testy?

Autor: Dariusz Zbyrad

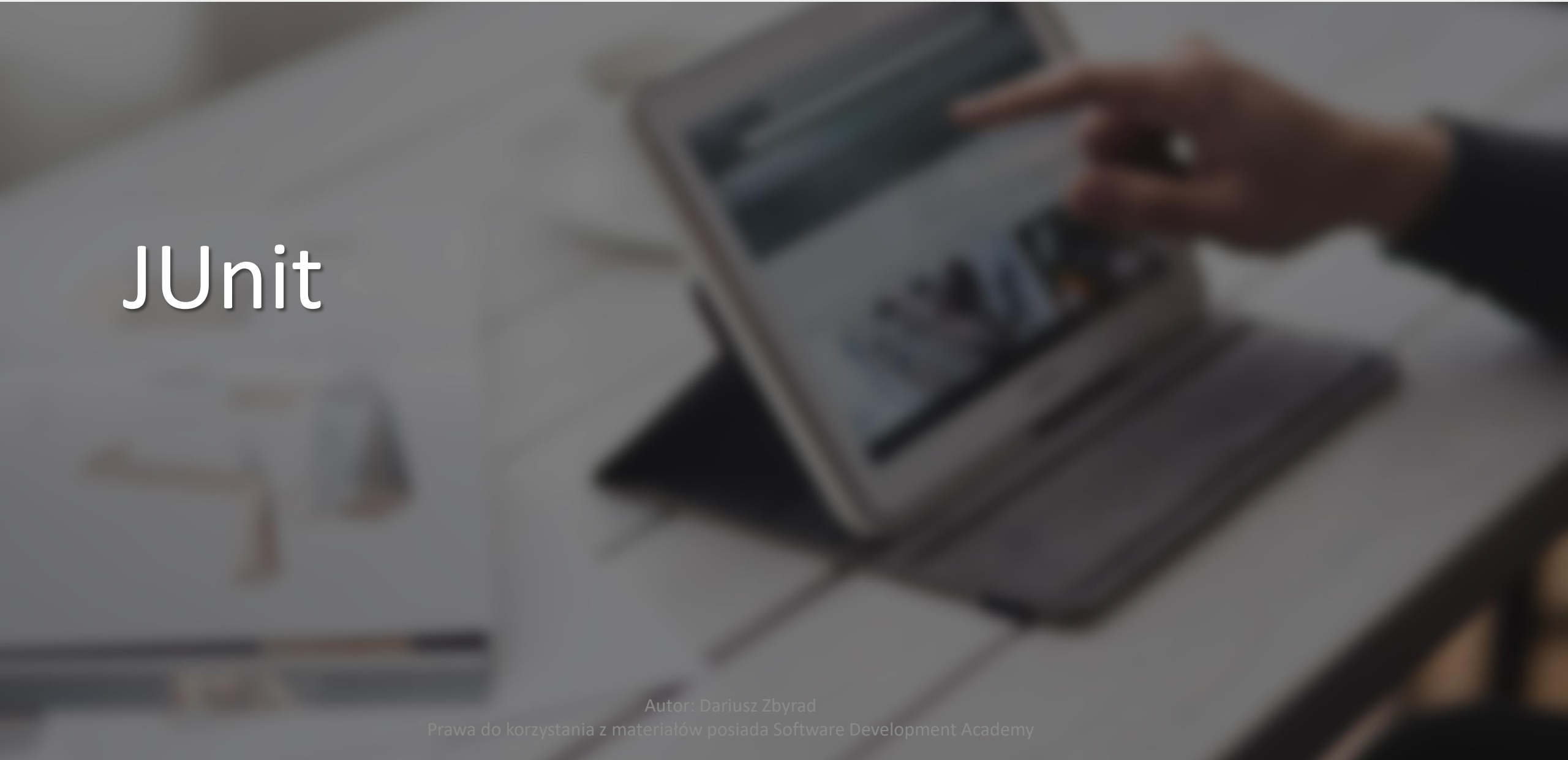
Redukcja czasu

Dokumentacja

Autor: Dariusz Zbyrad

Skuteczniejszy refaktoring

Autor: Dariusz Zbyrad



JUnit

Autor: Dariusz Zbyrad

JUnit

- Framework - steruje wykonaniem programu
- Zaawansowane narzędzie do pisania testów jednostkowych
- Wspierany przez IDE
- Raporty z wykonanych testów
- Oddzielenie testów od kodu

JUnit - czym jest test?

- Test to **metoda** w **klasie** testującej
- Test jest samoweryfikujący - wynik jednego testu nie może zależeć od wyniku innych testów
- Jedna klasa testująca - jedna klasa testowana

Konwencje nazewnictwa

```
@Test
public void shouldReturn50WhenAdding30And20() {
        ...
}

@Test
public void testReturn50WhenAdding30And20() {
        ...
}
```

Struktura metody testującej

```
@Test
public void shouldReturn50WhenAdding30And20() {
  // given

  // when

  // then

}
```

Przygotowujemy dane do testu

```
@Test
public void shouldReturn50WhenAdding30And20() {
  // given
  Calculator calculator = new Calculator();
  // when

  // then

}
```

Wywołujemy logikę, którą chcemy przetestować

```
@Test
public void shouldReturn50WhenAdding30And20() {
  // given
  Calculator calculator = new Calculator();
  // when
  int result = calculator.add(30, 20);
  // then

}
```

