

PROGRAMOZÁSI TECHNOLÓGIÁK

8. óra – CLEAN CODE ÉS TDD

2024.04.19

Szalai Patrik

 szalai.patrik@uni-milton.hu

TÉMAKÖRÖK:

Tiszta kód

- Bevezetés
- Cserkésszabály
- Clean Code röviden
 - Általános szabályok
 - Tervezési szabályok
 - Olvashatósági tippek
 - Névkonvenciók
 - Funkció szabályok
 - Komment szabályok
 - Forráskód struktúra
 - Objektumok és Adatstruktúrák
 - Tesztek
 - Kódszagok

Tesztelés

- Ismétlés
- Unit Test
- TDD és BDD
- AAA és GWT
- Mock és SUT

TDD

- Lépései
- Clean Code és TDD
- Előnyök
- Piros-Zöld-Piros
- Piros-Zöld-Kék-Piros
- Kettős könyvelés a szoftverfejlesztésben

TISZTA KÓD

MIRŐL SZÓL?

CSERKÉSSZABÁLY

CLEAN CODE RÖVIDEN

TISZTA KÓD

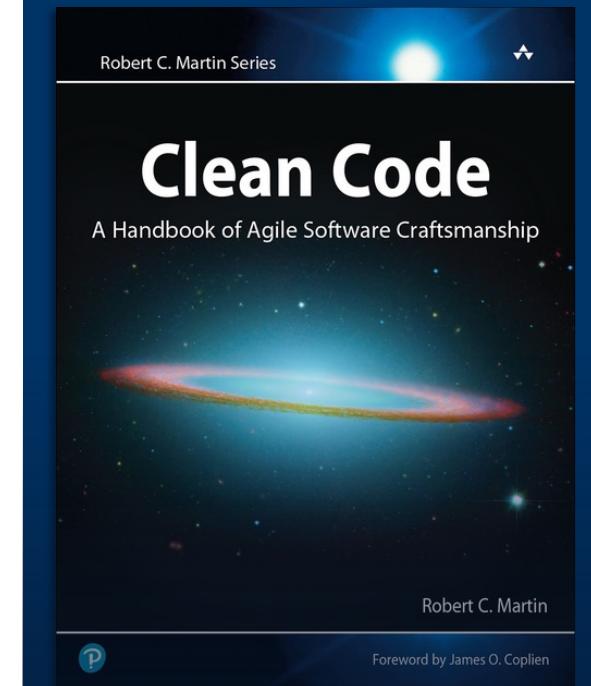
Robert C. Martin – Uncle Bob

- A kód akkor tiszta, ha könnyen érhető – a csapat minden tagja által.
- A tiszta kód olvasható és tovább építhető nem csak az eredeti szerző-, hanem bármilyen más fejlesztő számára.
- Az érthetőségből ered az olvashatóság, újrafelhasználhatóság, bővíthetőség, kiegészíthetőség és karbantarthatóság.

Rossz kód és tiszta kód

- A rossz kód is tud üzemelni
- De ha a kódunk nem tiszta, akkor könnyen letérdelteti a fejlesztést
- A szoftverkriticishez vezető egyenes út a rosszul megírt, nem tiszta kód

Ismétlő kérdés: Milyen károkkal járhat a szoftverkriticis kialakulása?
Soroljunk fel hármat!



CSERKÉSSZABÁLY

... és további általános szabályok

I Kövessük a standard konvenciókat

- I Alkalmazzuk az elismert programozási gyakorlatokat
- I Használjuk a programnyelv és keretrendszer technológiáit
- I Igazodjunk a munkaközösséggünk szokásaihoz

I Ne bonyolítsunk túl. Az egyszerű minden nagyszerű.

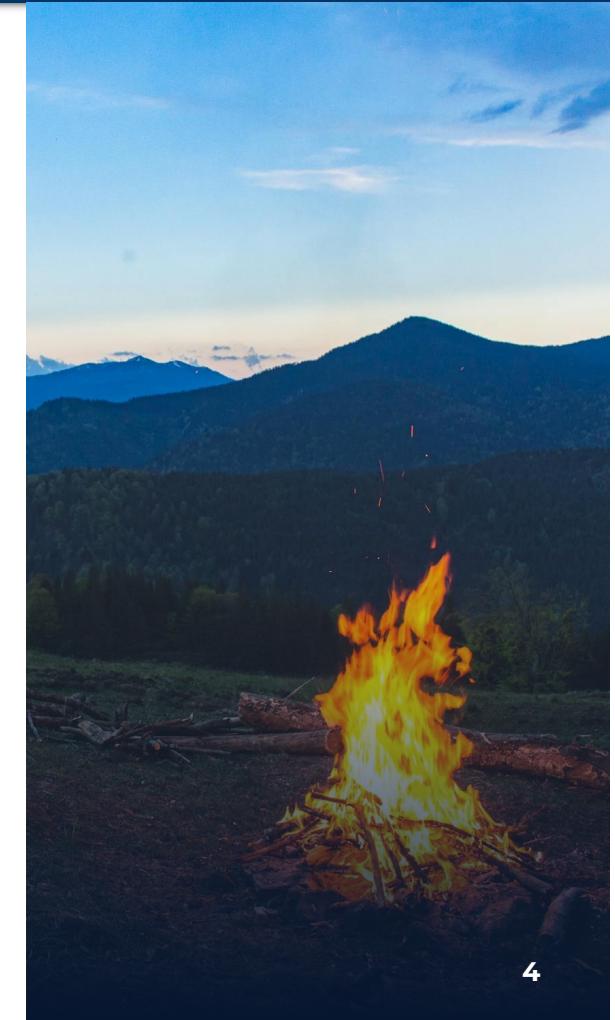
- I KISS – *Keep it simple, stupid (alapelv)*
- I Csökkentsük a komplexitást, ahol csak lehet
- I Tervezés és Implementáció során egyaránt
- I Az egyszerűség könnyen értelmezhetővé teszi kódunkat

I Cserkésszabály

- I Mindig hagyd a táborhelyet tisztábban, mint ahogy találtad
- I Mindig hagyd a kódot szebb állapotban, mint ahogy találtad.
- I Refaktorálás, dokumentáció, bugfixek, de minimum egy ticket

I Mindig találd meg a hiba gyökerét.

