Tesztelés

Nagy Ákos
Microsoft Certified Trainer
https://dotnetfalcon.com

Miről lesz szó?

- Alapok, néhány alapfogalom
- Tesztelési szintek
  - Mikor és mit tesztelünk a rendszerben?
- Tesztelési technikák
  - Hogyan határozzuk meg a teszteseteket?
- Unit tesztelés
  - Unit tesztek tervezése, Unit teszt smell, Unit teszt keretrendszerek

Tesztelés

- Célja, hogy meghatározza a rendszer?, szoftver? helyességét?, működési hibát? minőségét?
- Szoftvertesztelésről fogunk beszélni
- Mi is az a szoftver?
  - Nagyon fontos: a mi szemszögünkből termék
- A szoftvertesztelés céljai:
  - Visszajelzést ad a szoftver hibáiról
  - Visszajelzést ad a szoftver minőségéről

Tesztelés

FONTOS!

- A tesztelés nem tudja bizonyítani, hogy a szoftverben nincsenek hibák
  - Próbálkozások: formális módszerek
- A tesztelés nem mondja meg, hogy hogyan kell kijavítani a hibát

Mikor tesztelünk?

- Életciklus modelltől függ
  - Nem anyag…
- Minél hamarabb
  - Minél kisebb részt kell letesztelni, annál könnyebb
  - Minél kisebb lépésekben halad a rendszer, annál kisebb az esélye, hogy valami elromlik
  - Ha a hibákat sokáig hurcoljuk, drágábbá válnak
- Folyamatosan
  - Nem csak a fejlesztés végén!
  - Ideális esetben minden tervezési és implementációs szakaszban
    - Ideális esetben folyamatosan ÉS a végén

Mi a probléma?

- Nincs idő
- Nincs pénz

- Nem megfelelő szaktudás
- Rosszul megtervezett architektúra („tesztelhetetlen a rendszer”)
- Specifikációk hiánya

Cél

- Határozzuk meg azt a legjobb teszteset halmazt, amivel a „legtöbbet” le tudjuk fedni
  - Határozzuk meg, mi szerint ☺
  - Tesztelési technikák

- Határozzuk meg a tesztelendő elemek körét
  - Tesztelési szintek

Tesztelési technikák

- Próbáljuk meg a legjobb teszteseteket megtalálni
  - A lehető legkevesebb teszttel a lehető legnagyobb részét megvizsgálni a rendszernek
- Gyakran kötődnek szintekhez
  - Unit tesztelés => strukturális technikák
  - Rendszertesztelés => funkcionális tesztelési technikák
  - DE! Unit teszteket is gyakran funkcionális technikák alapján állítjuk fel
- Gyakran intuitíven használjuk őket

Tesztesetek

- Miből áll egy teszteset?
- Tankönyvi definíció szerint:
  - Adott bemenet
  - Elvárt kimenet
  - Előfeltételek
  - Utófeltételek
  - Invariáns feltételek (elő + utófeltételként leírhatóak)
- Gyakorlatban
  - Adott bemenetre elvárt kimenet
  - Az előfeltételek ellenőrzése fontos (kódban implicit módon jelenik meg)
  - Utófeltételek ellenőrzése gyakran integrációs szintre szorul

Statikus technikák

- A kód futtatása nélkül történő tesztelés
  - Hogyan lehet akkor tesztelni?
  - Mit lehet tesztelni?

- Kicsit kakukktojás
  - Nem igazán vannak tesztesetek
  - Nincsenek „bemenetek”, elvárt kimenetek

Statikus technikák

- Miért használjuk?
  - Nagyon hatásosak
  - Nem csak a tesztelést segítik (szakmai tapasztalatcsere)
  - Általános technikák, a legtöbb típusú termékre használható

- Problémák
  - Nagyon drága
  - Nehéz jól kivitelezni
  - Megfelelő szakértelem nélkül nincs értelme

Statikus technikák

- Code review
  - Olyan technikák, amikor a kódot „kézzel” nézzük át
  - Vizsgáljuk a program belső szerkezetét
  - Megpróbáljuk megtalálni a hibákat
  - Nem csak a hibákra, a tervezési hiányosságokra is fény derülhet
    - A refaktorálási feladatok megtalálásának nagyon fontos módszere!!
    - Code smellek beazonosítása
- Walkthrough
  - A szerző vezeti az „eseményt”
  - Elmagyarázza, hogy mit miért csinált
  - A résztvevők kérdezhetnek, megprbálják megérteni a kódot
  - A szerző „megvédheti” megoldását, információt gyűjthet a többiektől

Statikus technikák

- Inspection
  - Gyakran formális folyamat
  - Tipikusan egyenrangú felek végzik
  - Elsődleges célja a hibakeresés
  - Fagan-féle inspekció:
    - Szerző
    - Olvasó
    - Moderátor
    - Inspektorok
- Peer review
  - Általában informális
  - Felülvizsgálja a kódot

Statikus technikák

- Minden esetben nehéz bevezetni
  - A szerzőknek megfelelően kell hozzáállni
  - A reviewereknek megfelelően kell hozzáálni
  - Nehéz megtalálni a formalitás mértékét
    - Ha túl formális, akkor éppen a lényege veszhet el
    - Ha nem elég formális, akkor hogyan értékelhetőek az eredmények, hogyan javítható a folyamat
  - Fontos az eszköztámogatás

Statikus technikák

- Szintaxis elemzés
  - A fordítási folyamat alapvető része
  - Szintén statikus elemzési technika
    - Nem futtatjuk a kódot
    - Adott szabályrendszernek való megfelelést vizsgálunk
- Statikus ellenőrzés
  - Valamilyen szabályrendszer szerint megpróbáljuk a kódot elemezni
  - Itt sem fut a kód
  - Gyakran beépített szolgáltatás az IDE-kbe
    - VS: soha nem érjük el a kódot, soha nem teljesül a feltétel, soha nem lesz null az eredmény
    - StyleCop: népszerű eszköz kifejezetten ilyen célokra
    - ReSharper
- Szimbolikus végrehajtás
  - Itt sem a program fut
  - Lényegében szimulálásra kerül a szoftver
  - Korlátozott használati lehetőségek

