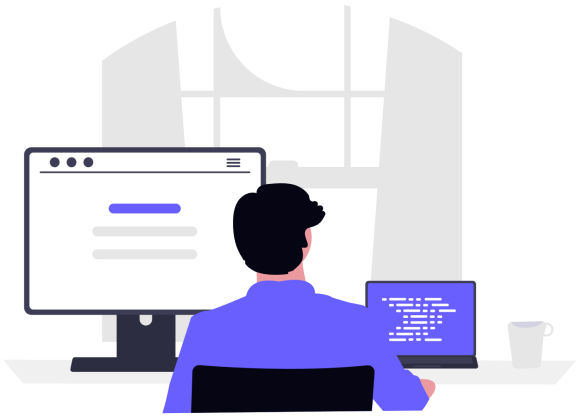
[SK] OOP 4 - Testovanie softvéru

Obsah



Čo je testovanie softvéru

Testovanie softvéru je **proces**

- V skratke: **proces** overovania, či softvér funguje podľa očakávania.
- “Definícia”: **proces** vykonávania programu alebo systému s cieľom odhaliť chyby a overiť, že softvér spĺňa špecifikované požiadavky.

Cieľ testovania:

- **odhaľovanie chýb (defektov)** - hľadanie problémov, ktoré by mohli spôsobiť nesprávne správanie softvéru
- **overenie požiadaviek** - kontrola, či softvér robí to, čo sa od neho očakáva
- **zabezpečiť kvalitu**

Testovanie môže byť:

- manuálne
- automatizované

Prečo testujeme softvér

Softvér obsahuje chyby (bugs).

Testovanie pomáha tieto chyby nájsť ešte predtým, než sa dostanú k používateľovi.

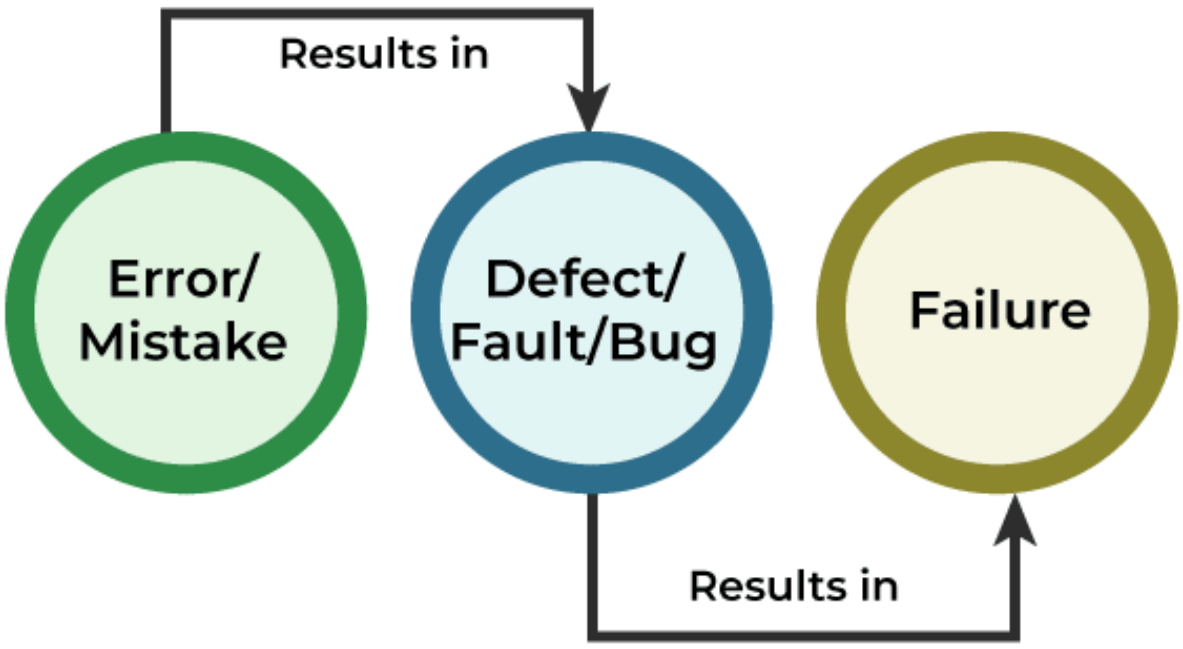
Hlavné dôvody:

- zlepšenie kvality softvéru
- zníženie rizika chýb v produkcii
- jednoduchšie refaktorovanie kódu
- rýchlejšia spätná väzba pri vývoji

📌 **Zaujímavosť:**

Oprava chyby v produkcii môže byť až **niekoľkokrát drahšia** ako počas vývoja.