Weryfikujemy wynik

```
@Test
public void shouldReturn50WhenAdding30And20() {
  // given
  Calculator calculator = new Calculator();
  // when
  int result = calculator.add(30, 20);
  // then
  assertEquals(50, result);
}
```

Asercje

- Weryfikują rezultat testu
- Zwracają błąd gdy test się nie powiedzie
- Ogólna reguła: unikamy ścian asercji - jeden test, jedna funkcjonalność

Asercje - przykłady

Wartość oczekiwana

```
assertEquals(50, result);

assertEquals(2, 2);

assertEquals(4, 4, "The optional assertion message");
```

Asercje - przykłady

```
assertAll("person",
        () -> assertEquals("John", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName()),
        () -> assertEquals(28, person.getAge()),
);
```

Asercje - przykłady

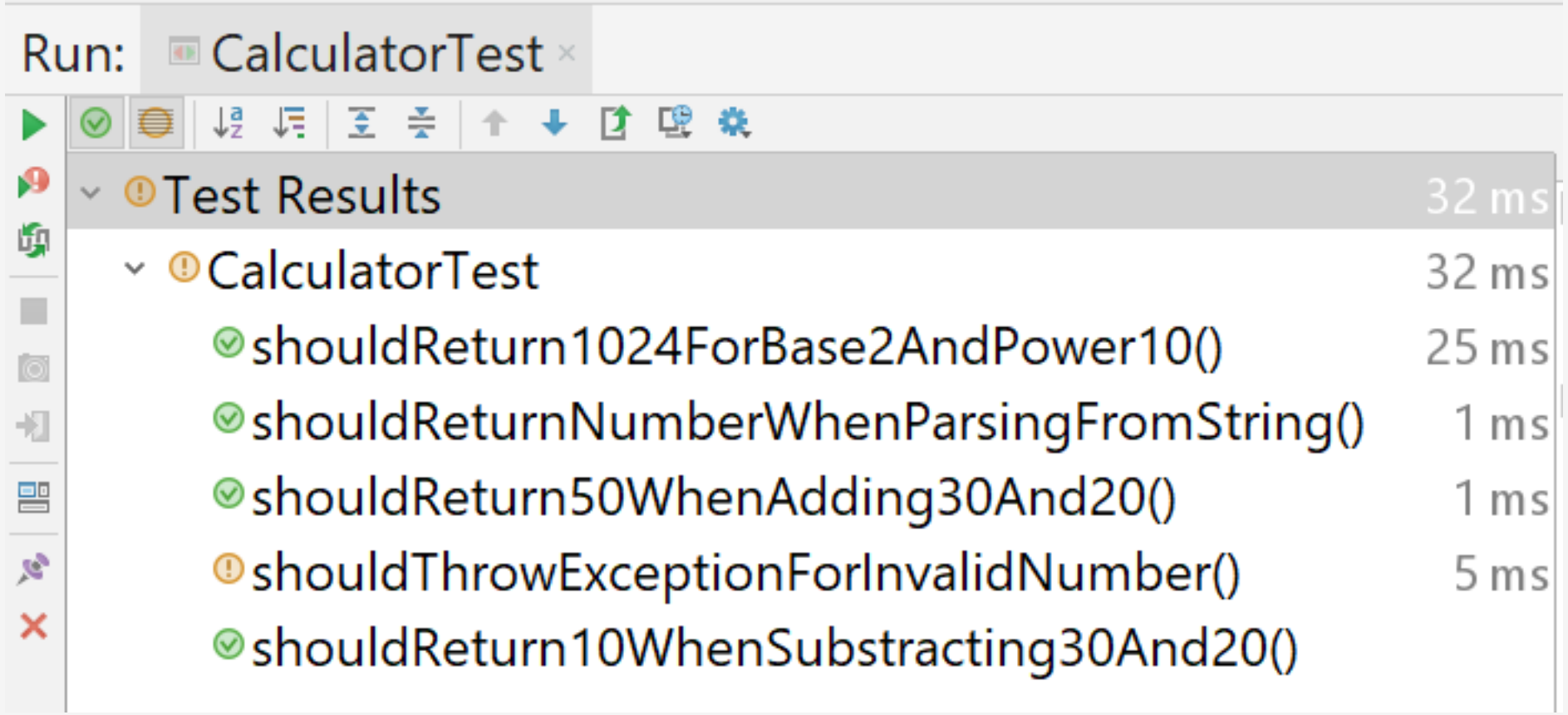
```
assertThrows(NumberFormatException.class, () -> calculator.parse("1dd23"));
```

```
NumberFormatException exc = assertThrows(NumberFormatException.class,
      () -> calculator.parse("1dd23"));

assertEquals("Number not valid", exc.getMessage());
```

