

Razvoj vođen testovima (Test driven development)

Predavanje 5-1

Sadržaj

1. TDD-osnovni koncepti
2. Veza TDD-a i DevOps-a

TDD

- Razvoj vođen testovima (Test-driven development -TDD) je metodologija razvoja softvera (Beck, Astels 2003 godine) koja:
 - startuje sa dizajniranjem i razvojem testova za svaku malu funkcionalnost aplikacije (*test-first development*),
 - nakon čega slijedi pisanje produkcijskog koda dovoljnog da zadovolji taj test,
 - proces se završava *refaktoringom* napisanog produkcijskog koda.
- Orijentacija TDD pristupa je prvenstveno specifikacijska jer se prvo razvija test na osnovu specifikacije.
- Metodologija forsira developera da razmotri korištenje i namjenu metode prije njene implementacije.
- Za TDD su važni agilni zahtjevi i agilne tehnike dizajna i pisanje čistog koda.

TDD iteracija sastoji se od 3 koraka koja se nazivaju Red-Green-Refactor.

Red:

1. korak je pisanje testa.

- Developer piše test za novu (ili dio) funkcionalnosti.
 - Pokretanjem testova očekuje se da novo napisani test padne.
 - Ako novo napisani test ne padne to znači da je kod za tu funkcionalnost već napisan.
 - Ako drugi testovi padnu to znači da postoji problem sa test *suitom* - vjerovatno deterministički pad nekog testa što je potrebno fiksirati prije sljedeće faze.
-
- Ova faza se naziva Red zato što mnoga unit okruženja prikazuju crveno testove koji padaju.

2. korak: Developer piše kod da novo napisani test u Red fazi prođe.

Green:

- Novo napisani kod ne smije uzrokovati da neki drugi test padne.
- Ako se to desi potrebno je fiksirati problem.
- Na ovoj tački se očekuje 'ružan' kod jer je fokus da kod korektno radi.
- Na kraju ove faze svi testovi prolaze (green).

3. korak: Refaktoring koda - promjena strukture koda pri čemu ponašanje ostaje isto.

Refactor:

- Kada testovi prođu potrebno je za novo napisani kod izvršiti neophodni i odgovarajući refaktoring – (promjena strukture koda pri čemu ponašanje ostaje isto).
- Fraza „*first make it green then make it clean*“ opisuje svrhu i povezanost Green i Refactor faze.
- Nakon završetka implementacije za jednu (ili dio) funkcionalnosti nastavlja se daljnja implementacija kroz 3 faze: **Red-Green-Refactor**.

Sumirano TDD proces se obavlja:

1. Napiši test za funkcionalnost koja još nije napisana.
2. Izvrši testni *suite* – samo novo napisani test treba da padne. Ako nije ustanovi zašto testovi padaju i fiksiraj problem.
3. Napiši dovoljno koda da test prođe, bez uzrokovanja da drugi testovi padnu.
4. Izvrši testni *suite* – ako neki test padne vrati se na korak 3. Ako svi testovi prođu nastavi.
5. Urađi refaktoring koda koji je napisan i eventualno povezanog koda.
6. Izvrši testni *suite*, ako neki od testova padne idi ponovo na korak 5. Ako svi testovi prođu nastavi.
7. Ako postoji još neka funkcionalnost za implementaciju idi na korak 1. Ako nema više takvih funkcionalnosti razvoj aplikacije je završen.

- Na kraju kompletne implementacije se dobija solidan skup testova - *testni suite* koji je direktno relevantan za funkcionalnost programa.

Principi TDD razvoja

- Ima više principa koje treba imati na umu kada se razvija softver primjenom TDD načina. Neki od njih su:

-**YAGNI** (*You Ain't Gonna Need It*): Ne pisati kod koji nije potreban u namjeri da testovi prođu.

-**KISS** (*Keep It Simple, Stupid*): Jedan od ciljeva TDD-a je da se osigura da je kod fleksibilan, proširiv i da nije kompleksan. Kompleksni sistemi su teški za razumijevanje i teško je dodavati nove karakteristike i funkcionalnosti.

-***Fake It'Til You Make It***: Uredu je koristiti *fake* metode.

-**Izbjegavati testove koji se izvršavaju sporo (*Avoid Slow Running Tests*):**

Svaka iteracija (red-green-refactor) izvršava 3 puta kompletan test *suite*. To je minimum uz pretpostavku da kod nikada ne pravi probleme u drugim testovima ili da sam ne sadrži neku grešku.