Chyba - defekt - zlyhanie



Zdroj: https://www.geeksforgeeks.org/software-engineering/software-engineering-differences-between-defect-bug-and-failure/

Každý krok ma vyššiu závažnosť. Defekt v kóde môže spôsobiť chybu a následne zlyhanie systému.

Príklad chyby

Jednoduchá funkcia:

```
int divide(int a, int b) {
    return a / b;
}
```

Čo sa stane keď:

```
divide(10, 0)
```

➡️ **ArithmeticException**

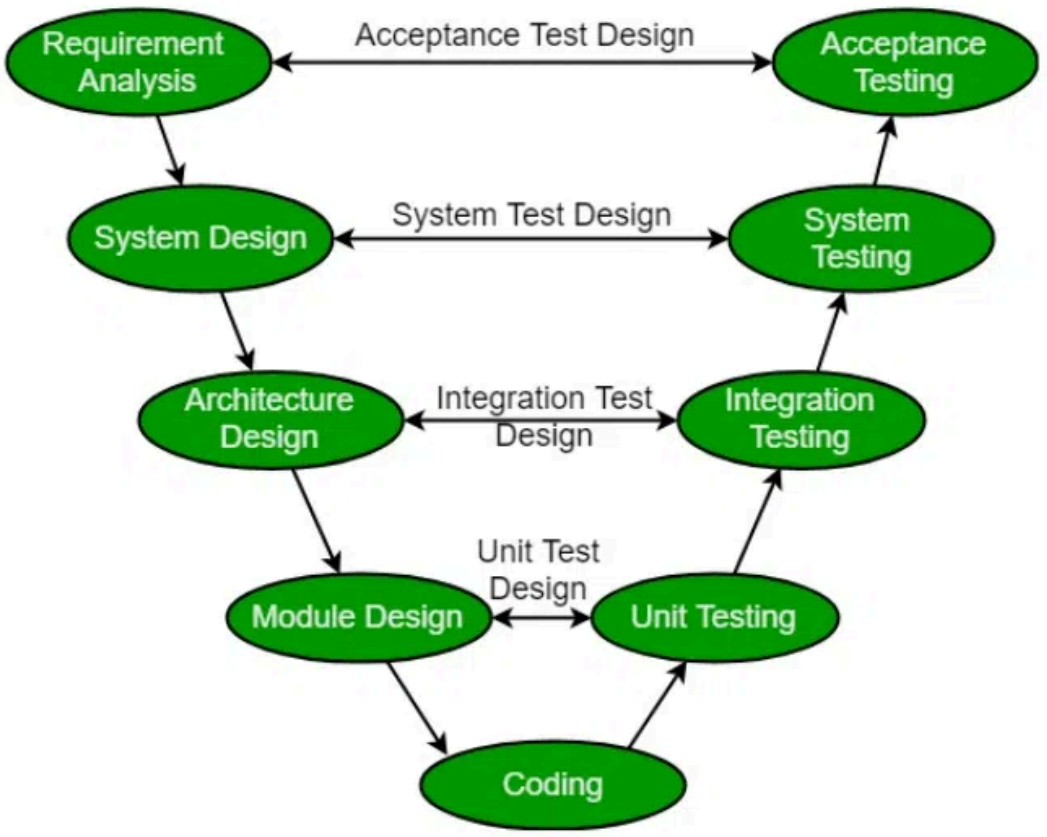
Testovanie pomáha takéto prípady odhaliť.

Takéto chyby sa často objavia až pri špecifických vstupoch.

V-model

V-model znázorňuje vzťah medzi fázami vývoja softvéru a jednotlivými úrovňami testovania.