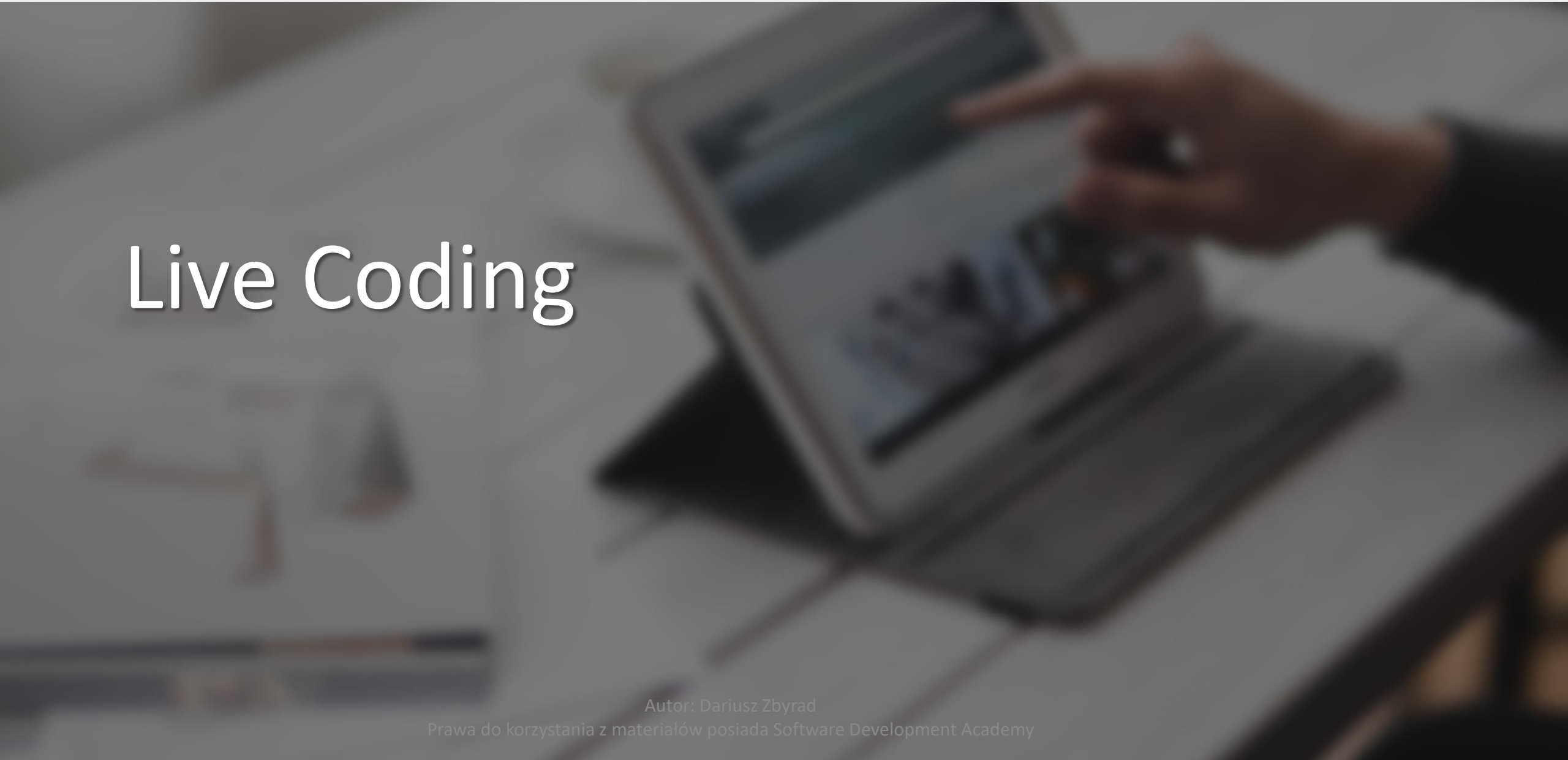
Asercje - przykłady

```
assertArrayEquals(new byte[] {1, 2, 3}, new byte[] {4, 5, 6});

assertTrue(result);

assertFalse(result);
```

Dlaczego nazwa testu jest ważna?



Autor: Dariusz Zbyrad



Live Coding

Autor: Dariusz Zbyrad

Adnotacje

`@Test`	Określa metodę jako testową
`@ParameterizedTest`	Określa test parametryczny
`@RepeatedTest`	Test powtarzalny - należy określić ilość powtórzeń
`@DisplayName`	Nazwa testu w formie dowolnej (w odróżnieniu od nazwy metody)
`@BeforeEach`	Metoda uruchamiana przed każdą metodą `@Test`
`@AfterEach`	Analogicznie do `@BeforeEach`
`@BeforeAll`	Metoda uruchamiana przed całą klasą testową
`@AfterAll`	Analogicznie do `@BeforeAll`

Autor: Dariusz Zbyrad

Adnotacje cd.

`@Tag`	Pozwala grupować testy i uruchamiać każdą z grup oddzielnie
`@Disabled`	Wyłącza test (równoznaczne z usunięciem `@Test`)

Ćwiczenie

- Zapoznaj się z zadaniem „Fizz Buzz” na stronie https://en.wikipedia.org/wiki/Fizz_buzz
- Napisz program, który wypisze na ekranie sekwencje „Fizz Buzz” dla liczb od 1 do 100
- Napisz odpowiednie testy jednostkowe

Testy parametryczne

- Ten sam wynik testu dla różnych wartości
- Uruchomienie tego samego testu z różnymi danymi wejściowymi
- Adnotacja `@ParametrizedTest`
- Dodatkowo w adnotacji podajemy źródło danych wejściowych
  - `@ValueSource`
  - `@MethodSource`
  - `@CsvSource`
  - …

Testy parametryczne

```java
@ParameterizedTest
@ValueSource(ints = {15, 30, 45})
public void shouldReturnFizzBuzzForNumberDivisibleByThreeAndFive(int param) {
    //given
    String expectedSeq = "Fizz Buzz";
    Generator generator = new Generator();

    //when
    String seq = generator.generateSeq(param, param);

    //then
    assertEquals(expectedSeq, seq);
}
```

Ćwiczenie JUnit

Dodaj testy parametryzowane do poprzedniego ćwiczenia

Czy powinniśmy testować wszystko?