Ako izvršavanje traje 2 -3 sec. to je minorna cijena koja se plaća za kvalitet koda koji daje TDD.

Međutim, ako izvršavanje traje dugo to može negativno uticati na dinamiku razvoja i fokus developera.

Primjer primjene TDD pristupa

- Potrebni je razviti FizzBuzz aplikaciju koja ispisuje:
 - (1) brojeve od 1 do 100 osim,
 - (2) ako je broj djeljiv sa 3 tada ispisuje Fizz umjesto broja,
 - (3) ako je broj djeljiv sa 5 ispisuje riječ Buzz umjesto broja,
 - (4) ako je broj djeljiv i sa 3 i 5 tada ispisuje FizzBuzz umjesto broja.

Prvo je potrebno razviti „radni skeleton“ aplikacije. Potrebno je postaviti **unit testno okruženje** i kreirati **FizzBuzz** klasu.

- Za početak se kreira *pure* funkcija – prima parametar i vraća prazan string.

```
namespace FizzBuzzProdukcija
{
    public class FizzBuzz
    {
        public static string fizzbuzzify(int num)
        {
            return "";
        }
    }
}
```

Za unit testiranje koristi se MsTest.

(1) brojeve od 1 do 100

- Primjenjujući TDD (korak 1) kreira se **prvi testni slučaj**.
- Da bi razvili kod koji odgovara zahtjevu (1) prvo ćemo napisati test kojim se testira da li `fizzbuzzify` metoda kada primi broj 1 vrati 1.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using FizzBuzzProdukcija;

namespace FizzBuzz1
{
    [TestClass]
    public class FizzBuzzTest
    {
        [TestMethod]
        public void test1Return1()
        {
            string returnedVal = FizzBuzz.fizzbuzzify(1);
            Assert.AreEqual("1", returnedVal);
        }
    }
}
```

Test `test1Return1()` kojim se testira da li produkcijski kod vraća 1 ako primi 1

test1Return1

Source: `UnitTest1.cs` line 12

✖ Test Failed - test1Return1

Message: `Assert.AreEqual` failed. Expected:<1>. Actual:<>.

Elapsed time: 19 ms

▲ StackTrace:

`FizzBuzzTest.test1Return1()`

Potrebno je napisati kod koji zadovoljava test.

```
public class FizzBuzz
{
    public static string fizzbuzzify(int num)
    {
        return "1";
    }
}
```

Kod `fizzbuzzify` metode koji zadovoljava `test1Return1()`

-Kada se ponovo pokrene već napisani test test će proći.

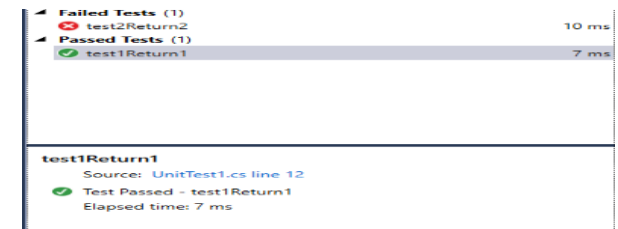
-Sljedeći korak je razmatranje refaktoringa. U ovom slučaju još uvijek nema razloga za provođenje refaktoringa u cilju poboljšanja koda.

Razmatrajući i dalje zahtjev iskazan pod **(1)** dodaje se drugi test kojim se **testira da li produkcijski kod odnosno fizzbuzzify metoda ako primi 2 vrati 2** što je u skladu sa već pomenutim zahtjevom.

Kada se izvrše testovi zadnji dodani test pada jer metoda `fizzbuzzify(int num)` uvijek vraća 1 nikada ne vraća 2.

```
[TestClass]
public class FizzBuzzTest
{
    [TestMethod]
    public void test1Return1()
    {
        string returnedVal = FizzBuzz.fizzbuzzify(1);
        Assert.AreEqual("1", returnedVal);
    }
    [TestMethod]
    public void test2Return2()
    {
        string returnedVal = FizzBuzz.fizzbuzzify(2);
        Assert.Equals("2", returnedVal);
    }
}
```

Uz postojeći test1Return1 dodan je test koji testira da li za ulaz 2 fizzbuzzify metoda vrati 2



Potrebno je izvršiti promjenu metode `fizzbuzzify(int num)` da test prođe.