- ľavá strana predstavuje **analýzu a návrh systému**
- spodná časť predstavuje **implementáciu (coding)**
- pravá strana predstavuje **testovanie**



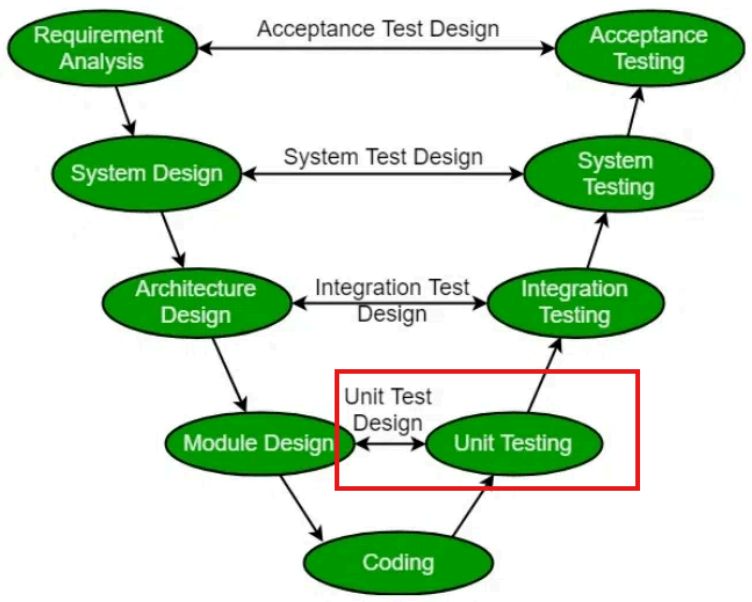
Zdroj: https://www.geeksforgeeks.org/software-engineering/software-engineering-sdlc-v-model/

V-model

Každá fáza návrhu má svoju zodpovedajúcu fázu testovania:

- **Module Design → Unit Testing**
- **Architecture Design → Integration Testing**
- **System Design → System Testing**
- **Requirements → Acceptance Testing**

📌 Testovanie sa plánuje už počas návrhu systému, nie až po dokončení implementácie.

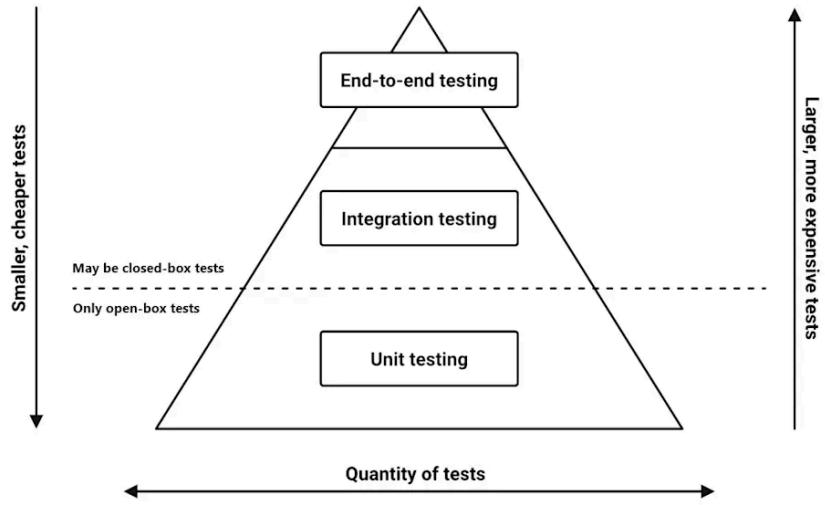


Zdroj: https://www.geeksforgeeks.org/software-engineering/software-engineering-sdlc-v-model/

Typy testovania

Existuje viacero úrovní testovania:

- **Unit testing**
- **Integration testing**
- **System testing**
- **End-to-end testing**
- **Acceptance testing**



Zdroj: https://circleci.com/blog/testing-pyramid/

Typy testovania - porovnanie

	Unit	Integračné	Systémové	E2E	Akceptačné
Rozsah	Jedna funkcia/trieda	Viacero komponentov	Celý systém	Celá používateľská cesta	Biznis požiadavky
Kto píše	Vývojár	Vývojár	QA/Vývojár	QA/Vývojár	QA/Biznis/Klient
Závislosti/dáta	Mockované	Reálne alebo takmer reálne	Reálne	Reálne	Reálne
Rýchlosť	Veľmi rýchle	Stredné	Pomalé	Veľmi pomalé	Veľmi pomalé
Kde beží	Dev + CI	CI	Staging	Staging/Pre-prod	Staging/Pre-prod
Príčina zlyhania	Logická chyba	Chyba prepojenia	Systémová chyba	Prerušený používateľský tok	Nespĺňa biznis potrebu
Príklad	`vypocitajDan()` vracia správnu hodnotu	API správne uloží do DB	Aplikácia funguje pod záťažou	Používateľ sa zaregistruje, prihlási, nakúpi	„Ako používateľ si môžem resetovať heslo"

Čo sú unit testy

Unit test je test, ktorý overuje **malú časť kódu (jednotku/unit)**.