Autor: Dariusz Zbyrad

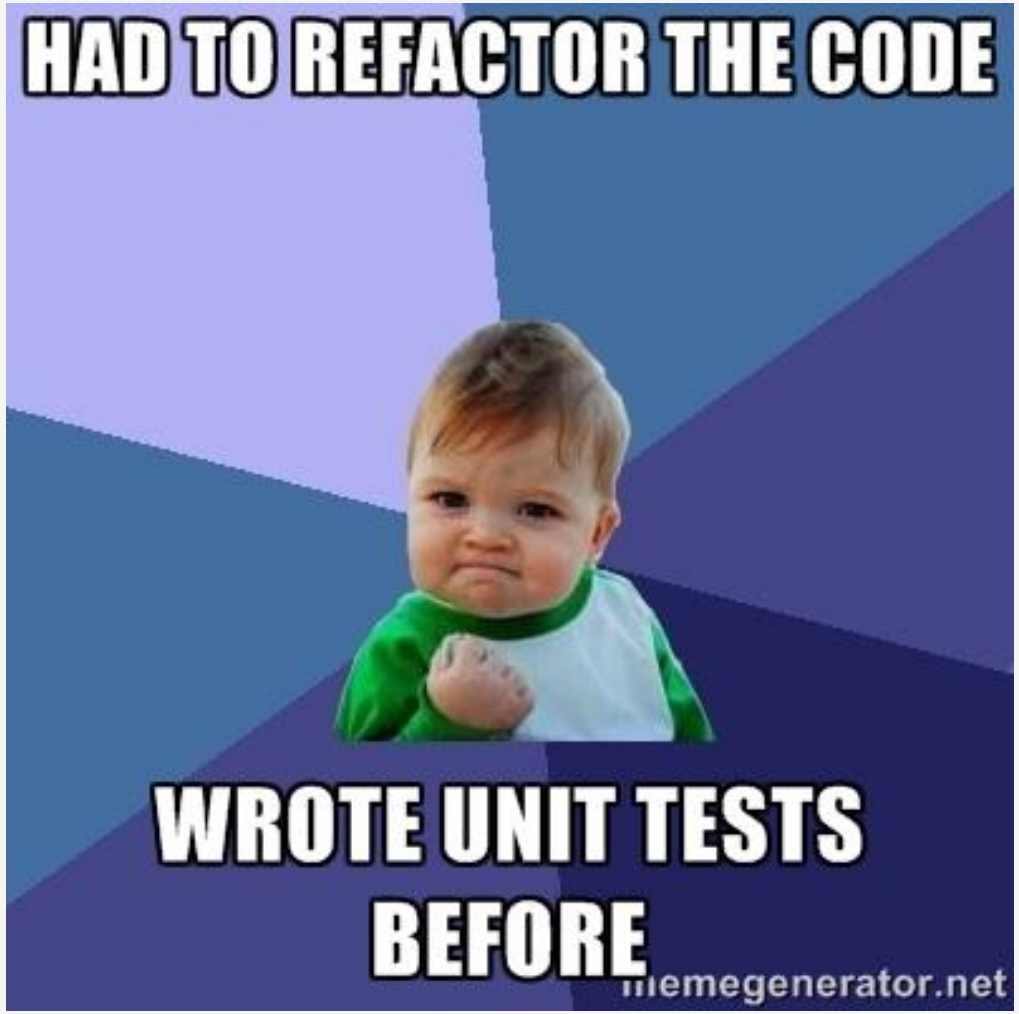
Czy powinniśmy testować wszystko?