Dinamikus technikák

- Futtatjuk a kódot
  - Adott bementre adott inputot várunk el
- Tesztkörnyezetben történik a futtatás
- Tesztadatokkal

- Fontos feladat, hogy a tesztkörnyezet jól reprezentálja az éles környezetet
- A tesztadatok illeszkedjenek a tesztesetekhez

Dinamikus technikák

- Feltételezzük-e a kód ismeretét vagy sem?
- Fekete doboz tesztelés
  - A kódot fekete doboznak tekintjük
  - Nem tudjuk, hogyan működik, csak a bemenetet tudjuk megadni és a kimenetet tudjuk ellenőrizni
- Fehér doboz tesztelés
  - Látjuk a kód belsejét is, ez alapján próbálunk meg teszteseteket felállítani

Dinamikus technikák

- Ekvivalencia-osztály alapú tesztelés
  - Leggyakrabban ezt használjuk teljesen intuitíven
  - Matematikai háttér
  - Gyenge, erős
- Határérték tesztelés
  - Gyakran az elfogadható értékek határán hibázunk
  - Ciklusvégértékek, relációs jelek
  - Itt segíthetnek a statikus technikák is (pl.: típusellenőrzés)
  - min, min+, max, max-, nom
  - Független változók, fizikai jellegű mennyiséget reprezetálnak

Dinamikus technikák

- Robosztussági tesztelés
  - Határérték-tesztelés kibővítése
  - Kritikus kérdés: mi történik, ha nem megengedett értéket adnak meg?
  - max+, min-
- Ekvivalencia-osztályok meghatározásánál

Dinamikus technikák

- Döntési tábla alapú tesztelés
  - Fura technika
  - Nem ismerjük a program belső szerkezetét
  - De sejtenünk kell, hogy milyen feltételrendszer alapján áll elő az input
  - Segít, ha van jó specifikáció

Dinamikus technikák

- Fehérdoboz technikák
  - Feltételezzük, hogy ismerjük a kód belső felépítését
  - Nehéz intuitíven alkalmazni
  - A programkódot gráfként értelemezzük
  - Különböző módon „partícionáljuk”
  - Lefedettségi metrikákat kapunk

Áttekintés: tesztelési technikák

- Statikus technikák
  - Manuális
    - Walkthrough
    - Inspekció, review
  - Automatikus
    - Szintaxis analízis
    - Automatizált analízis
    - Szimbolikus végrehajtás
- Dinamikus technikák
  - Feketedoboz technikák
    - Ekvivalencia-osztályok
    - Határérték
    - Robosztusság
    - Döntési tábla
  - Fehérdoboz technikák
    - Metrika alapú lefedettség

Unit tesztelés

Nagy Ákos
Microsoft Certified Trainer
https://dotnetfalcon.com

Tesztelési szintek

- Fejlesztési modelltől, életciklusmodelltől függ
- Megpróbálja behatárolni, hogy mit kell tesztelni
  - Az egyes metódusokat
  - Interakciókat az egyes komponensek között
  - Felhasználói igényeket
  - stb.

Tesztelési szintek

- Általában megjelennek az alábbi, vagy hasonló fogalmak:
  - Unit tesztek (egységtesztek)
    - Teszteljük a „legkisebb egységet”
  - Integrációs tesztek
    - Teszteljük az egységek közötti integrációt
  - Rendszertesztek
    - Teszteljük a rendszer egészét

Unit tesztelés

- Mit tekintünk a legkisebb egységnek?
  - Függ a szoftverfejlesztési paradigmától is
    - OO esetben pl.: osztály vagy metódus
    - Nem OO esetben?
  - Gyakori szabályok:
    - „Amit nem osztanál szét emberek között”
      - Nem mindig jó definíció
    - „Ami önállóan is tesztelhető”
      - Ha rosszul van megtervezve az architektúra, akkor nem triviális

Unit tesztelés

Kent Beck, Smalltalk:

A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed.

A “unit” is a method or function.

(Wikipedia)

Unit tesztelés

Mitől lesz jó egy unit teszt?

- Automatikus, megismételhető, automatikusan megismételhető
  - Véd a regressziótól
  - „Egy gombnyomásra fut”
- Lehetőség szerint egyszerű
  - Egyszerű a tesztadatokat összeszedni
- Karbantartható
  - Cél, hogy védjen a regressziótól
- Független
  - Nem függenek egymástól a tesztek
  - Minden input rendelkezésre áll a tesztkörnyezetben (nincs DB!)

Unit tesztelés

Mitől lesz jó egy unit teszt?

- F.I.R.S.T (Robert C. Martin)
  - Fast: Gyorsan lefut. Ha nem futnak elég gyorsan, akkor úgysem fogod őket futtatni.
  - Independent: Függetlenek. A tesztek nem függenek egymástól, sem az eredményeik, sem pedig az inputjaik.
  - Repeatable: Megismételhetőek. Futtathatom őket a saját gépemen, a build serveren, vagy akár a laptopomon.
  - Self-validating: Önvalidálóak. Ha lefut, akkor zöld vagy piros. Nem kell bogarászni semmilyen hosszú outputot.
  - Timely. „Időszerűek”. Nem csak a fejlesztés végén, hanem közben is írjuk őket.

Unit tesztelés

Ki végzi a unit tesztelést?