- I Ne csak hegesszünk/workaroundoljunk, tárjuk fel a hiba eredetét
- I Vezessük vissza a problémát annak gyökeréig



TERVEZÉSI SZABÁLYOK

A konfigurálható adatokat tartsuk meg magas szinten

- | Konstansok, paraméterek, melyek módosítják a kód viselkedését
- | Magas absztrakciós szinten helyezkedjenek el
- | Ideális esetben szeparáltan a főlogikától
- | Ezáltal könnyebb kezelni- és módosítani ezeket az adatokat anélkül, hogy a megvalósítás mélyére kellene nyúlnunk értük

Alkalmazzunk többalakúságot if/else és switch/case helyett

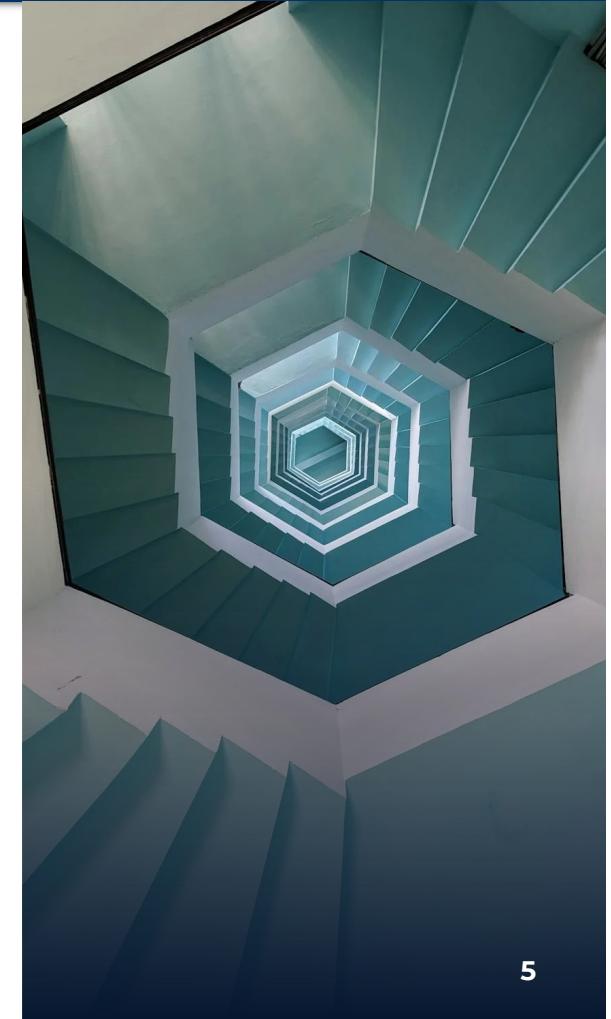
- | A többalakúság lehetővé teszi, hogy különböző típusú objektumokat közös interfészkről működtessünk
- | If/else és switch/case struktúrák helyett a többalakúsággal javasolt megoldani olyan eseteket, ahol kondíciók szabják meg hogyan járunk el

Ismétlés: Emlékszünk a Macska példára?

Elméleti Gyakorlat: Madár :Strucc/:Papagáj – Soroljunk képességeket és osszuk fel!

Különítsük el a multi-threading kódot

- | A **szekvenciális** folyamatok és kalkulációk nem sok előnyt nyújtanak
- | Ha a probléma lebontható kisebb feladatokra, melyek párhuzamosan futtathatók (vagy maga a probléma is masszívan párhuzamos), akkor a multi-threading hatalmas performancia megtakarítást eredményez
- | A multi-threading-ért felelős kódot viszont javasolt leválasztani az alkalmazás többi logikájától, hogy izolálhassuk az egyidejűségből eredő problémákat
- | **Például:** Matematika analitikai problémák, Képfeldolgozás, Renderelés



TERVEZÉSI SZABÁLYOK

■ Akadályozzuk meg az agyon-konfigurálhatóságot

- A konfigurálhatóság a rugalmasságot szolgálja
- De a túlzott konfigurálhatóság komplexitást növel és üzemeltetési **overhead-et**
- Adjunk értelmes alapértékeket
- Csak azokat a lehetőségeket hagyjuk nyitva, melyeket a felhasználóknak módosítaniuk is kell a használat során

■ Használunk felelősség injektálást

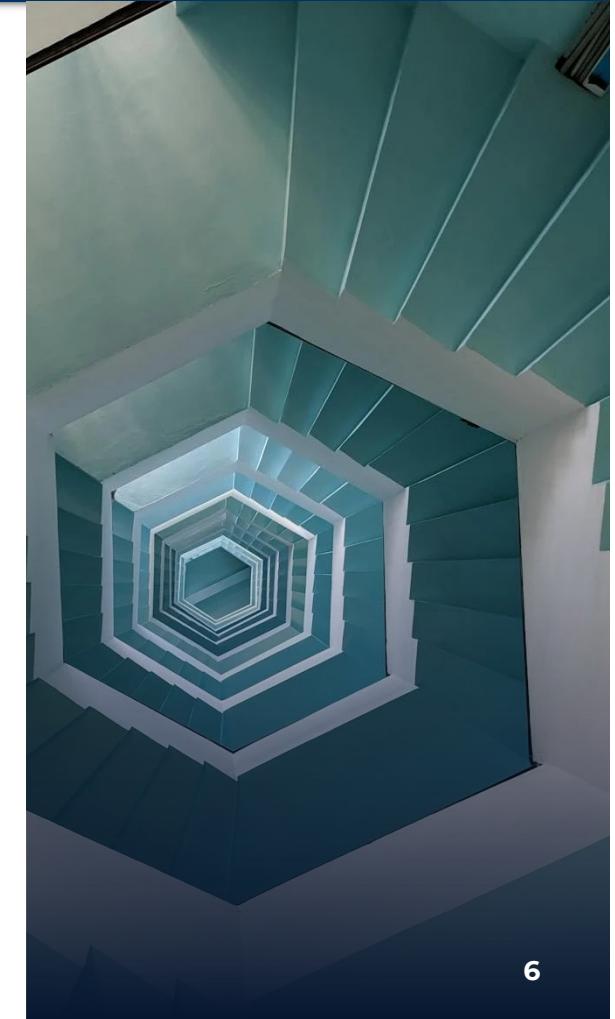
- Tervezási minta!
- A felelősség átadása (objektumok, szolgáltatások) egy külső komponensbe ahelyett, hogy a mi komponensünk hozna létre és kezelné azokat belsőleg
- Modularitás, tesztelés és karbantarthatóság szempontjából előnyös

■ Kövessük Demeter Törvényét

Kérdés: Mi Demeter Törvénye?