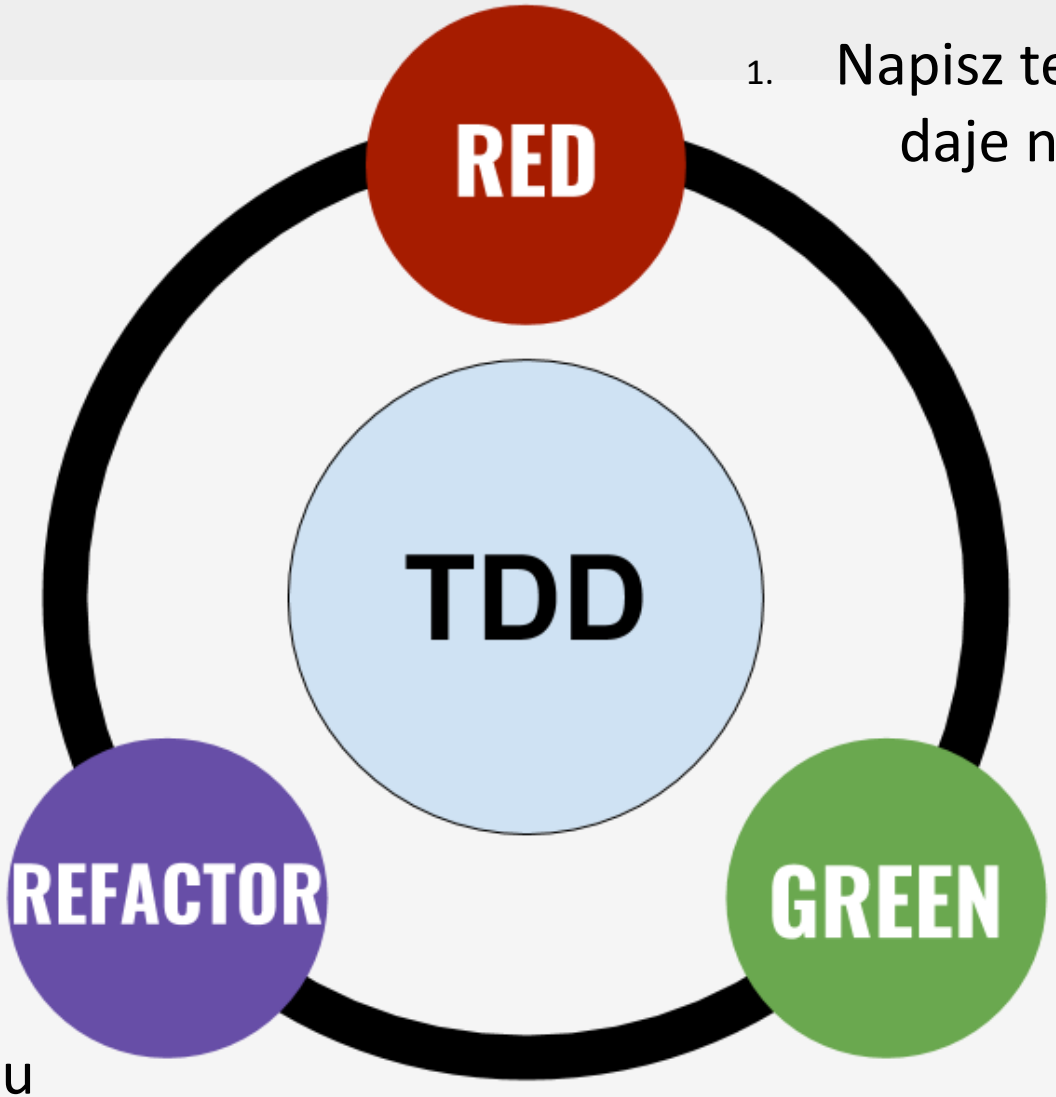
- Nie ma sensu testować banalnego kodu np. **setterów** i **getterów**, powinny być one testowane „przy okazji”
- Nie ma potrzeby testowania zewnętrznych bibliotek (najprawdopodobniej zostały już przetestowane)
- Testować powinniśmy to co napisaliśmy sami, złożone algorytmy, kluczowe funkcjonalności

Autor: Dariusz Zbyrad

F.I.R.S.T

- **F** ast – relatywnie szybkie
- **I** ndependent – niezależne od środowiska
- **R** epeatable – za każdym razem daje te same wyniki
- **S** elf validating – nie jest wymagana kontrola
- **T** imely – pisane w odpowiednim momencie





1. Napisz test, który daje negatywny wynik
2. Zaimplementuj funkcjonalność aby test dawał pozytywny wynik
3. Popraw czytelność kodu