- Tesztelési módszertantól, szoftverfejlesztési modelltől függhet, de
  - Általában a kód ismeretét igényli
  - Éppen ezért leggyakrabban maguk a fejlesztők végzik

Unit tesztelés

Technikák

- Hogyan tudunk jó unit teszteket készíteni?
- A tervezés során hozott döntések kritikusak
  - Lazán csatoltság
- De az egyes teszteknek önállóan kell futniuk
  - A függőségeket valahogyan kezelni kell

Unit tesztelés

Stub, fake, mock, dummy

- Vezessünk be tesztobjektumokat
- Sokféle terminológia
- Martin Fowler:
  - Dummy: egy nem használt objektum
  - Fake: olyan objektum, amely valamilyen „hack” megoldást tartalmaz (pl.: in-memory adatbázis)
  - Stub: Adott inputra ad adott válaszokat, esetleg megjegyzi őket
  - Mock: Egy olyan objektum, amelynek viselkedését be tudjuk állítani

Unit tesztelés

Mi számít külső függőségnek?

- Cél: tesztkörnyezet, tesztfutás teljes kontrollálása
- Külső függőség:
  - Fájlrendszer
  - Adatbázis
  - Véletlenértékek generálása
  - Időlekérdezés

Unit test smells

- Hasonló a code smell fogalmához
- „Valami nem stimmel”
- A legtöbb hatással van több faktorra is
- Az irodalom klasszikusan 3 kategóriába sorolja őket
  - Olvashatóság
  - Karbantarthatóság
  - Megbízhatóság

Unit test smells

- Olvashatóság
  - A Unit test is kód!
  - Nem production kód, de ugyanúgy karban kell tartani, módosítani
  - Ehhez pedig meg kell felelnie az olvashatóság követelményeinek
    - Általános követelmények a kód olvashatóságára vonatkozóan
    - Speciális követelmények a unit tesztek olvashatóságára vonatkozóan
  - Fejezze ki a kód azt, amit csinálni szeretne

Primitív feltételezések (primitive assertions)

Az assertek valamilyen világos szándékot kell, hogy kifejezzenek

- **Világos-e a szándék?**

  ```
  var list=GetMySuperList();
  Assert.AreNotEqual(list.IndexOf(18),-1)
  ```

- **Világosabb szándék:**

  ```
  var list=GetMySuperList();
  var isInList=list.Contains(18);
  Assert.IsTrue(isInList);
  ```

Bitenkénti feltételezések (bitwise assertions)

- A primitív feltételezések egyik formája
- A bitenkénti műveleteket is érdemes kiszervezni és aztán ráellenőrizni

Túlzásba vitt feltételezések (Hyperassertions)

Az assertek olyan részletekkel is foglalkoznak, amelyekkel nem kellene

```
//do something, which logs some data into the log.txt
var logContent=”…”;
logContent.Append(”…”)
…
var actualLogContent=GetActualLogFileContent();
Assert.AreEqual(logContent,actualLogContent);
```

Incidental details (fölösleges részletek)

- **A teszt célja valamilyen feltételhalmaz ellenőrzése**
- **Ha túl sok egyéb kód van a tesztben, nehezebben lehet a valódi célját megtalálni**

- **Mit tehetünk?**
  - A teszt egyes részeit (pl.: setup) szervezzük ki privát metódusokba
  - Megfelelő, leíró nevek használata
  - Single level of abstraction

Setup sermon (setup ceremónia)

- Ez az incidental details problémának az a speciális esete, ami kifejezetten a test setupra vonatkozik (testadatok betöltése, mock-ok felkészítése a tesztelésre)
- Ugyanazok a megoldási lehetőségek

Személyiséghasadás (Split personality)

- A teszt egyszerre több dolgot is ellenőriz
- Nem ugyanaz, mint a több assert (lehet, hogy egy feltételt csak több asserttel lehet ellenőrizni)!
- Egy hibás tesztnek mindig meg kell tudnunk mondani az okát
  - Ld. Single responsiblity
- Szedjük szét két, független tesztbe!

Logika darabolódása (Split logic)

- A teszt persze nem production kód
- Szeretjük egyszerűen tartani
  - Érdemes inline-olni, amit csak lehet (de ne legyen a vége incidental details)
- Ne legyen hosszú metódushívási lánc a tesztekben

Magic numbers (mágikus számok)

- Ez ugyanaz a probléma, mint ami jelentkezik production code esetében is
- Általánosítható „magic literals” koncepcióként
- Egyszerű megoldás: adjunk nekik nevet, amely leírja a célját

Overprotective test (túlaggódó teszt)

A teszt túlaggódja a tesztesetet

```
var x=myList.SingleOrDefault(t=>t.Age>=18);
Assert.IsNotNull(x);
x.DoSomething();
```

Fölösleges kikényszeríti, hogy ne legyen null, a következő sor miatt úgyis hibára futna a teszt

Unit test smells

- **Karbantarthatóság**
  - A unit tesztek fontos előnye, hogy védenek a regressziótól…
  - … de csak ha megfelelően karban vannak tartva
  - Alapvető karbantarthatósági követelmények
    - Elsősorban a tesztekre vonatkozó követelmények

Duplication (duplikáció)

- Ugyanaz a probléma, mint a production kód esetében
- Ugyanazok az okok, a megoldások is

Conditional logic (feltételes logika)

- **A feltételek a tesztekben általában rossz gyakorlatra utalnak**
- **Ha a két részben más az assert, akkor különösen**

```
var currentDate=DateTime.Now;
if (currentDate.DayOfWeek==DayOfWeek.Monday)
   Assert.IsTrue(a);
else
  Assert.IsFalse(a);
```

Akkor ez legyen két teszteset, olyan tesztadatokkal, amik az egyik ill. másik feltételágat eredményezik

- Persze lehet közös az előkészítő rész

Flaky test (megbízhatatlan teszt)