Példát is nézzünk: A – B – C – D modulok

- Csökkentjük a csatoltságot, és élvezzük ennek előnyeit.



OLVASHATÓSÁGI TIPPEK

| Légy konzisztens

- | Ha valamit egy bizonyos módon csinálsz, akkor minden hasonló esetén is így járj el
- | Kódolási stílus, névkonvenciók, **létrehozási minták**

| Használj beszédes változókat

- | A változóknak legyen értelmezhető nevük, mely tükrözi az értékük céljait
- | Reális értékeket alkalmazzunk bennük

| Zárd egységbe a határeseteket

- | **Edge case** és **Corner case**
- | A határeseteket nehéz számon tartani, ne hagyjuk, hogy szemetelje a főlogikánkat
- | Ezeket egy helyen egységezárva könnyebben értelmezhető a kódunk, könnyebben kezeljük az ezekből eredő hibákat és a határesetek feldolgozását

| Primitív típusok helyett használjunk dedikált értékű objektumokat

- | Ahelyett, hogy mindenhol sok-sok integert, stringet stb... viszünk fel
- | Készítsünk objektumokat, melyek dedikáltan tartalmazzák ezeket
- | Adjunk beszédes absztrakciókat és viselkedést, melyek sok helyen újrafelhasználhatók

| Kerüljük a logikai függőséget

- | Az osztályunk metódusai és komponensei ideális esetben függetlenek egymástól és a többi metódus belső állapotától vagy viselkedésétől

| Kerüljük a negatív kondíciókat – „If not”, „Unless” és társaik...



NÉVKONVENCIÓK

Használunk leíró és nem félreérthető neveket

- Az elnevezések pontosan tükrözzék az elem célját, funkcionálitását vagy jelentését

Tegyük láthatóvá a különbségeket

- A hasonló elemeknek legyen jól elkülöníthető (de konzisztens) nevük, hogy elkerüljük a félreírásokat, kavarodásokat
- Ideális esetben a név írja le a különbséget magát

Használj kiejthető neveket

- A szóban való használhatóság is fontos a csapatmunka szempontjából

Mond ki:

- functionhelper932475938459345D92F4
- UvuvwevweOnyetenyevweUgwemubwemOssas

Használj kereshető neveket

- Kellően egyedi és leíró nevek, melyeket gyorsabb keresni a kód bázisban

Magic Number helyett neves konstans

- Magic Number-nek hívjuk a magyarázat nélküli hard-code-olt értékeket.
- Ezeket foglaljuk konstansokba, melyek nevei leírják jelentésüket

Kerüljük az enkódolást, prefixeket és típus információt a nevekben

- A Hungarian notation-t mára már elhagyhatjuk
- Az új IDE-k és programnyelvek ezek alternatíváinak széleskörű kínálatát ajdák

Hello
my name is

```
class azthiszemösszeaddenéhanem  
class odinSzeme  
class azizéamihozé  
class crt1  
class hogyAFrancÜsseMeg
```

FUNKCIÓSZABÁLYOK

I Kicsi

- I A funkcióink legyenek a lehető legkisebbek és fókuszáltak

I Egy dolgot, de azt jó!

- I Egyetlen feladatot lássanak el, teljes egészében

Kérdés: Melyik alapelvünk mondja ki ezt?

I Kevesebb paraméter

- I Csoportosítsuk és fogjuk össze azokat a paraméter adatokat, melyekkel funkcióink dolgoznak adatstruktúrákká vagy alkalmazzunk alapértékeket

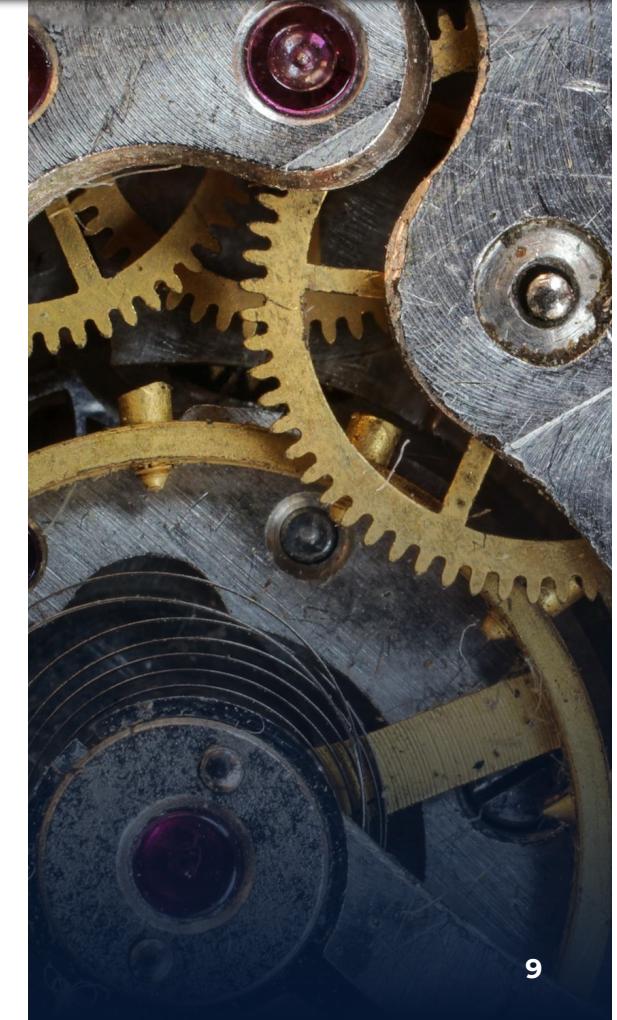
I Mellékhatások kerülése

- I Ideális esetben nincs mellékhatás

Kérdés: Mi a mellékhatás definíciója?