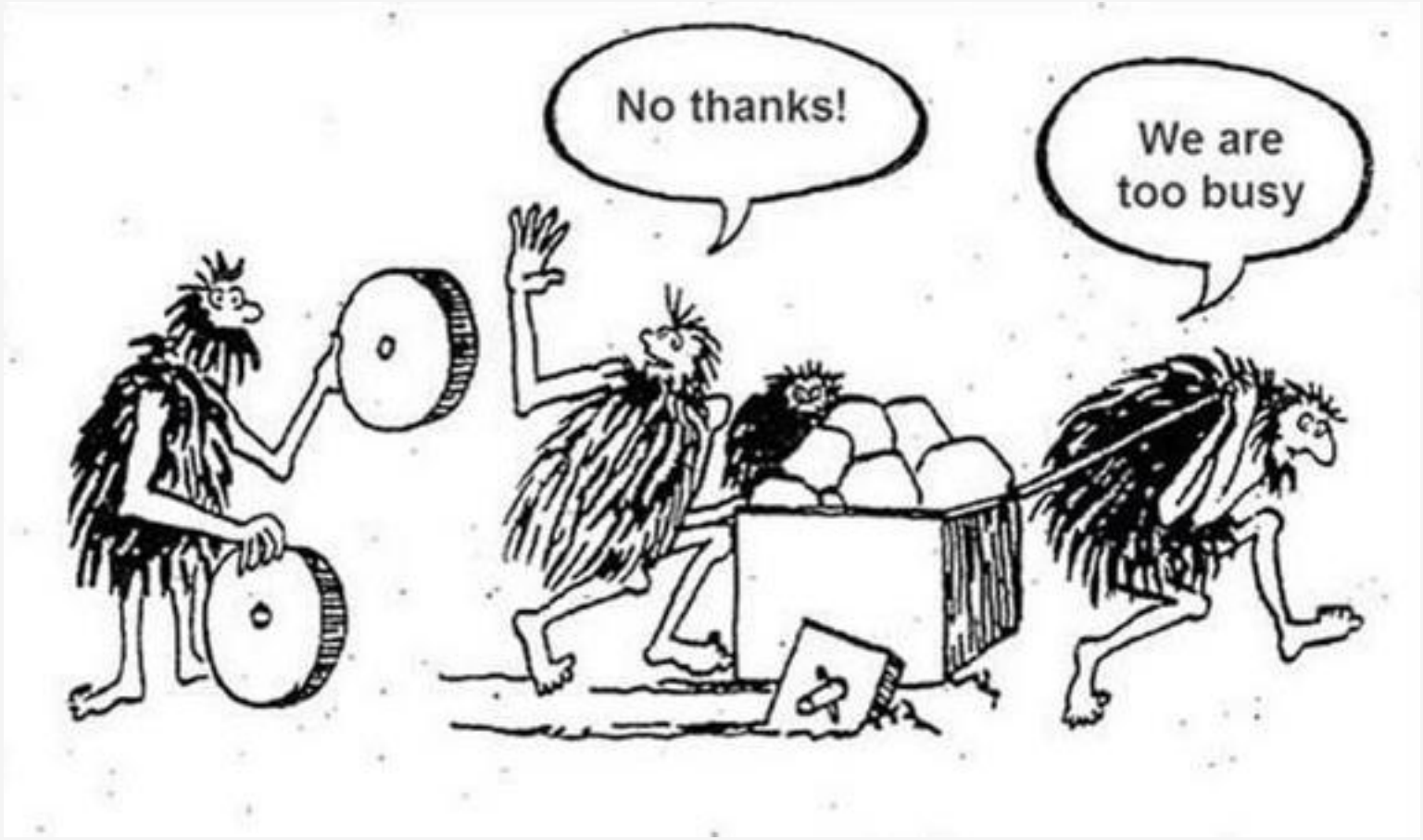
Test Driven Development

- Wywodzi się od Test-First, jednego z konceptów **programowania ekstremalnego**
- Czystszy i bardziej przejrzysty kod
- Tylko kod niezbędny do zaliczenia testów
- **YAGNI** – You ain’t gonna need it

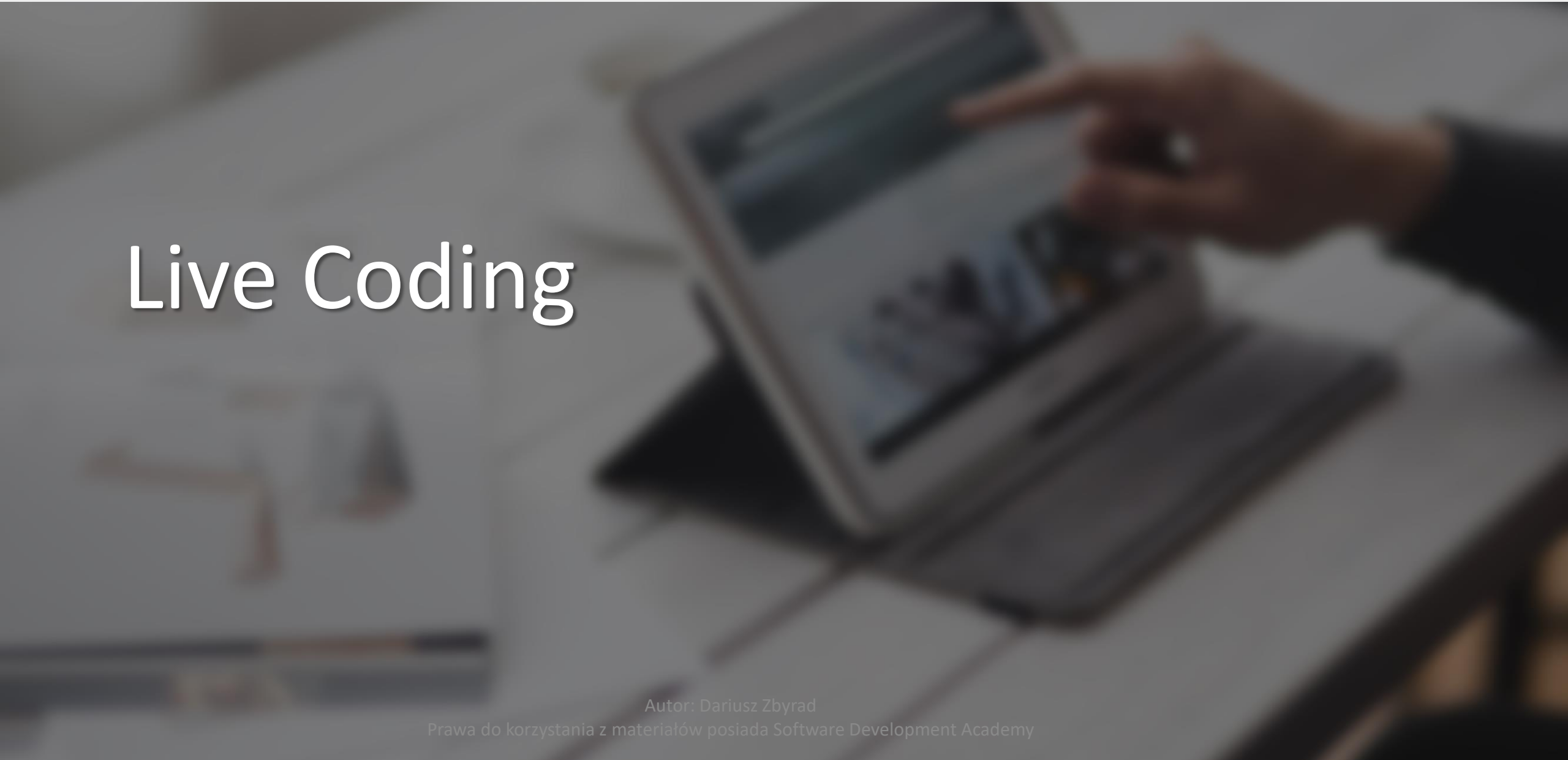
Test Driven Development

- Ułatwia zrozumienie wymagań
- Zapewnia skuteczność kodu testowego
- Podnosi jakość programowania
  - Dobry kod, to taki, który łatwo jest przetestować
- Podnosi produktywność w dłuższym wymiarze czasu - redukcja **długu technicznego**

Dług techniczny



https://medium.com/isovera/the-silent-website-killer-technical-debt-2c582896c6db



Live Coding

Autor: Dariusz Zbyrad

Ćwiczenie

Program powinien umożliwiać dodawanie znajomości pomiędzy wieloma osobami, gdzie przez osobę rozumiemy wyłącznie jej imię.