- **Egy tesztnek mindig ugyanazt az eredményt kell adnia adott bemenetre**
- **Ez a teszt hibára fut a hétfő esti automatizált builden, de kedden már jól fut**

```
var currentDate=DateTime.Now;
if (currentDate.DayOfWeek==DayOfWeek.Monday)
   Assert.IsTrue(a);
else
  Assert.IsFalse(a);
```

- Miért?
  - Nincs megfelelően kontrollálva a teszt
    - Olyan adatok vannak benne, amiket nem mi állítunk be (idő/dátum, véletlen)

Crippling file path (megrokkantó elérési útvonal)

- Az elérési útvonal bele van kódolva a tesztbe
  - Más gépén is megvan a fájl?
  - Ugyanabban a mappában?
- Ez éles kódban is rossz gyakorlat
  - Tesztben is elfogadhatatlan
  - A konstansba kiszervezés is rossz (automatizáltnak kell lennie a tesztnek!)
- Ha lehet, ne használjunk fájlokat sem (elvégre ez külső függőség)
- Használjunk relatív elérési útvonalakat

Persistent temp files (nem-ideiglenes ideiglenes fájlok)

- Ha a teszt ideiglenes fájlokkal dolgozik, akkor azokat le kell törölni
- Akkor is, ha az assert elszállt (ez sok keretrendszerben lényegében egy kivételt dob!!)

- Jó gyakorlat, ha a fájl neve/helye egyértelműen mutatja, hogy az egy ideiglenes fájl

- Crippling file path probléma felléphet (van joga a tesztkörnyezetnek írni a megadott helyre?)

- Töröljük az ideiglenes fájlokat a teszt futása előtt is!
- Használjunk valamilyen egyedi nevet, toldalékot a névben!

Sleeping snail (alvó csiga)

- A tesztek fontos tulajdonsága, hogy gyorsak
- Ha lassú a teszt, az semmiképpen nem jó
- Mitől lehet lassú a teszt?
  - File I/O
  - Thread.Sleep()
  - Egyéb szálkezelési problémák (Flaky test)

Pixel perfection (pixelpontos egyezés)

- Primitive assertion és magic numbers speciális esete
- Általában generált képek, koordináták ellenőrzésénél
- Egyenes vonal?

Parameterized mess (paraméterezett rendetlenség)

- Paraméterezzük a tesztjeinket…
  - Hogy elkerüljük a duplikációt
- … de túlzásba visszük
- Megoldás: ne vigyük túlzásba
  - Ne használjuk a paraméterezett teszteket, ha ront az olvashatóságon
    - Incidental details, setup sermon, split logic

Lack of cohesion in methods (kohézió hiánya a metódusok között)

- A teszteseteket általában nagyobb logikai egységekbe szervezzük
  - Test fixture, Test Class
- Ezekben a teszteseteknek függetleneknek kell lenniük
- De legyen a tesztesetek között logikai kohézió
- Szedjük szét több egységre a teszteseteket
  - A megosztott logikát szervezzük ki egy harmadik helyre

Unit test smells

- Megbízhatóság
  - A unit tesztek feladata, hogy bízhassunk abban, jól működik a kód
  - De mi van, ha csal a unit teszt?

Commented-out tests (kikommentezett tesztek)

- Módosul a kód, nem fordul a teszt
- Kikommentezzük
- Lényegében a unit tesztek legnagyobb előnyét veszítjük így el (regresszió)

- Módosul a kód, nem fut a teszt
- Kikommentezzük
- De hát az a dolgunk, hogy jó legyen továbbra is

- Persze lehet, hogy már nincs rá szükség
- De akkor töröljük inkább ☺

Misleading comments (félrevezető megjegyzések)

- Megjegyzések általában érdekes téma
- Ha van egy félrevezető megjegyzés a tesztben, az az elvárásokat félreviszi

Never failing test (mindig helyesen futó teszt)

- A teszt mindig zöld…
- … de nem azért, mert mi ügyesek vagyunk, hanem lehetetlenség piros futást eredményezni

- Mit mond az ilyen teszt?
  - Semmit ☺

- Magas lefedettség, elégedett menedzsment
- De használhatatlanok a kód karbantartásához vagy hibakereséshez

Shallow promises (gyenge ígéretek)

- A teszt mindig zöld…
  - … mert nem csinál semmit
  - … mert nem tesztel semmit (nincs benne assert)
  - … mert nem annyira alapos, mint a név suggallja

- Mit mond az ilyen teszt?
  - Semmit ☺

- Magas lefedettség, elégedett menedzsment
- De használhatatlanok a kód karbantartásához vagy hibakereséshez

Lowered expectations (lejjebb adott elvárások)

- Módosul a kód, nem fut helyesen a teszt
- Nem kommentezzük ki, mert az unit test smell
- Hanem az asserteket kiszedjük ☺

- Mit mond az ilyen teszt?
  - Semmit ☺

- Magas lefedettség, elégedett menedzsment
- De használhatatlanok a kód karbantartásához vagy hibakereséshez

Platform prejudice (platform előítéletek)

- Ha több platformra fejlesztünk, akkor mindegyiken tesztelni kell
- Minden platformra egyenlő feltételekkel

- Ne tegyünk egyes platformokra speciális megkötéseket vagy épp engedményeket

Conditional tests (feltételes tesztek)

- Olyan tesztek, amelyekben rejtőzik valahol egy feltétel, amitől a teszt működése a névtől nagyon eltérő eredményt hozhat

- Platform prejudice ennek speciális esete
  - TestLogin vs TestAndroidLogin

OO ismétlés

Nagy Ákos
Microsoft Certified Trainer
https://dotnetfalcon.com

Egységbe zárás

- Az adatok és a rajtuk műveleteket végző műveletek csoportosítva vannak => egységbe vannak zárva
- Alapegység: osztály
- Alapvető feltétele az átláthatóságnak és a karbantarthatóságnak!