```
public class FizzBuzz
{
    public static string fizzbuzzify(int num)
    {
        if (num == 1)
            return "1";
        else
            return "2";
    }
}
```

Kod koji zadovoljava testni suit sa dva testa - test1return1 i test2Return2

- Sada oba napisana testa prolaze.

-
- Potrebno je **uraditi refaktoring koda** tako da radi sa svim integer vrijednostima umjesto da se dodaje novi `else` za svaku individualnu vrijednost.

```
public class FizzBuzz
{
    public static string fizzbuzzify(int num)
    {
        return (num.ToString());
    }
}
```

Metoda `fizzbuzzify` nakon refaktoringa radi sa svim ne-buzzi i ne-fuzzi vrijednostima

- Testovi prolaze tako da se sada može **razmatrati zahtjev (2)**.
- Naravno, već se uviđa da kod neće dobro raditi za sve *ne-bizzi*, *ne-fuzzi* vrijednosti.

(2) ako je broj djeljiv sa 3
tada ispisuje `Fizz` umjesto
broja

`fizzbuzzify(3)` će vratiti 3 i to će
uzrokovati da test padne.
Potrebno je izvršiti promjenu metode da
zadovolji zadnji dodani test.

- Potrebno je dodati test koji će se koristiti za implementaciju ovog zahtjeva, tj. **test treba da testira da li `fizzbuzzify` metoda ako primi 3 vrati "Fizz".**
- Dodatni test za ovu namjenu (nije prikazan cijeli testni *suite*).

```
[TestMethod]
public void test3ReturnsFizz()
{
    string returnedVal = FizzBuzz.fizzbuzzify(3);
    Assert.AreEqual("Fizz", returnedVal);
}
```

Novi test `test3ReturnsFizz` testira da li metoda `fizzbuzzify` za vrijednost 3 vrati `Fizz`

```
public static string fizzbuzzify(int num)
{
    if (num == 3)
        return "Fizz";
    else return (num.ToString());
}
```

Implementacija metode `fizzbuzzify` koja zadovoljava `test3ReturnsFizz`

Ovo nije idealno rješenje jer će raditi samo za 3. Sa refaktoringom može se postići da kod radi dobro za bilo koji broj koji je djeljiv sa 3.

```
public static string fizzbuzzify(int num)
{
    if (num % 3 == 0)
        return "Fizz";
    else return (num.ToString());
}
```

Implementacija metode `fizzbuzzify` koja zadovoljava zahtjeve (1) i (2)

- Sada test prolazi. Mogu se napisati i dodatni unit testovi za 6,9,.. i provjeriti da li gornji test zadovoljava namjenu.

(3) ako je broj djeljiv sa 5 ispisuje riječ `Buzz` umjesto broja

- Prikaz samo dodatnog testa za ovu namjenu (nije prikazan cijeli testni *suite*).

```
[TestMethod]
public void test5ReturnsBuzz()
{
    string returnedVal = FizzBuzz.fizzbuzzify(5);
    Assert.AreEqual("Buzz", returnedVal);
}
```

Novi test `test5ReturnBuzz` testira da li metoda `fizzbuzzify` za vrijednost 5 vrati `Buzz`

Test pada jer metoda vraća 5.

Potrebno je izvršiti promjenu `fizzbuzzify` metode tako
što ćemo dodati dodatni `else`.

```
public class FizzBuzz
{
    public static string fizzbuzzify(int num)
    {
        if (num % 3 == 0)
            return "Fizz";
        else if (num % 5 == 0)
            return "Buzz";
        else return (num.ToString());
    }
}
```

Implementacija metode `fizzbuzzify` koja zadovoljava zahtjeve (1), (2) i (3)

Test sada prolazi i za sada
nije neophodan neki
dodatni refaktoring.

(4) ako je broj djeljiv i sa 3 i 5 tada ispisuje FizzBuzz umjesto broja

- Sada treba dodati test za zahtjev (4) koji nalaže da metoda vrati „FizzBuzz“ ako je na ulazu vrijednost djeljiva i sa 3 i sa 5.
- Odabrati ćemo vrijednost 15 i napisati test `test15ReturnsFizzBuzz()`.

