

A **TDD (Test-Driven Development)** és a **BDD (Behavior-Driven Development)** két népszerű fejlesztési módszer, amelyek tesztelési folyamatokra építenek, de eltérő szemlélettel és célkitűzésekkel.

Fő különbségek:

1. Fókusz

- **TDD (Test-Driven Development):**

- A TDD középpontjában a **kód implementációja** áll. A cél az, hogy a fejlesztő **előre megírja a tesztek** a kód implementálása előtt, majd a tesztek alapján hozza létre a funkciókat. A TDD alacsonyabb szintű tesztekre (pl. unit tesztekre) összpontosít, amelyek konkrét kódrészleteket vizsgálnak.
- A TDD lépései:
 - 1. Írj egy tesztet (amely először el fog bukni).
 - 2. Írd meg a kódot, hogy a teszt sikeres legyen.
 - 3. Refaktoráld a kódot, ha szükséges.

- **BDD (Behavior-Driven Development):**

- A BDD a **viselkedésre** összpontosít, azaz arra, hogy a rendszernek hogyan kell viselkednie a felhasználó szempontjából. A tesztek a rendszer viselkedését írják le, nem pedig a kód részleteit. A BDD tesztek természetes nyelv közeli, mindenki által érthető formában írják meg, gyakran felhasználva a Gherkin szintaxist (`Given`, `When`, `Then` struktúrában).
- A BDD célja az üzleti elemzők, fejlesztők és tesztelők közötti **együtműködés elősegítése**, hogy minden érintett jobban megértse a rendszer elvárt viselkedését.

2. Szint

- **TDD:**

- Főként alacsony szintű (unit tesztek) tesztelésre összpontosít. A tesztek a kódrészletek helyes működését ellenőrzik.

- **BDD:**

- Magasabb szintű tesztelés, amely a rendszer viselkedését vizsgálja, például hogyan reagál bizonyos felhasználói interakciókra vagy üzleti folyamatokra.

3. Szemléletmód

- **TDD:**

- A kódtervezés **teszt-alapú**. A fejlesztő először tesztet ír, majd ehhez igazítja a kódot. A TDD során a fejlesztők inkább a funkciók implementálására és a kód helyességére koncentrálnak.

- **BDD:**

- A tervezés **viselkedés-alapú**. A tesztek a rendszer által elvárt viselkedést írják le, tehát a felhasználói élményt és üzleti igényeket helyezik előtérbe.

4. Nyelvezet

- **TDD:**
 - Teszteket gyakran programozási nyelveken íróják meg, amelyeket elsősorban a fejlesztők értékelnek. Például egy TDD teszt Pythonban, JUnitban stb. íródik.
- **BDD:**
 - A teszteket emberi nyelven közeli módon fogalmazzák meg, így nemcsak fejlesztők, hanem üzleti elemzők és más érintettek is megértik. A Gherkin szintaxis egy példa erre:

```
Given the user is on the login page
When they enter valid credentials
Then they should be logged in successfully
```

Példa

A nulláról indulunk, lépésenként bemutatjuk a módszert.

1. Projekt inicializálása

Először hozzunk létre egy új projekt könyvtárat, és inicializáld a Node.js projektet.

```
mkdir tdd-project
cd tdd-project
npm init -y
```

Ez létrehoz egy alap **package.json** fájlt.

2. Függőségek telepítése

Telepítsük a szükséges fejlesztői függőségeket: **Mocha** a teszteléshez, **Chai** az aszertálásokhoz, és **Sinon** a mockoláshoz és stuboláshoz. Mivel a projektben jelszó hash-elésre is szükség lesz, telepítük a **bcrypt** könyvtárat is.

```
npm install mocha chai sinon bcrypt --save-dev
```

3. Mappastruktúra létrehozása

Hozzuk létre a szükséges mappákat és fájlokat a projekt szerkezetéhez.

```
mkdir test
mkdir services
```

```
mkdir repositories
ni ./test/userService.test.js
ni ./services/userService.js
ni ./repositories/userRepository.js
```

megjegyzés: az `ni` parancs powershell-ben a linuxos `touch` parancs megfelelője.