- Jako użytkownik mogę dodać znajomych do siebie, nawet jeżeli te osoby nie istnieją jeszcze w bazie
- Jako użytkownik mogę pobrać liczbę znajomych jaką posiada inna osoba
- Jako użytkownik mogę pobrać listę znajomych innej osoby (na początku zwraca null)
- Jako użytkownik mogę sprawdzić czy dwie osoby są znajomymi (na początku relacja jednostronna)

Zaimplementuj wymagania wykorzystując podejście TDD

Mockito

Autor: Dariusz Zbyrad

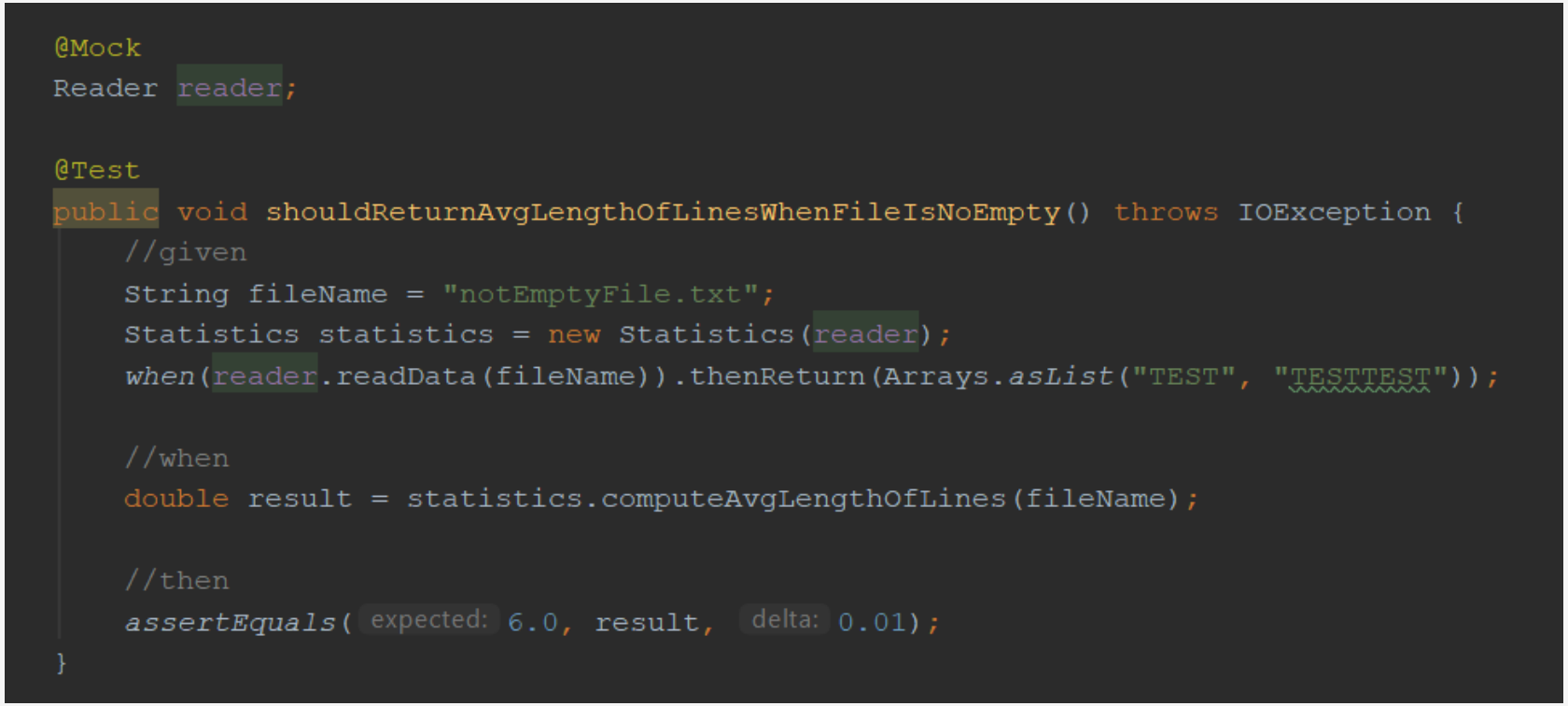
Atrapa obiektu

- Symuluje działanie prawdziwego obiektu
- Programista ma pełną kontrolę nad działaniem atrapy
- Atrapa nie wywołuje ciężkiej logiki prawdziwego obiektu
- Pomaga zachować zasadę Isolated z F.I.R.S.T