Najčastejšie:

- metóda
- funkcia

Charakteristika:

- rýchle
- izolované

> 📌 **jeden test = jedna vec**

Prečo píšeme unit testy

Výhody:

- rýchla spätná väzba
- jednoduchšie refaktorovanie

- dokumentácia správania kódu
- menej chýb v produkcii

📌 **Dobrý unit test by mal byť:**

- jednoduchý
- čitateľný
- deterministický - rovnaký výsledok pre rovnaký vstup
- rýchly
- nezávislý od iných testov
- mať jasný výsledok - pass/fail

Kto píše unit testy

Najčastejšie:

👨‍💻 **Developeri**

Prečo?

- najlepšie poznajú kód
- testy sa píšu spolu s implementáciou

Často sa používajú prístupy:

- **TDD (Test Driven Development)**

Štruktúra unit testu (AAA pattern)

Väčšina testov používa štruktúru:

AAA

1. **Arrange** – príprava dát
2. **Act** – vykonanie operácie
3. **Assert** – overenie výsledku

Príklad:

```
// Unit test
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEqual
s;

public class CalculatorTest {

    @Test
    public void testAdd_returnsCorrectSum() {
        // Arrange
        Calculator calculator = new Calculator();
        int a = 5;
        int b = 3;

        // Act
        int result = calculator.add(a, b);

        // Assert
        assertEquals(8, result);
    }
}
```

Unit testy v Jave

Najčastejšie používaný framework:

JUnit

Umožňuje:

- písať testy
- spúšťať testy automaticky
- kontrolovať výsledky testov

JUnit je štandard v Java ekosystéme.

JUnit

- JUnit používa anotácie na definovanie testov a assertions na overenie správnosti výsledkov programu.

Základné anotácie

- `@Test` – označuje metódu ako test
- `@BeforeEach` – vykoná sa pred každým testom (napr. inicializácia objektov)
- `@AfterEach` – vykoná sa po každom teste (napr. cleanup)
- `@BeforeAll` – vykoná sa raz pred spustením všetkých testov v triede
- `@AfterAll` – vykoná sa raz po dokončení všetkých testov

Najčastejšie assertions

- `assertEquals(expected, actual)` – overí, že očakávaná a skutočná hodnota sú rovnaké
- `assertTrue(condition)` – overí, že podmienka je pravdivá
- `assertFalse(condition)` – overí, že podmienka je nepravdivá
- `assertNull(object)` – overí, že objekt je `null`
- `assertNotNull(object)` – overí, že objekt nie je `null`
- `assertThrows(Exception.class, () -> ...)` – overí, že kód vyhodí očakávanú výnimku

Príklad 1

Trieda:

```
public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

}
```

Príklad 1

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    void shouldAddTwoNumbers() {

        Calculator calculator = new Calculator();
```

```
        int result = calculator.add(2, 3);

        assertEquals(5, result);
    }
}
```

Príklad 2

Trieda:

```
public class DiscountCalculator {

    public double calculateDiscount(double price, boolean isMember) {

        if (price > 100) {
            if (isMember) {
                return price * 0.8; // 20% discount
            } else {
                return price * 0.9; // 10% discount
            }
        }