Öröklés

- IS-A típusú kapcsolat
- Manager is an employee – a menedzser alkalmazott
- Amit az employee tud, azt tudja a menedzser is
- Alapvető feltétele a kódduplikáció elkerülésének
  - Részben a hierarchiában szereplő kódok miatt
  - Részben a hierarchiát használó kódok miatt

Polimorfizmus

- Szoros kapcsolatban az örökléssel
- Virtual, abstract, override kulcsszavak
- Attól függ, hogy mit csinál egy objektum, hogy mi az ő pontos típusa
  - public int DoSomething(A myParam)
  - myParam típusa A vagy A-nak bármely leszármazottja
  - myParam objektum viselkedése polimorfikus, a valódi típustól függ

S.O.L.I.D.

Nagy Ákos
Microsoft Certified Trainer
https://dotnetfalcon.com

Single responsiblity

- Válasszuk szét a felelősségeket
- Ez így összhangban van az OO egységbe zárás fogalmával
- Próbáljuk meg korlátozni azoknak a változtatásoknak a számát, amik miatt az osztályt módosítani kell

Open/closed principle

- Open for extension
- Closed for modification

- Csak olyan helyeken tudjak belenyúlni, amik tervezett kibővítési pontok (pl.: örökléssel, metódusfelülírással)
- A valóban polimorfikus részeket lehessen ennek megfelelően kezelni!

Liskov substitution

- Legyen q(x) egy tulajdonság, amit be lehet látni T típusú x objektumokra. Ekkor q(y)-nak beláthatónak kell lennie az S típusú y objektumokra, ahol S altípusa T-nek.
- Magyarul: Az előfeltételeket nem lehet szigorítani, az utófeltételeket nem lehet gyengíteni
- Még inkább magyarul: ha valahol eddig egy T típust használtam, de most S típust szeretnék inkább használni, akkor annak „működnie kell” (pl.: erősen típusos nyelvek esetén fordul, a korábbi unit tesztjeim jók)
- Kontravariancia, kovariancia
- Kivételek?
- Négyzet-téglalap?

Interface seggregation

- Sok kis interfész
- Kevesebb a függőség

Dependency inversion

- A magasszintű moduloknak nem szabad alacsonyszintű moduloktól függeniük. Mindkettőknek absztrakcióktól kell függeniük.
- Az absztrakcióknak nem szabad részletektől függeniük. A részletes implementációknak kell az absztrakcióktól függeniük.
- Magyarul? ☺

Tesztelhető alkalmazások készítése

Nagy Ákos
Microsoft Certified Trainer
https://dotnetfalcon.com

Dependency Injection

- Alapfogalmak, módszerek, definíciók
- Minták, anti-patternek, refaktorálás

Miről lesz szó?

- Általános bevezetés
  - Értjük hogy mi ez? Mire jó? Miért van? Hogyan kapcsolódik az OO-hoz? Mi a terminológiája? Hogyan segíti a tesztelést?
- Minták
  - Jó és rossz gyakorlatok, refaktorálási megoldások

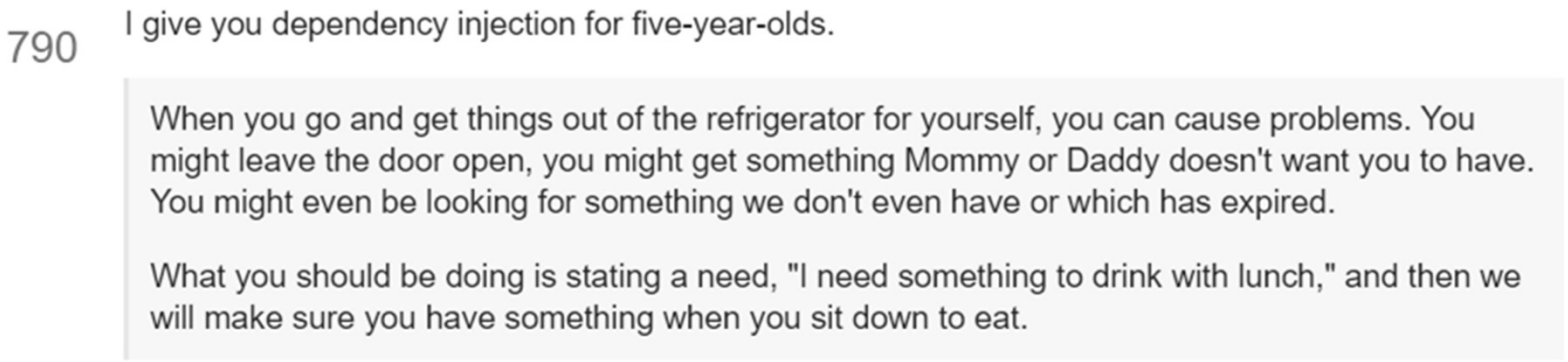
Alapozzunk…

- Mi az a DI?
- Mi nem a DI? Tévhiedelmek…
- A DI célja
- A DI előnyei
- Mikor alkalmazzuk?

Mi az a DI?

https://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old

How to explain dependency injection to a 5 year old?



Mi az a DI?

Szoftvertervezési elvek összessége, amelyek lazán csatolt kód írását teszik lehetővé.

Miért fontos a lazán csatoltság?

Hogyan lesz a kód lazán csatolt?

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:

Program to an interface, not an implementation

Mi nem a DI?