```
[TestMethod]
public void test15ReturnsFizzBuzz()
{
    string returnedVal = FizzBuzz.fizzbuzzify(15);
    Assert.AreEqual("FizzBuzz", returnedVal);
}
```

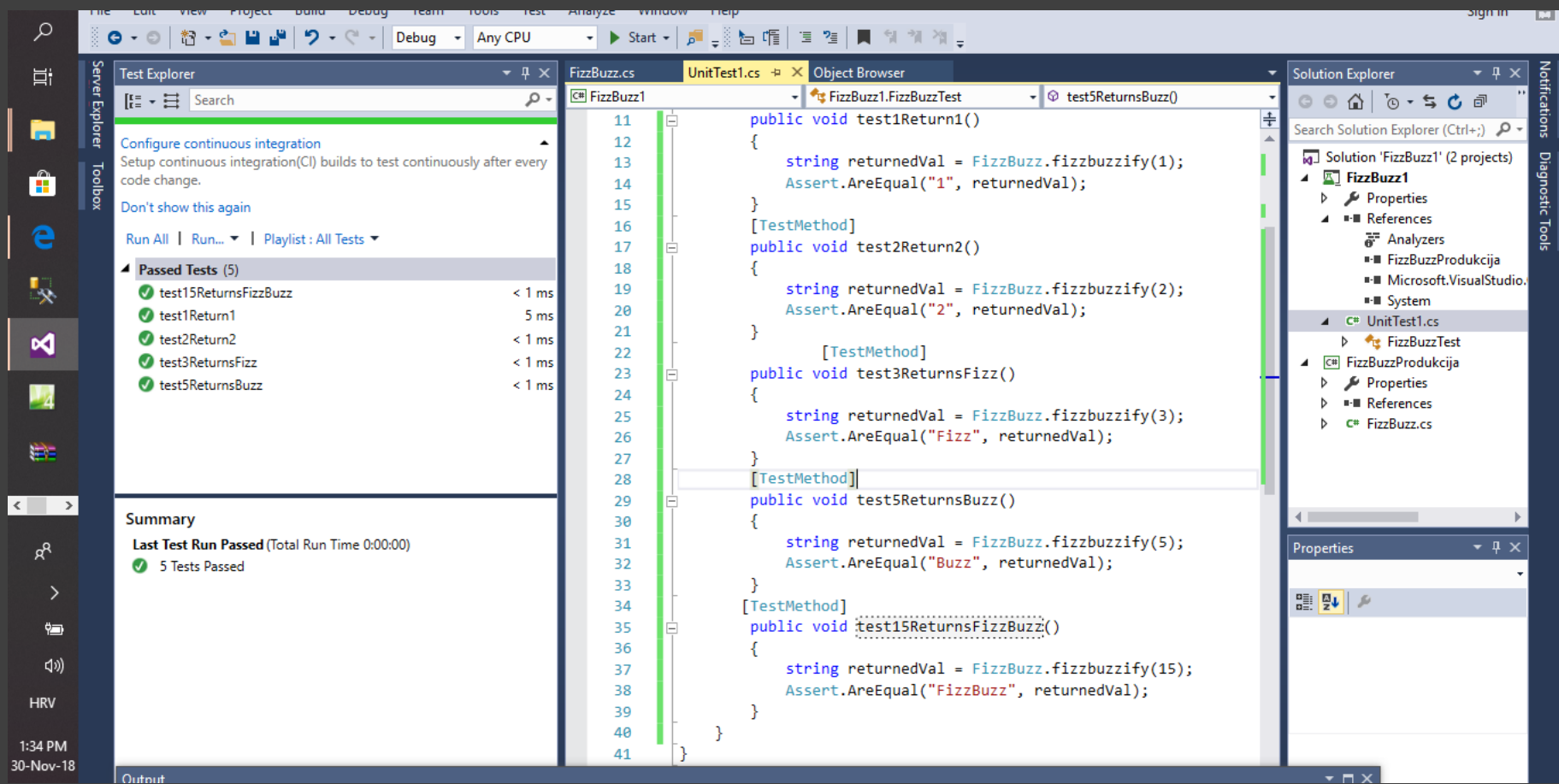
Novi test `test15ReturnFizzBuzz` testira da li metoda `fizzbuzzify` za vrijednost 15 vrati FizzBuzz

Novo napisani test pada jer `fizzbuzzify` metoda vraća „Fizz“.

Potrebno je implementirati kod u metodi da prođe i novo napisani test.