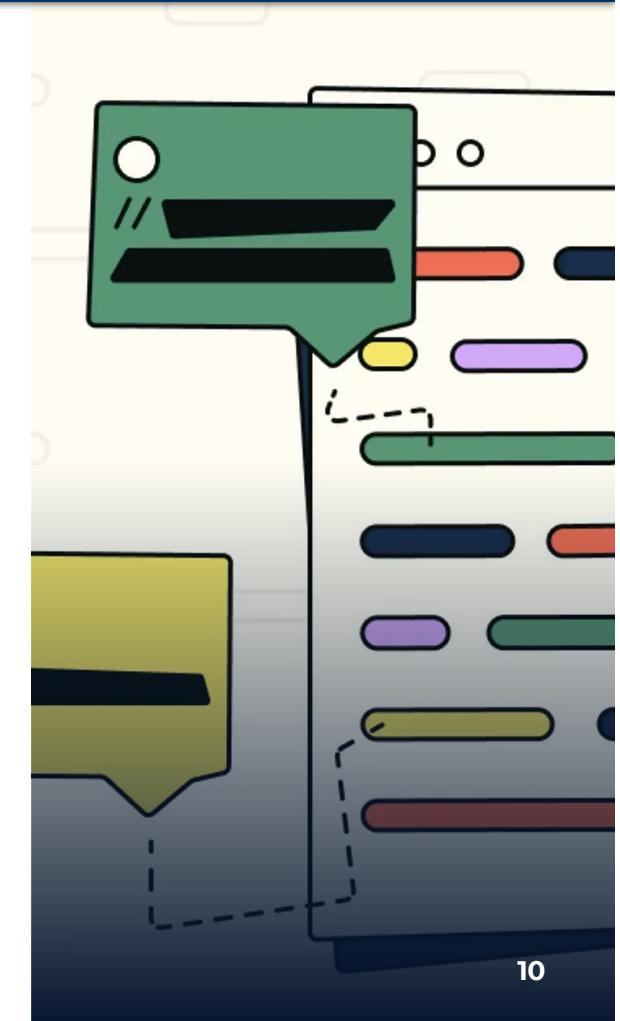
I Ne használj flag-eket

- I Kerüljük a boolean flagek és metódusokon belüli paraméterek használatát amikor a funkciónk viselkedését akarjuk irányítani
- I Bontsuk szét a metódusokat különálló metódusokra, melyek mindenike egy adott esethez tartozó viselkedésért felel



KOMMENT SZABÁLYOK

- | **Mindig próbáld magad kódban kifejezni**, ne kommentben
- | **Ne ismételgesd magad**, ha már egyszer leírtad
- | **Ne kelts felesleges zajt**, ha érthető miről van szó
- | **Ne használj zárókommenteket**, feleslegesen dolgozol az IDE helyett
- | **Alkalmazd a szándék leírására**, főleg ha a logika nehezen érthető (nem a kód!)
- | **Alkalmazd a bonyolult kód tisztázására**, például algoritmusok, adatstrukturák esetén
- | **Alkalmazd figyelmeztetésként**



FORRÁSKÓD STRUKTÚRA

Vertikális elválasztás

- Az összetartozó kódrészleteket vertikálisan csoportosítsd tömören, egymás mellett

A változókat a felhasználásuk helyéhez közel deklaráld

- Az első felhasználás helyéhez a lehető legközelebbi

Az egymással összefüggésben lévő- és a hasonló funkciók legyenek közel egymáshoz

Funkciók felülről-lefelé

- Magas szintű funkciók felül, alacsony szintűek alattuk
- A végrehajtásuk logikai menete szerint

Legyenek rövidek a sorok

A whitespace, mint eszköz

- Szorosan összetartozókat összehúzza
- Gyengén összetartozókat szétválasztja

Ne törd el az indentation-t

- Nem csak zavaró, de van ahol problémákhoz is vezet (pl.: Yaml)

```
using System;
```

```
namespace Introduction
```

```
{
```

```
internal class Program
```

```
{
```

```
    public static void Main()
```

```
{
```

```
    var test = new TestClass();
```

```
    Console.WriteLine(test.Foo());
```

```
//
```

```
}
```

```
}
```

```
}
```

```
}
```

OBJEKTUMOK ÉS ADATSTRUKTÚRÁK

Rejtsd el a belső struktúrát

- Egységbázárral rejtsünk és csak azokat az interfészket, viselkedéseket tegyük láthatóvá, melyeket külső irányból használni szeretnénk

Adatstruktúrák előnyei

- Például olyan osztályok, melyek metódusok nélkül reprezentálnak adatokat ([Példa](#))

Kerüljük a hibrid struktúrákat

- Hibrid az a struktúra, ami attribútumokat kever viselkedéssel egy elemen belül ([Példa](#))

Kicsi és SRP itt is érvényes

Az ős osztály ne tudjon semmit a gyermekiről

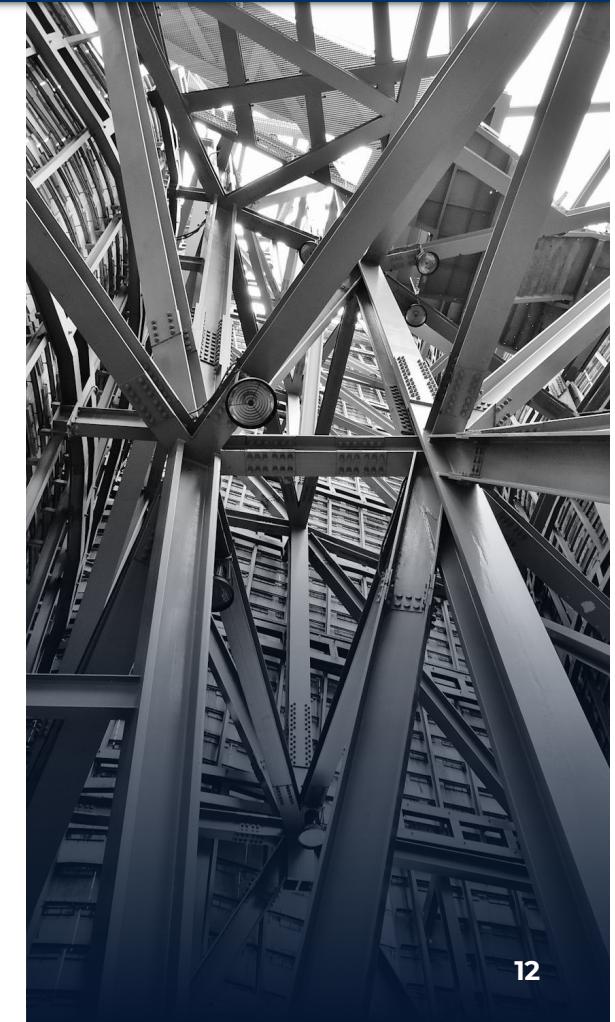
- Úgy tegyünk interfészket és viselkedéseket elérhetővé, hogy az ne következtessen a gyermek osztályok kinézetére, tartalmára