- A DI csak a late binding miatt kell
- A DI csak a tesztelhetőség miatt kell
- A DI igazából egy felpimpelt abstract factory
- A DI valamilyen containerrel együtt megy

Nem csak late bindinghoz

Late binding: anélkül tudjuk kicserélni az alkalmazás egyes részeit, hogy újra kellene fordítani

Klasszikus példa: Adatvezérelt alkalmazás, amely egy config beállítástól függően SQL Server vagy Oracle db-vel kommunikál

A DI ezt lehetővé teszi – de nem csak ezt! Ld. következő érv ☺

A rosszul kivitelezett DI nem teszi lehetővé ☹

Nem csak unit tesztek miatt

Ld. előző érv: ha nem unit tesztelsz, a late binding még akkor is igaz és jól jöhet ☺

Ha teszteket nem is írsz, de a DI eredeti definíciója miatt bármilyen karbantartási feladatnál hasznos lesz

Felpimpelt abstract factory

A DI pont a factory ellentéte. A factory igazából egy olyan szolgáltatás, amelynek a segítségével el tudsz kérni más szolgáltatásokat. Ld. Service Locator

A DI egy olyan strukturálási forma, ahol a szolgáltatást használóktól kikényszerítjük, hogy adják meg a szükséges szolgáltatásokat, ahelyett, hogy nekünk explicit módon el kellene kérni.

Kell egy container

Nem.

A container hasznos és sokat tud segíteni, sok feladatot átvállal és enélkül a gyakorlatban nem érdemes nekiindulni egy nagyobb projektnek.

De nem szükséges!!

Mikor használjuk (és mikor ne)?

- Felmerül a kérdés: akkor most nem használhatom a List<T>-t sehol? XmlWriter? Bármilyen .NET osztály?

A válasz sokszor attól fog függeni, hogy hol akarod pontosan használni és mire.

Stabil függőségek

- Stabil egy függőség ha (mindegyik igaz):
- Már létezik
- Az új verziókban nem lesz breaking change
- Determinisztikus algoritmusokkal dolgozik
- Még elméleti esélye sincs annak, hogy ezt ki fogod cserélni

- Például: XmlWriter ☺, vagy általában a .NET BCL osztályai (de nem mind, és nem mindenhol használva)

Változó (volatile) függőségek

- Volatile egy függőség ha (mindegyik igaz):
- A függőség miatt szükség van valamilyen runtime vagy környezet felállítására
- A függőség még nem létezik, vagy éppen fejlesztés alatt van.
- A függőség nem elérhető mindenhol
- Nemdeterminisztikus viselkedésű komponens.

Stabil és nem stabil függőségek

- A DI célja kevésbé absztrakt módon megfogalmazva az, hogy a lazán csatoltságot úgy segítse elő, hogy a volatile függőségek kezelésére ad egy megoldást (hiszen éppen azok miatt nem élvezhetjük a lazán csatoltság előnyeit).

DI 3D

- A DI „három dimenziója”; három olyan aspektus, amivel foglalkozik, hogy lehetővé tegye a lazán csatoltságot

- Objektumkompozíció
- Életciklusmenedzsment
- Interception

Objektumkompozíció

- Ahhoz, hogy valóban előnyként jelenjen meg a lazán csatoltság, ahhoz persze össze kell valahogy építeni a dolgokat.

- Ez volt az „első” dolog, aminek kapcsán a DI-t Martin Fowler leírta.

Életciklusmenedzsment

- Azzal, hogy feladtuk a függőségek explicit példányosítását és rábíztuk másra ennek a kívülről történő átadását, elvesztettük az irányítást az életciklus felett is (majdnem mindig).

- Mi legyen a felszabadítandó dolgokkal?
- Mi legyen azokkal a felszabadítandó dolgokkal, amiket közösen használnak?
- Egyáltalán lehet közösen használni dolgokat? Hányszor lehet újrahasználni valamit?

Interception

- Azzal, hogy van valaki, aki a példányosítást elvégzi helyettünk, azzal van egy központi hely, ahol a példányokat egységesen, még felhasználás előtt módosíthatjuk (becsomagolhatjuk, kicserélhetjük).

- Ez persze csak azért lehet, mert interfészek fölé programozunk ☺

Minták, antipatternek és refaktorálás

Constructor injection

Hogyan tudjuk garantálni azt, hogy az éppen fejlesztett komponens függőségei mindig a helyükön lesznek?

Úgy, hogy elvárjuk a komponens felhasználóitól, hogy adják meg a függőségeket constructorargumentumként.

Constructor injection

```
private readonly IProductRepository productRepository;
0 references
public ProductService(IProductRepository productRepository)
{
    if (productRepository == null)
    {
        throw new ArgumentNullException("productRepository");
    }

    this.productRepository = productRepository;
}
```

Constructor injection

Mikor használjuk?

Kötelező függőségeknél, amelyeknek nincs jó lokálisan használható alapértelmezett értékük.

Constructor injection

☺ Garantáltan meglesz a jó függőség.

☺ Könnyű implementálni.

☹ Sokszor módosítani kell az alkalmazást emiatt (sok helyre kellene default ctor, de az nem lesz) => van rá framework segítség

☹ Az egész objektumgráf létrejön egyben => ez egyrészt nem akkor baj (teljesítményszempontból sem), másrészt pont ezt volt a cél igazából

Property injection

Hogyan tudunk opcionális függőségeket megadhatóvá tenni, ha van jó lokális default?

Írtható property-ként vesszük fel a függőséget, amelynek lesz egy default értéke.

Property injection

```
0 references
public class ProductService
{
    private IProductRepository productRepository;

    0 references
    public IProductRepository ProductRepository
    {
        get { return productRepository; }
        set { productRepository = value; }
    }
}
```

Property injection

```
public class ProductService
{
    private IProductRepository productRepository;

    0 references
    public IProductRepository ProductRepository
    {
        get
        {
            if (productRepository == null)
                productRepository = new DefaultProductRepository();
            return productRepository;
        }
        set
        {
            if (value == null)
                throw new ArgumentNullException("value");
            if (this.productRepository != null)
                throw new InvalidOperationException("Already set");
            productRepository = value;
        }
    }
}
```

Property injection

Mikor használjuk?