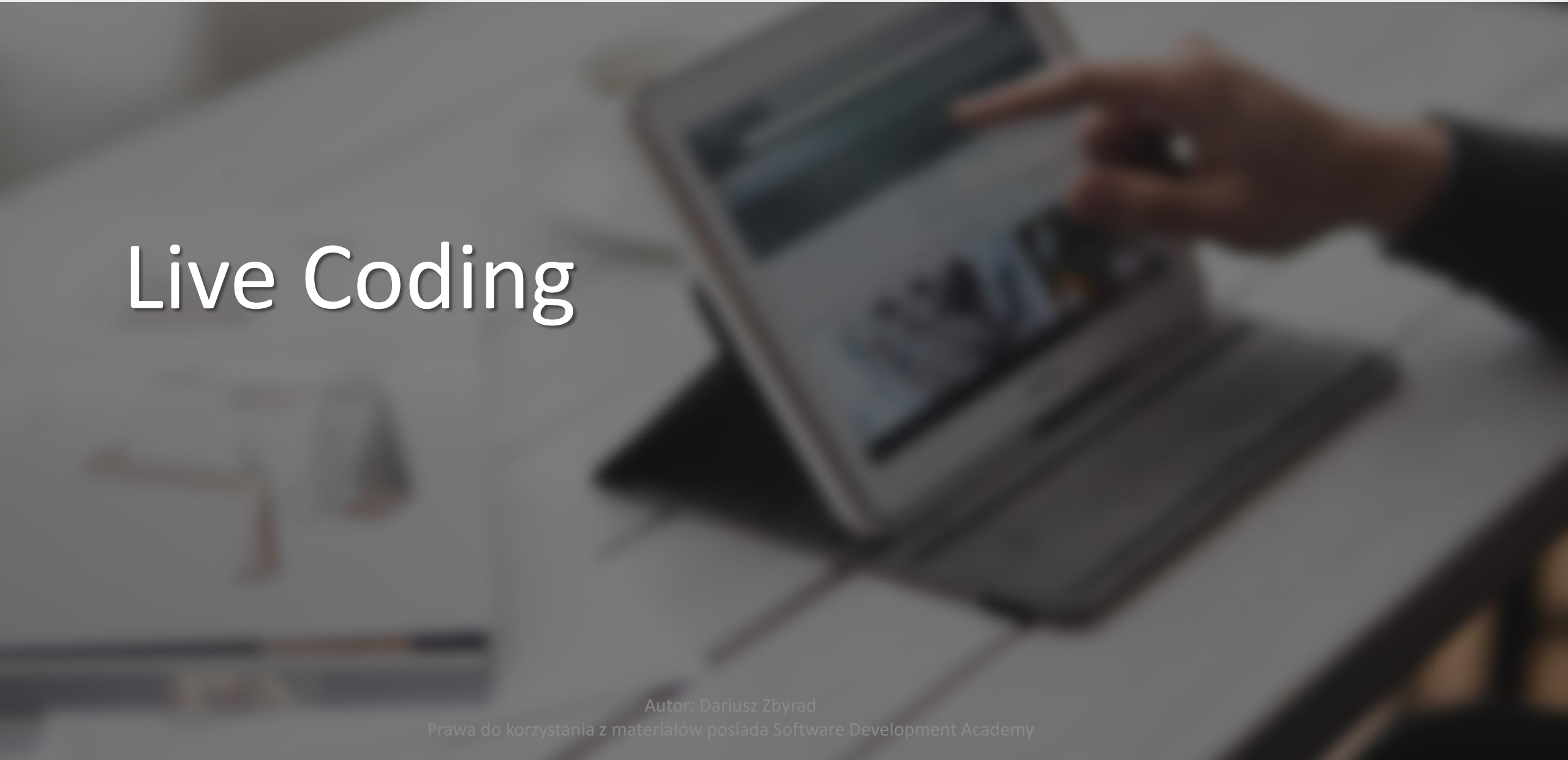
Jak to działa?

- Generowany jest byte code w trakcie działania programu (**runtime**)
- Wygenerowany byte code to klasa, która **implementuje** lub **rozszerza** dany interfejs/klasę
- Implementowana metoda po prostu zwraca oczekiwany wynik

Mokowanie - przykład



```java
@Mock
Reader reader;

@Test
public void shouldReturnAvgLengthOfLinesWhenFileIsNoEmpty() throws IOException {
    //given
    String fileName = "notEmptyFile.txt";
    Statistics statistics = new Statistics(reader);
    when(reader.readData(fileName)).thenReturn(Arrays.asList("TEST", "TESTTEST"));

    //when
    double result = statistics.computeAvgLengthOfLines(fileName);

    //then
    assertEquals( expected: 6.0, result,  delta: 0.01);
}
```



Live Coding

Autor: Dariusz Zbyrad

Dodatkowe materiały

Niewielki przykład wykorzystania Mockito
https://chlebik.wordpress.com/2012/10/02/testng-mockito-tdd-receptury/

Kompleksowy tutorial Mockito
https://www.tutorialspoint.com/mockito/index.htm

Kompleksowy tutorial JUnit
https://www.tutorialspoint.com/junit/

Najpopularniejszy tutorial Mockito na youtubue
https://www.youtube.com/playlist?list=PLe_YI35xeGcCk8vFU1LBoPTng4imSmWAi

Trochę krótszy tutorial Mockito na przykładzie
https://examples.javacodegeeks.com/core-java/mockito/mockito-tutorial-beginners/

Dobry przykład i zadanie TDD
https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/