```
public class FizzBuzz
{
    public static string fizzbuzzify(int num)
    {
        if ((num % 3 == 0) && (num % 5 == 0))
            return "FizzBuzz";
        else if (num % 3 == 0)
            return "Fizz";
        else if (num % 5 == 0)
            return "Buzz";
        else return (num.ToString());
    }
}
```

Implementacija metode `fizzbuzzify` koja zadovoljava zahtjeve (1), (2), (3) i (4)



- Svi testovi sada prolaze.
- Može se dalje raditi refaktoring.
- Prilikom stvarnog razvoja ponekad su koraci veći ali treba voditi računa da se napišu i grupišu svi specifični testovi za ulazne i izlazne vrijednosti.
- Također, treba voditi računa i o imenovanju testova.

Primjer 2: Želimo napisati klasu `DiscountCalculator` koja računa popust:
Ako je iznos veći od 100, popust je 10%.
Inače nema popusta.

1. RED – napiši test koji ne prolazi

Prvo pišemo **unit test**, iako klasa još ne postoji.
Koristićemo npr. **xUnit**.

```
using Xunit;

public class DiscountCalculatorTests
{
    [Fact]
    public void
    GetDiscountedPrice_PriceOver100_ShouldApply10PercentDiscount()
    {
        // Arrange
        var calculator = new DiscountCalculator();

        // Act
        var result =
        calculator.GetDiscountedPrice(200);

        // Assert
        Assert.Equal(180, result); // 10% popusta =
        200 - 20 = 180
    }
}
```

Očekivano: Test ne prolazi (klasa `DiscountCalculator` ne postoji).

2. GREEN – napiši minimalan kod da test prođe

```
public class DiscountCalculator
{
    public double GetDiscountedPrice(double price)
    {
        if (price < 0)
            throw new ArgumentException("Price
            cannot be negative.");

        return price > 100 ? price * 0.9 : price;
    }
}
```

xUnit primjer.

3. REFACTOR

– možemo npr. dodati provjeru da cijena ne bude negativna

Dodatni testovi:

```
public class DiscountCalculator
{
    public double GetDiscountedPrice(double price)
    {
        // Provjera negativne cijene
        if (price < 0)
            throw new ArgumentException("Price cannot be
negative.");

        // Primjena popusta
        if (price > 100)
            return price * 0.9; // 10% popust

        return price;
    }
}
```

```
[Fact]
public void
GetDiscountedPrice_PriceEqualTo100_ShouldNotA
pplyDiscount()
{
    var calculator = new
DiscountCalculator();
    var result =
calculator.GetDiscountedPrice(100);
    Assert.Equal(100, result);
}
```