Nem kötelező függőségeknél, amelyeknek van jó lokálisan használható alapértelmezett értékük.

Property injection

☺ Könnyű megérteni

☺ Könnyű használni

☹ Nehezebb implementálni

☹ Kevésbé robosztus

☹ Kevésbé rugalmas (kell referencia a default érték típusára)

Method injection

Hogyan tudunk olyan függőségeket megadni egy komponensnek, amelyek műveletenként mások?

Úgy, hogy a műveleteknek adjuk át paraméterként.

Method injection

```
public class ProductService
{
    2 references
    private void ThrowIfServiceIsNull(object service, string serviceName)...
    0 references
    public void DoSomething(IProductRepository productRepository)
    {
        ThrowIfServiceIsNull(productRepository, nameof(productRepository));
        productRepository.DoSomething();
    }

    0 references
    public void DoSomething2(IProductRepository productRepository)
    {
        ThrowIfServiceIsNull(productRepository, nameof(productRepository));
        productRepository.DoSomething2();
    }
}
```

Method injection

Mikor használjuk?

A függőség más-más műveletenként.

Method injection

☺ Műveletspecifikus kontextust lehet megadni.

☹ Limitált alkalmazhatóság

☹ A hívónak a metódushívás helyén kell megadni a függőséget, ez kicsit szembe megy a korábbi elvekkel (minél kevesebb Resolve, minél korábban, senki nem Resolve-ol saját maga)

Service aggregation / façade services

```
public abstract class TimeProvider
{
    1 reference
    public abstract DateTime UtcNow { get; }
}

0 references
public class DefaultTimeProvider : TimeProvider
{
    1 reference
    public override DateTime UtcNow ⇒ DateTime.UtcNow;
}

1 reference
public abstract class Logger
{
    1 reference
    public abstract void Log(string s);
}

0 references
public class ConsoleLogger : Logger
{
    1 reference
    public override void Log(string s)
    {
        Console.WriteLine(s);
    }
}
```

Service aggregation aggregation / façade services

```
1 reference
public class ProductService
{
    0 references
    public ProductService(TimeProvider timeProvider, Logger logger)
    {

    }
}
```

```
public abstract class TimeProvider
{
    1 reference
    public abstract DateTime UtcNow { get; }
}

0 references
public class DefaultTimeProvider : TimeProvider
{
    1 reference
    public override DateTime UtcNow => DateTime.UtcNow;
}
```

```
reference
ublic abstract class Logger
{
    1 reference
    public abstract void Log(string s);
}

0 references
public class ConsoleLogger : Logger
{
    1 reference
    public override void Log(string s)
    {
        Console.WriteLine(s);
    }
}
```

Service aggregation / façade services

```
2 references
public interface IApplicationServices
{
    2 references
    Logger Logger { get; }
    2 references
    TimeProvider TimeProvider { get; }
}
```

```
public class ProductService
{
    0 references
    public ProductService(IApplicationServices appServices)
    {

    }
}
```

```
1 reference
public class ApplicationServices : IApplicationServices
{
    0 references
    public ApplicationServices(Logger logger, TimeProvider timeProvider)
    {
        this.Logger = logger;
        this.TimeProvider = timeProvider;
    }
    2 references
    public Logger Logger { get; }
    2 references
    public TimeProvider TimeProvider { get; }
}
```

Anti-patternek

Control freak

Inversion of control: Megfordul a vezérlés – nem a komponens vezérel, őt vezérlik.

Control freak: Nem engedi, hogy őt vezéreljék; ragaszkodik az irányításhoz.

Első megközelítésben: volatile függőségek new-zása

Control freak: new

```
public class ProductService
{
    private readonly IProductRepository productRepository;
    0 references
    public ProductService()
    {
        this.productRepository = new ProductRepository();
    }
    0 references
    public void DoSomething()
    {
        this.productRepository.GetSomething();
    }
}
```

Control freak: static factory

```
public class ProductService
{
    private readonly IProductRepository productRepository;
    0 references
    public ProductService()
    {
        this.productRepository = ProductRepositoryFactory.CreateRep();
    }
    0 references
    public void DoSomething()
    {
        this.productRepository.GetSomething();
    }
}
```

Control freak: concrete factory

```
public class ProductService
{
    private readonly IProductRepository productRepository;
    0 references
    public ProductService()
    {
        this.productRepository = new ProductRepositoryFactory().CreateRep();
    }
    0 references
    public void DoSomething()
    {
        this.productRepository.GetSomething();
    }
}
```

Control freak: abstract factory

```
public class ProductService
{
    private readonly IProductRepositoryFactory productRepositoryFactory;
    0 references
    public ProductService()
    {
        this.productRepositoryFactory = new ProductRepositoryFactory();
    }
    0 references
    public void DoSomething()
    {
        this.productRepositoryFactory.CreateRep().GetSomething();
    }
}
```

Control freak: foreign default

```
public class ProductService
{
    private readonly IProductRepository productRepository;

    // Teszteléshez
    1 reference
    public ProductService(IProductRepository productRepository)
    {
        this.productRepository = productRepository;
    }

    // Élesben
    0 references
    public ProductService() : this(CreateDefaultRepository()) { }

    1 reference
    private static IProductRepository CreateDefaultRepository() ⇒ new ProductRepository();

    0 references
    public void DoSomething()
    {
        this.productRepository.GetSomething();
    }
}
```