Most a projekt struktúrája így néz majd ki:

```
tdd-project/
├── test/
│   └── userService.test.js    // Tesztek a UserService-hez
├── services/
│   └── userService.js        // UserService osztály
├── repositories/
│   └── userRepository.js     // UserRepository osztály
└── package.json              // Node.js projekt leíró fájl
```

Létrejött három üres állomány.

4. Mocha konfigurálása

A **Mocha** futtatásához a `package.json` fájlban hozzá kell adni egy részt, amely a `mocha` parancsot futtatja a `test` mappában:

Nyissuk meg a `package.json` fájlt, és adjuk hozzá a `scripts` részhez a következőt:

```
"scripts": {
  "test": "mocha"
}
```

5. Tesztek írása (TDD módszerrel)

Most kezdhethetjük a TDD folyamatot: először a tesztekét írjuk meg. Például a `userRepository.test.js` fájlba írjuk a következő tesztek:

```
const assert = require('assert');
const UserRepository = require('../repositories/userRepository');

describe('UserRepository', function() {
  let userRepository;
```

```
beforeEach(function() {
  userRepository = new UserRepository();
});

it('should return null if user is not found by email', async function() {
  const user = await
userRepository.findUserByEmail('notfound@example.com');
  assert.strictEqual(user, null);
});

it('should save a user and retrieve it by email', async function() {
  const newUser = { email: 'test@example.com', password: 'hashedPassword'
};
  await userRepository.saveUser(newUser);
  const foundUser = await
userRepository.findUserByEmail('test@example.com');
  assert.strictEqual(foundUser.email, 'test@example.com');
  assert.strictEqual(foundUser.password, 'hashedPassword');
});

it('should handle saving multiple users', async function() {
  const user1 = { email: 'user1@example.com', password: 'password1' };
  const user2 = { email: 'user2@example.com', password: 'password2' };

  await userRepository.saveUser(user1);
  await userRepository.saveUser(user2);

  const foundUser1 = await
userRepository.findUserByEmail('user1@example.com');
  const foundUser2 = await
userRepository.findUserByEmail('user2@example.com');

  assert.strictEqual(foundUser1.email, 'user1@example.com');
  assert.strictEqual(foundUser2.email, 'user2@example.com');
});
});
```

6. Tesztek futtatása

Futtasd a Mocha teszteket, hogy megbizonyosodj arról, hogy a tesztek elbuknak (mivel még nem írtad meg a tényleges implementációt).

```
npm test
```

Ez a parancs futtatja a mocha parancsot, amely végigmegy a test mappában lévő teszteken. Mivel a UserRepository még nem implementált, a tesztek elbuknak, ami a TDD módszer lényege: először a tesztek buknak el, majd az implementáció következik.

7. Implementáció megírása

Most írd meg a tényleges kódot a tesztek sikeressé tételéhez.

`userRepository.js`:

```
>
// repositories/userRepository.js

class UserRepository {
  constructor() {
    this.users = []; // Szimulált adatbázis tömbként
  }

  // Felhasználó keresése e-mail alapján
  async findUserByEmail(email) {
    const user = this.users.find(user => user.email === email);
    return user || null;
  }

  // Új felhasználó mentése az adatbázisba
  async saveUser(user) {
    this.users.push(user);
    return user;
  }
}

module.exports = UserRepository;
```

8. Tesztek újrafuttatása

Most futtasd újra a teszteket:

```
npm test
```

Most a teszteknek sikeresen át kell menniük, mivel az implementáció megfelel a tesztek elvárásainak.

9. További tesztek és fejlesztés

A TDD ciklusban a következő lépés az újabb tesztek írása új funkciókra, majd a funkciók implementálása annak érdekében, hogy a tesztek mindig megfeleljenek. A TDD ciklus három lépése a következő:

1. **Teszt írás** (a teszt el fog bukni).
2. **Implementáció írás** (hogy a teszt átmenjen).
3. **Refaktorálás** (a kód minőségének javítása a teszt sikerességének megtartásával).

Ezzel a módszerrel mindig biztosak lehetünk abban, hogy a kód megfelel a tesztelt

követelményeknek.

From:

<https://edu.iit.uni-miskolc.hu/> - **Institute of Information Science - University of Miskolc**

Permanent link:

https://edu.iit.uni-miskolc.hu/tanszek:oktatas:tdd_es_bdd?rev=1728732658

Last update: **2024/10/12 11:30**