Inkább legyen több funkció

- Mintsem átadjunk egy halom adatot egyetlen funkciónak, ami kitalálja melyik viselkedést kell alkalmazni

Használunk nem-statikus metódusokat

- Az Instance method-ok egy adott osztályhoz csatoltak, nem annak példányaihoz
- A nem-statikus metódusok támogatják a többletlakúságot és öröklődést, alacsonyabb csatoltságot teremtve



TESZTEK

1 Assert tesztenként

- Minden tesztnak egyetlen specifikus viselkedést vagy kód részletet kell vizsgálnia
- Sikertelen teszt esetén így tudjuk hol keressük a hibát

Olvashatóság

- Tesztelt kód elvárt viselkedésének dokumentációját beleérte!

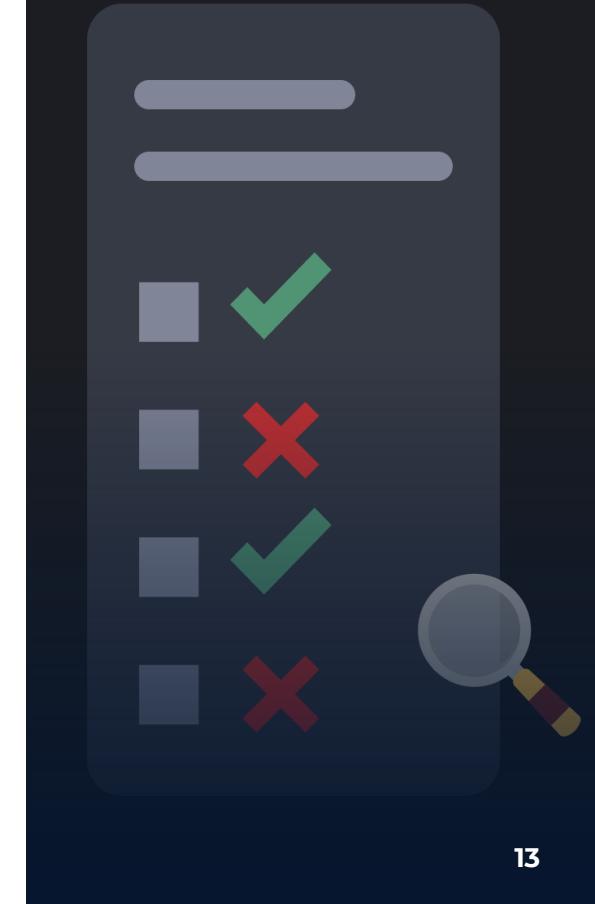
Gyorsaság

Független

- Minden teszt egymástól függetlenül kell, hogy futtatható legyen
- A teszteket környezeti faktorok ne manipulálják

Ismételhető

- Konzisztens eredményeket kell adniuk többszöri futtatás esetén is (ugyanazokkal a kondíciókkal persze)



KÓDSZAGOK

A rotható kód előjele

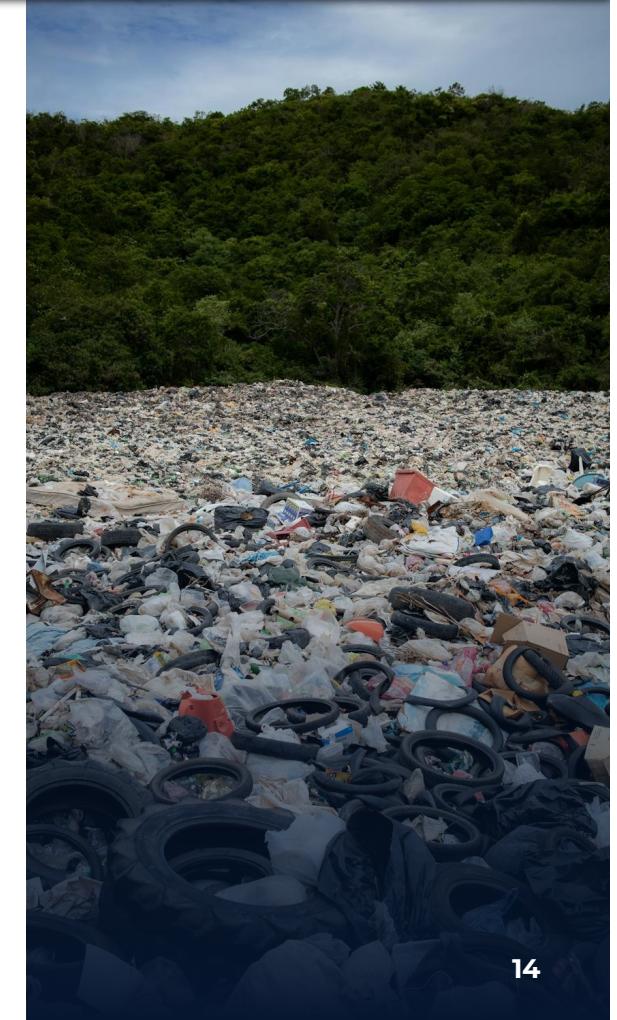
- ─ A rotható kódhoz a programozók már nemigazán szeretnek hozzányúlni
- ─ Tisztítása nagy erőfeszítéseket vesz igénybe, sőt a hibák javítása újabb hibákat okozhat
- ─ Szerencsére érezhető előjelei vannak