```
[Fact]
public void
GetDiscountedPrice_NegativePrice_ShouldThrowE
xception()
{
    var calculator = new
DiscountCalculator();
    Assert.Throws<ArgumentException>(() =>
calculator.GetDiscountedPrice(-50));
}
```

Ponovo se pokrenu testovi. Svi prolaze .

RED GREEN REFACTOR - sumarno

Faza	Aktivnost	Naučeni koncept
RED	Pišemo test prije koda.	Razmišljanje o očekivanom ponašanju.
GREEN	Implementiramo minimalno, da test prođe.	Fokus na funkcionalnost.
REFACTOR	Promjena strukture koda bez promjene ponašanja.	Čist kod i pokrivenost testovima.

Prednosti TDD metode

- TDD doprinosi bolje dizajniranom, čistijem i proširivom kodu.
- Pomaže da se razumije kako se kod koristi i koja je njegova interakcija sa drugim modulima.
- TDD forsira manje programske cjeline i poštovanje principa pojedinačne odgovornosti kao i pisanje samo produkcijskog koda koji prolazi testove bazirane na korisničkim zahtjevima.
- Brzi (mali) krugovi razvoja održavaju developera fokusiranog na specifične ciljeve. To je mnogo produktivnije nego pisanje koda u velikim koracima.
Mnogo je lakše pronaći i fiksirati eventualne defekte u tom slučaju nego da je odjednom napisano stotine linija koda.
- Dobra osobina je i što se refaktoring provodi rano i često i postaje dio razvoja i dobra navika.
- Sa TDD-om gradi se automatizirani skup testova koji se može ponovo pokretati.
- TDD prirodno vodi dizajnu klasa koji poštuju SRP princip jer kada se pišu testovi, lakše se uoče granice odgovornosti.
- TDD mijenja način razmišljanja: *prvo razmišljamo o očekivanom ponašanju*, a ne o implementaciji.
- To vodi do „design for testability“ — koda koji je modularniji i bolje strukturiran.

TDD nije samo tehnika testiranja — to je pristup izgradnji kvalitetnog softvera.

Tipični **uzroci koji vode do siromašnih rezultata TDD-a**

- pisanje više testova odjednom,
- pisanje prevelikih testova,
- zaboravlja se često pokretanje testova,
- samo nekoliko članova tima primjenjuje ovu metodu,
- siromašno održavanje testnog *suita*.

- Tipični **uzroci koji vode do siromašnih rezultata TDD-a**

- TDD usporava razvoj - U početku da, ali kasnije ubrzava otklanjanje grešaka.
- Sve mora biti testirano - Fokus je na kritičnom i logičkom dijelu koda.
- TDD doprinosi kvalitetu - Samo uz kombinaciju s dobrim dizajnom i disciplinom.

.

Uvod u DevOps

- DevOps predstavlja skup praksi koje povezuju razvojne (**Development**) i operativne (**Operations**) timove s ciljem brže, pouzdanije i kvalitetnije isporuke softvera. Osnovu ovog pristupa čine automatizacija procesa, kontinuirano testiranje i stalna saradnja između članova tima, čime se smanjuje mogućnost grešaka, ubrzava isporuka i povećava stabilnost softverskog sistema.
- **Ključni elementi DevOps-a:**
 - Kontinuirana integracija (CI)** – Svaki put kada developer napravi promjenu u kodu, ta promjena se automatski testira i integriše u zajednički repozitorij. Na ovaj način se brzo otkrivaju greške i osigurava stabilnost sistema.
 - Kontinuirana isporuka (CD – Continuous Delivery)** – Proces automatizirane distribucije, testiranja i isporuke softvera. Kada kod uspješno prođe sve testove, on se automatski priprema ili isporučuje u produkcijsko okruženje.
 - Kultura saradnje** – Developeri, tester i sistem administratori blisko sarađuju i dijele odgovornost za kvalitet, stabilnost i pouzdanost proizvoda. Razvojni i operativni timovi djeluju kao jedinstvena cjelina, što doprinosi efikasnijem razvoju i bržoj isporuci softvera.

Unit testovi / TDD i DevOps – kako rade zajedno

- **Fokus testiranja:** osigurati da je kod ispravan i kvalitetan od samog početka (*unit testovi ili TDD*).
- **Fokus DevOps-a:** koristi testove u CI/CD pipeline-u da spriječi da neispravan kod dospije u produkciju.
- **Zajednička filozofija:** kvalitet, automatizacija i kontinuirana isporuka.
- **Automatizacija procesa:** svaki *commit* pokreće testiranje i *deploy*.
- Testovi grade kvalitet, a DevOps ih koristi.
- Rezultat: kvalitet koda od prve linije do produkcije.

Kako se inspekcije koda povezuju s DevOps-om

- U modernom DevOps okruženju, inspekcije koda (code review) i automatizirane statičke analize postaju sastavni dijelovi CI/CD procesa.
- Inspekcije koda su ugrađene u *pipeline*, tako da svaka promjena prolazi kroz *code review* i/ili automatsku statičku analizu.
- Ključne prednosti:
 - Kvalitet se provjerava kontinuirano, a ne samo prije isporuke.
 - Smanjuje se broj grešaka i povećava pouzdanost isporuka.
 - DevOps automatizira i osnažuje inspekcije – više nisu ručni proces „na kraju“, nego ugrađen mehanizam kvaliteta u svakoj iteraciji razvoja.
 - U DevOps kulturi, inspekcija koda nije faza – to je dio svakog *commita*.

TDD, Unit Testovi, Inspekcija i DevOps

1. TDD / Unit Testovi

Kod se piše uz testove (TDD) ili se naknadno dodaju unit testovi.

Unit testovi otkrivaju greške odmah, prije nego što kod dođe u repozitorij.

2. *Code Review* / Inspekcija

Obično se radi nakon lokalnog testiranja, prije spajanja u glavni repozitorij.

Provjerava stil, arhitekturu, sigurnost i kvalitet koda koji je već testiran.

3. DevOps / CI/CD

Pipeline automatski pokreće sve testove (unit i integracijske) nakon *commita* ili PR (Pull Request) – zahtjeva za spajanje promjena u glavnu granu.

Na taj način DevOps osigurava da samo kod koji je testiran i pregledan dospije u produkciju.