Service locator

```
public class Locator
{
    private static readonly Dictionary<Type, Type> registrations;
    0 references
    public static void Register<TType, TInterface>() ⇒
        registrations.Add(typeof(TInterface), typeof(TType));
    1 reference
    public static T GetService<T>() ⇒
       (T)Activator.CreateInstance(registrations[typeof(T)]);
}
```

```
public class ProductService : IProductService
{
    private readonly IProductRepository productRepository;
    1 reference
    public ProductService(IProductRepository productRepository)
    {
        this.productRepository = Locator.GetService<IProductRepository>();
    }
}
```

Service locator: analízis

- A kód továbbra is lazán csatolt, annak minden előnyével: regisztrációkor azt regisztrálunk, amit csak szeretnénk
- Sőt, a Locator igazából egy DI containerként képzelhető el
- A probléma abból ered, ahogyan használjuk: nem a szolgáltatást hívóktól kényszerítjük ki a függőségek megadását, hanem mi magunk oldjuk fel őket

Service locator: analízis

- A Locator miatt lesz egy függőségünk, amit mindig magunkkal kell cipelni (DE: a Locator stable dependency-nek számít)
- A Locator megvalósítása miatt nagyon oda kell figyelni a szálbiztosságra; mi lesz ha párhuzamosan futnak a unit tesztek (mert futhatnak), de két külön teszt dublőr kell két párhuzamosan futó tesztnek egy adott komponens helyébe?
- Nem látszanak explicit módon a függőségek

Service locator: analízis

- Ki fogja a lekért komponens élettartamát majd felügyelni?
- Nem felügyelheti a szolgáltatás, hiszen akkor ez control freak
- Ha a service locator felügyeli, akkor ez inkább „service object locator”, ami tovább nehezíti az implementációt
- Ki kezeli a service locator-t?

Ambient context

Statikus hozzáférési pont biztosítása egyetlen szolgáltatáshoz, de annak valamilyen absztakcióján keresztül

- Félig Singleton
- Félig Service Locator

Ambient context

```
3 references
public abstract class TimeProvider
{
    2 references
    public abstract DateTime UtcNow { get; }
    1 reference
    public static TimeProvider Current { get; set; } = new DefaultTimeProvider();

}
```

```
1 reference
public class DefaultTimeProvider : TimeProvider
{
    2 references
    public override DateTime UtcNow => DateTime.UtcNow;
}
```

```
public class ProductService
{
    0 references
    public void DoSomething()
    {
        var x = TimeProvider.Current.UtcNow;
    }
}
```

Ambient context

☺ Nem jelennek meg mindenhol a függőségek

☹ Nehéz jól, robosztusan implementálni (pl.: szálbiztosság)

☹ Nagyon megnehezíti a tesztelést

☹ Nem jelennek meg mindenhol a függőségek (túlságosan implicit)

Constrained construction

- Ha sikerült mindent a composition rootba kimozdítani, akkor is lehetnek még problémák
- Az absztrakciók nem, vagy nem mindig tartalmaznak constructort – megkötni a létrehozást egy speciális constructorral olyan tervezési hiba, amely megnehezíti később komponensek hozzáadását
- Akkor jön elő, ha egy spéci függőségre tervezünk, aminek van valamilyen paramétere
- Klasszikus: in-memory db connection stringje?

Constructor overinjection

- A constructor injection miatt előfordulhat, hogy túl sok a constructor paraméter
  - Ez nem biztos, hogy baj, hiszen egyrészt úgyis containerből lesz feloldva…
  - … másrészt a függőségek ha összetartoznak, akkor ez még akár lehet is jó.
  - De persze a sok függőség utalhat arra, hogy nagy az osztály, túl sok felelőssége van.
  - Ha a függőségek viszont klasszikus CCC függőségek, akkor megint csak nincs baj.

Service aggregation / facade services

```
public abstract class TimeProvider
{
    1 reference
    public abstract DateTime UtcNow { get; }
}

0 references
public class DefaultTimeProvider : TimeProvider
{
    1 reference
    public override DateTime UtcNow ⇒ DateTime.UtcNow;
}

1 reference
public abstract class Logger
{
    1 reference
    public abstract void Log(string s);
}

0 references
public class ConsoleLogger : Logger
{
    1 reference
    public override void Log(string s)
    {
        Console.WriteLine(s);
    }
}
```

Service aggregation / facade services

```
1 reference
public class ProductService
{
    0 references
    public ProductService(TimeProvider timeProvider, Logger logger)
    {

    }
}
```

```
public abstract class TimeProvider
{
    1 reference
    public abstract DateTime UtcNow { get; }
}
```

```
0 references
public class DefaultTimeProvider : TimeProvider
{
    1 reference
    public override DateTime UtcNow => DateTime.UtcNow;
}
```

```
1 reference
public abstract class Logger
{
    1 reference
    public abstract void Log(string s);
}
```

```
0 references
public class ConsoleLogger : Logger
{
    1 reference
    public override void Log(string s)
    {
        Console.WriteLine(s);
    }
}
```

Absztrakció szivárgás

- Abstraction leaking/leaky abstraction: Az absztrakció célja, hogy elrejtse egy komponens elől egy másik felépítését/belső működését, hogy csak a használatra lehessen fókuszálni. Ha ez csak részben sikerül, akkor ott "szivárog" az absztrakció.

Absztrakció szivárgás: abstract factory

Hogyan kontrolláljuk egy függőség élettartamát egy másik komponensen belül?

- Abstract factory létrehozása, injektálása
  - Ez önmagában absztrakció-szivárgás, hiszen így az eredeti komponens helyett, csak azért, hogy az életciklust tudjuk menedzselni, a factory-ját kell injektálni
  - Ráadásul sokszor a függőség IDisposable, amit interfészre tenni önmagában is absztrakciószivárgás
- Lazy<T> injektálása
  - Még rosszabb, ennek ráadásul semmilyen interfésze nincs ☺
- Proxy tervezési minta használata
  - Ez lenne az ideális megoldás, egy virtuális proxy implementálásaával