        return price; // no discount
    }
}
```

Logika:

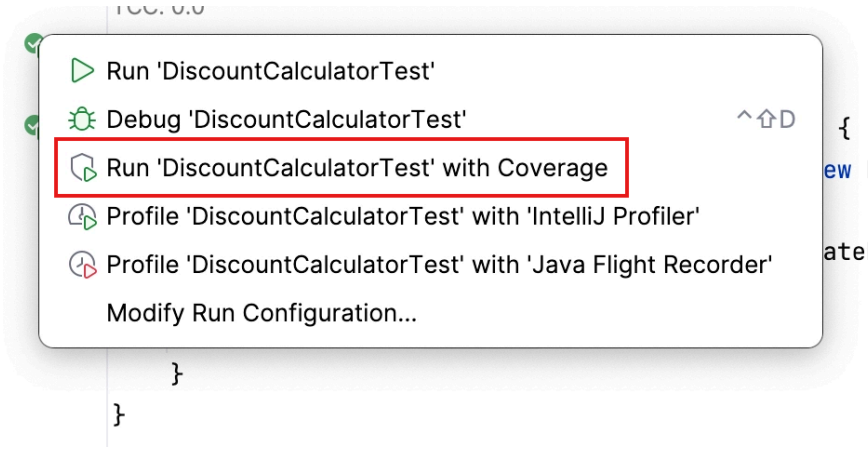
- cena > 100 a člen → **20% zľava**
- cena > 100 a nie je člen → **10% zľava**
- cena ≤ 100 → **bez zľavy**

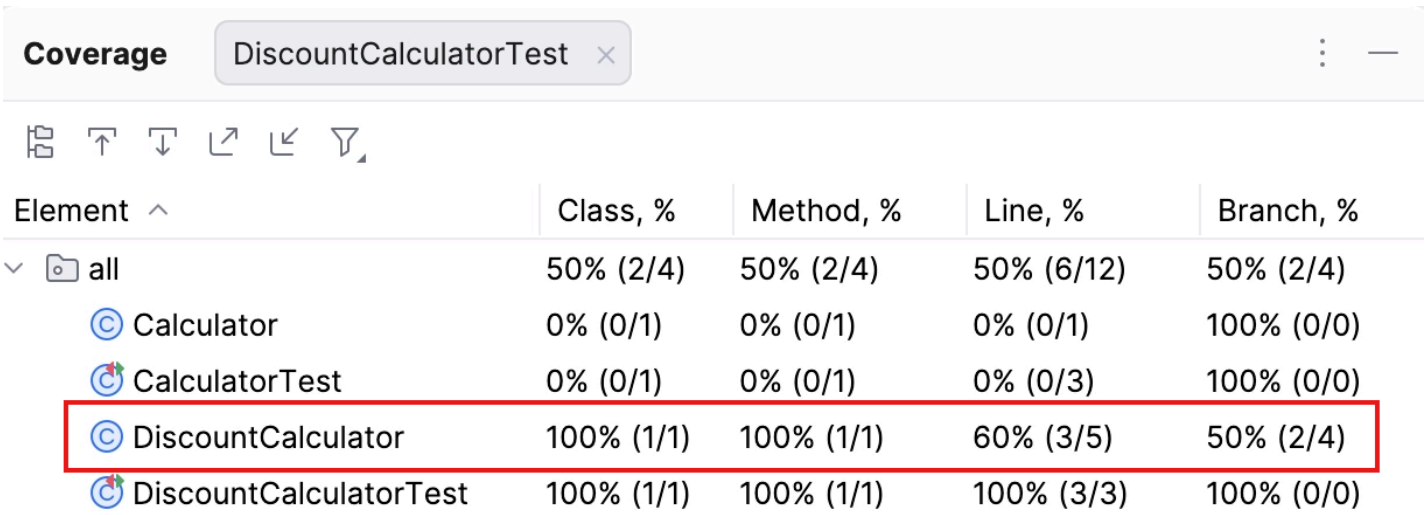
Príklad 2

```
@Test
void shouldApplyMemberDiscount() {

    DiscountCalculator calculator = new DiscountCalculator();

    double result = calculator.calculateDiscount(200, true);

    assertEquals(160, result);
}
```

- Tento test overuje len **jednu vetvu logiky**
- Coverage teda nebude 100 %. Prečo? ... Lebo ostatné vetvy **neboli** vykonané.

Príklad 2





Spúšťanie testov

Testy je možné spustiť:

- v **IDE (IntelliJ, Eclipse)**
- cez **Maven / Gradle**
- v **CI/CD pipeline**

Test coverage

Test coverage meria **aká časť kódu je pokrytá testami**.

- Čo to znamená?
- Ukazuje, koľko kódu bolo **spusteného/vykonaného** počas testov.

Nástroje:

- JaCoCo
- IntelliJ podpora pre code coverage - https://www.jetbrains.com/help/idea/code-coverage.html

⚠️ Vysoké coverage ≠ kvalitné testy.

Best practices

Dobrý unit test by mal byť:

- rýchly
- izolovaný
- čitateľný
- nezávislý od iných testov

Odporúčania:

- testuj jednu vec
- používaj jasné názvy testov
- nepíš príliš komplexné testy

Automatizácia testov

Testy sa často spúšťajú automaticky:

CI/CD pipeline

```
commit → build → tests → deploy
```

Výhody:

- chyby sa zachytia okamžite
- stabilnejší softvér

Zhrnutie

- Testovanie zlepšuje kvalitu softvéru
- Existuje viacero typov testovania
- Najväčšiu časť by mali tvoriť **unit testy**
- V Jave sa často používa **JUnit**
- Testy umožňujú bezpečnejší vývoj