Kódszagok

- ─ Túl nagy metódusok
- ─ Túl nagy osztályok
- ─ Osztályok sok-sok mezővel
- ─ Duplikációk
- ─ Burjánzó if-else-if és switch szerkezetek
- ─ Megvalósításra programozás, Implementációs függőség
- ─ Erős csatoltság

Rossz szagok

- ─ Ezek már konkrét, kialakult problémákra hívják fel a figyelmet



ROSSZ SZAGOK

Merevség – Rigidity

- | A rendszert nehéz megváltoztatni
- | Egy-egy változtatás sokkal többidőbe telik, mint amennyit becsültünk a feladatra
- | Egy-egy változás nem várt nehézségekbe ütközik

Törékenység – Fragility

- | A rendszer nagyon érzékeny a változásokra
- | Egy-egy változás nagyon sok további változást von maga után
- | Egy-egy hibajavítás minden újabb és újabb hibákat szül (**Kérdés**)

Mozdulatlanság – Immobility

- | A rendszer egyes részeit funkciójuk szerint jó lenne újra felhasználni, de az újrafelhasználás komoly erőfeszítéseket igényelne.

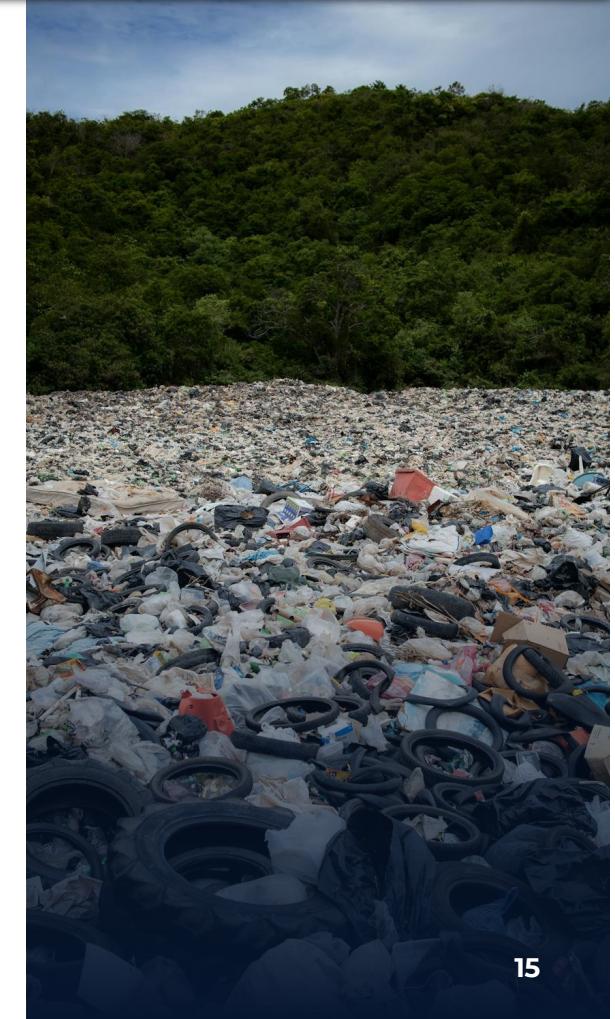
Viszkozitás – Viscosity

- | Egy-egy változtatás nagyon körülményes, inkább a „workaround” megoldást választjuk
- | Vagy olyan lassú a fordítás, hogy arra törekszünk, hogy csak kevés osztályt kelljen újrafordítani, ezért nem oda teszem a metódusokat, ahova kellene, hanem oda, amit könnyű újrafordítani.

Homályosság – Opacity

- | Nehezen olvasható, nehezen érthető kód.

Szükségtelen komplexitás – Needless Complexity



TESZTELÉS

UNIT TEST

TDD

TOVÁBBI FOGALMAK

ISMÉTLÉS: A TESZTELÉS ALAPELVEI

1. A tesztelés a hibák jelenlétét jelzi

- | Képes felfedni azokat
- | A szoftver minőségét és megbízhatóságát növeli

2. Nem lehet „mindent is” tesztelni

- | Általában csak a magas kockázatú és prioritású részeket teszteljük

3. Korai tesztek

- | Minél korábban jön ki a hiba, annál könnyebb és olcsóbb azt javítani
- | Már az elkészült dokumentációkat is lehet (és érdemes) tesztelni!

4. Hibák csoportosulása

- | Nincs végtelen időnk tesztelésre, emiatt a leginkább kitett és érzékeny rendszerekre kell koncentrálnunk
- | A bemenetek esetén használjunk szélső értékeket!

5. A „féregírtó” paradoxon

- | Ha ugyanazokat a teszteket futtatjuk, azok egyre kevesebb hibát fognak találni, ezért bővíteni kell őket!
- | Antibiotikumos kezelés esete – A fennmaradó 1% rezisztens, ők a legveszélyesebbek

6. A tesztelés függ a körülményektől

- | Másképp tesztelünk a környezet és idő függvényében

7. A hibátlan rendszer téveszméje

- | A hibák kijavítása nem egyenlő az igények teljesítésével!

ISMÉTLÉS: TESZTELÉSI TECHNIKÁK

Black-box

- ─ Csak a specifikációk ismertek, a forráskód nem
- ─ Specifikáció alapúnak is nevezzük
- ─ Szükség van lefordított szoftverre.
- ─ Tudjuk például, hogy egy adott bemenetre milyen kimenet az elvárt.

White-box

- ─ A Forráskód ismeretében végezzük a teszteket
- ─ Strukturális tesztelésnek is nevezzük
- ─ A lefedettség – A meglévő teszt esetek a struktúrának mekkora részét fedik le
 - ─ kódSOROK
 - ─ elágazásOK
 - ─ metódusOK
 - ─ osztályOK
 - ─ funkcióK
 - ─ modulOK

Grey-box

- ─ A forráskódnak csak egy része ismert

ISMÉTLÉS: A TESZTELÉS SZINTJEI

Komponensteszt

- ─ A rendszernek csak egy adott komponensére tér ki
- ─ Egységeszt – Funkcionális teszt
- ─ Modulteszt – Általában nemfunkcionális teszt

Integrációs teszt

- ─ Komponensek és rendszerek közötti interfészket teszteljük
- ─ Komponens integrációs teszt – Komponensek közötti hatások vizsgálata
- ─ Rendszer integrációs teszt – Rendszerek közötti hatások tesztje

Rendszerteszt

- ─ A szoftverterméket teszteli
 - ─ Követelményspecifikáció alapján
 - ─ Funkcionális specifikáció alapján
 - ─ Rendszerterv alapján

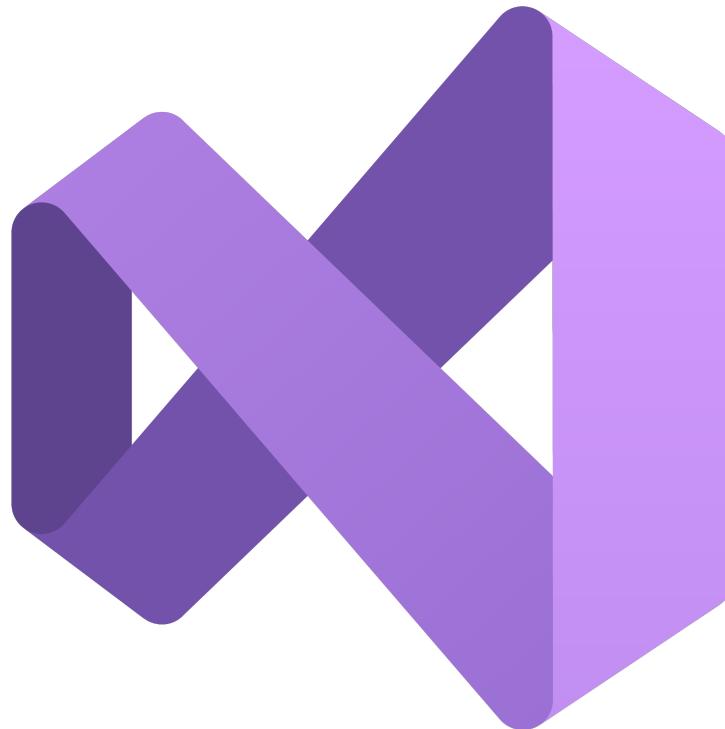
Átvételi teszt

- ─ Alpha, Beta, Felhasználói átvételi teszt (UAT) és Üzemeltetői átvételi teszt

Regressziós teszt

- ─ A módosítás nem okozott-e hibát
- ─ Komponenstesttől a Rendszertesztig bezárólag találkozhatunk vele

UNIT TEST GYAKORLAT



TESZTELÉS – ÚJ FOGALMAK

TDD és BDD

- | Test-Driven Development
- | Behaviour-Driven Development

AAA és GWT

- | Arrange-Act-Assert
- | Given-When-Then

SUT és Mock

- | System Under Test
- | Mock elemek teszteléshez

TESZTELÉS – ÚJ FOGALMAK

Test-Driven Development

- █ A teszteket már az implementáció előtt megírják
- █ A bukó teszt eset írja le a viselkedést és funkcionálitást, melyet el kívánnak érni
- █ A teszt teljesítéséhez szükséges legkevesebb kódot írják csak meg
- █ Gyakran használ egységes teszteket, melyek a kódot egységekre bontják (pl.: funkciók, metódusok) és azokat izoláltan tesztelik
- █ Célja, hogy a kód bizonyítottan helyesen működik és az elvártak szerint viselkedik
- █ Alacsony absztrakció

Behaviour-Driven Development

- █ A TDD kiegészítése, mely a rendszer viselkedésére fókuszál
- █ A stakeholders és végfelhasználók szemszögéből közelíti meg a tesztelést és fejlesztést
- █ A BDD támogatja az együttműködést az üzleti- és a fejlesztési csapatok között, hogy validálják azt egy közös nyelvet használva
- █ Specifications by example – A rendszer viselkedése egyszerű nyelven specifikálva- vagy GWT formátumban leírt esetekkel van definiálva
- █ A BDD fő célja validálni, hogy a rendszer teljesíti a megrendelő oldali elvárásokat
- █ Magas absztrakciós szint

TESZTELÉS – ÚJ FOGALMAK

AAA

- TDD esetén alkalmazzuk
- **Arrange** – Előfeltételek és a rendszer kezdeti állapota
- **Act** – Tesztelt cselekvés végrehajtása vagy metódus meghívása
- **Assert** – Ellenőrizzük az Act lépés eredményeit, hogy egyeznek-e az elvártakkal.

GWT

- BDD esetén alkalmazzuk
- **Given** – Előfeltételek és a rendszer kezdeti állapota
- **When** – Esemény, mely elindítja a tesztelt viselkedést
- **Then** – Várt eredmény és a When lépésben kapott eredmény

TESZTELÉS – ÚJ FOGALMAK

SUT

- System Under Test
- Ezt a rendszert teszteljük
- **Lehetnek olyan műveletek a rendszerben, melyek külső objektumokat vagy könyvtárakat hívnak**
- A teszteknek függetlennek kell lenniük a környezetüktől

Mock

- Mockingbird után kapta a nevét
- Lemásol egy kívánt működést
- **Behelyettesíti a SUT dependenciáit, hogy tesztelhető legyen a rendszer azok nélkül is**
- Így azokat nem kell példányosítani

Példa:

Egy dependenciát nem tudok példányosítani, mert:

- Nem elérhető
- Nem képezi a teszt scope-ját
- Olyan mellékhatása van, melyet nem tudunk tesztelni

A függvényünk egy Mock objektummal tudja helyettesíteni azt, mely csak úgy tesz, mintha az eredeti objektum lenne.

TDD

LÉPÉSEK

PIROS-ZÖLD-KÉK...

NAPLÓZÁS ÉS AOP

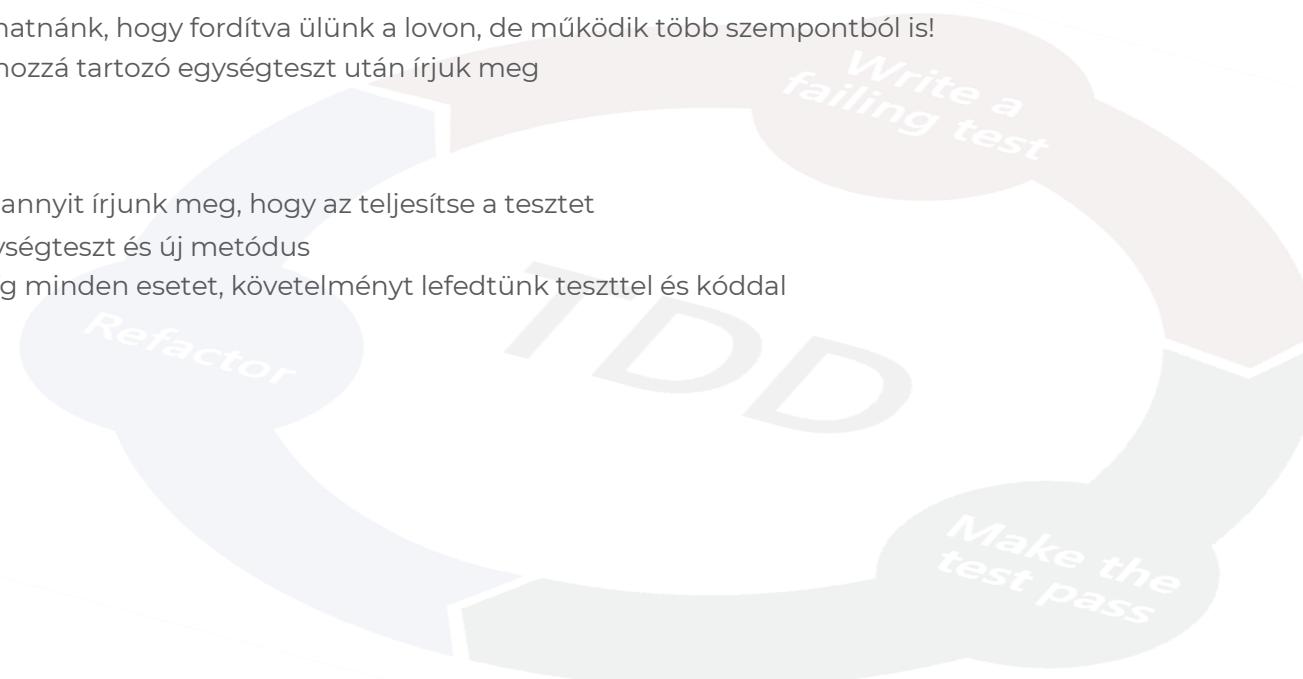
TESZTVEZÉRELT FEJLESZTÉS

Best-practice

- | Az agilis módszerekkel és a tiszta kód eszményével együtt terjedt el
- | Első hallásra mondhatnánk, hogy fordítva ülünk a lovon, de működik több szempontból is!
- | A metódust csak a hozzá tartozó egységeszt után írjuk meg

Alapelvek

- | A metódusból csak annyit írunk meg, hogy az teljesítse a tesztet
- | Ezután jön az új egységeszt és új metódus
- | Egészen addig, amíg minden esetet, követelményt lefedtünk teszttel és kóddal



TDD ÉS CLEAN CODE

I TDD Lépések

1. Írunk egységesztet
2. minden egységeszt az eddig nem tesztelt lehető legegyszerűbb eseteket írja le
3. Metódusból csak annyit írunk meg, hogy az épp átmenjen a tesztünkön

Loop

I Clean Code kiegészítés

1. Write no production code except to pass a failing test
2. Write only enough of a test to demonstrate a failure

Határértékek vs. minden eshetőség

3. Write only enough production to pass the test

Közös alapelv: Apró szeletekben fejlesszük a metódusokat azután, hogy írtunk hozzájuk egységeszteket.
A tiszta kód egyik alapja a TDD.

TDD ÉS CLEAN CODE

TDD Előnyök

- Mindig lesz egységesünk, kizárt, hogy elfelejtsük
- Tesztelő fejjel gondolkozunk a fejlesztő fej helyett
- Gyorsabb és egyszerűbb regressziós tesztek, avagy magabiztosabb hozzáállás a változtatásokhoz
- Széleskörű tesztlefedettség = Bizonyított magabiztosság a kód minőségében
- Részletes tesztek sora = Kevesebb idő kell a hibakereséshez
- Az egységeszt a legjobb dokumentáció egy fejlesztő számára

Kettős könyvelés

- Ugyanúgy, mint a könyvelésben, a követel és tartozik oldalon is ellenőrizve van a szándékunk és az értékek
- A TDD kettős-könyvelés szempontjából:
 1. Maga a teszteset megírásakor ellenőrizzük, hogy az teljesítse az adott eset követelményeit
 2. A metódus megírásakor a teszt esetre kódolunk, annak helyességét a teszttel újraellenőrizzük

Avagy: minden esetet lefedünk 1: Teszttel és 2: Kóddal is

TDD MÓDSZEREK

Piros – Zöld – Piros

- | **Piros** – A sikertelen egységeszt egy adott esetre
- | **Zöld** – Metódusban teljesített egységeszt
- | **Piros** – Újabb egységeszt más esetre

Piros – Zöld – Kék – Piros

- | **Kék** – Beérkezik a refaktorálás

TDD MÓDSZEREK

PIROS

- | Teszteset megírása
- | A követelményspecifikációk alapján
- | A tesztek szerződésként írják elő, hogy a megvalósításunk minek kell eleget tegyen
- | A tesztek kezdetben mindenig elbuknak, mert még nincs azokat kielégítő kódunk.

ZÖLD

- | A PROD kódot megírjuk
- | Úgy, hogy az a tesztek követelményeit teljesítse
- | Így már sikeresen átmennek a teszten

KÉK

- | A PROD kódot refaktoráljuk
- | Szépítjük, tisztítjuk, komplexitást csökkentünk
- | A tesztjeinket újra le lehet futtatni, hogy biztosak lehessünk róla, a refaktorálás nem okozott funkcióvesztést.

KÖSZÖNÖM A FIGYELMET!

8. óra – CLEAN CODE ÉS TDD

2024.04.19

Szalai Patrik

 szalai.patrik@uni-milton.hu