Unit testing, DI

Albert István

ialbert@aut.bme.hu

Q.B. 221, 1662

Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Mindig tesztelj!

Ne adj át olyan kódot, amit még sosem indítottál el !

Mi egy unit teszt?

- Egy darab kód, ami egy másik darab kódot ellenőriz

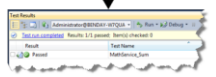
```
[TestMethod]
public void MathService_Sum()
{
    var facade = new MathService();

    int expected = 5;

    int actual = facade.Sum(2, 3);

    Assert.AreEqual<int>(
        expected, actual,
        "Got the wrong sum.");
}
```

```
public class MathService
{
    public int Sum(int value1, int value2)
    {
        return value1 + value2;
    }
}
```



Függőségek kezelése

- **Cél**: a tesztelt kód legyen minél kisebb, minél kevesebb külső függőséggel
- **Megoldás**: minden felhasznált külső modult (=komponenst/függőséget) helyettesítsünk be fix, a teszt által megadott implementációval!
- **Meglepetés**: Ez nagyon komoly architekturális követelményeket állít a rendszerrel és a felhasznált komponensekkel szemben!

IoC / DI konténer

- Objektum gráfok létrehozása automatikusan
  - > Nem new-val, a pontos típust kiírva
- Függőségek automatikus, rekurzív kielégítése
  - > Például constructor injection
- Az interfészhez tartozó konkrét típusokat deklaratívan adhatjuk meg a konténer példányon

```
var c = new Container();
c.Register<IClient, SomeClient>();
c.Register<IService, SomeService>();

// somewhere else
IClient client = c.Resolve<IClient>();
```

Mockolás és hibaforrások

- Direkt integrációs tesztelés – **nem tudni hol a hiba**
  - > Kód + adatbázis
  - > Én kódom + Te kódod
- Integrációs teszt mock objektumokkal – **pontosan tudni, hol a hiba**
  - > Kód + mock adatbázis
  - > Én kódom + Te mockkolt kódod
  - > Én mockkolt kódom + Te kódod
- A mockok egyértelműsítik hogy mit tesztelünk
  - > Mint a tudományos módszereknél: csak egy paramétert változtassuk, minden más legyen fix

nUnit keretrendszer

- JUnitból nőtte ki magát
- Nyílt forráskódú
- Élő közösség
- Jól integrálódik a Visual Studio-ba

Mindig tesztelj!

Ne adj át olyan kódot, amit még sosem indítottál el !

Edsger W. Dijkstra, in 1970

A tesztelés csak azt tudja megmutatni, ha a rendszerben hiba van, azt sose, ha nincs!

- Lehetetlen ellenőrizni a program össze lehetséges lefutását
- Fontos, hogy okosan teszteljünk!

Szoftver tesztelés

- A **szoftver tesztelés** során ellenőrizzük, hogy a program úgy működik-e ahogy elvárjuk
  - Verifikáció: a terméket jól készítjük-e el?
  - Validáció: a megfelelő terméket készítjük-e el?
- Tesztelés nélkül nincs visszajelzés a minőségről!
- A tesztelést érdemes a fejlesztés elején bevezetni
  - Ez meghatározza azt is, hogy hogyan fogod írni a kódot!
  - Minél korábban találod meg a hibát, annál olcsóbb/egyszerűbb kijavítani

A javítás költsége attól függően, hogy hol és mikor találjuk meg a hibát

Hol van a hiba	*Mikor ismerjük fel a hibát…*				
	Spec. elemzés	Tervezés	Megvalósítás	QA	Kiadás után
Követelmény	1x	3x	5-10x	10x	10-100x
Terv	-	1x	10x	15x	25-100x
Megvalósítás	-	-	1x	10x	10-25x

Módszertan - tesztelés

- Gyakori üzleti elvárás
  - Reagáljunk gyorsan (/olcsón/megbízhatóan) az üzleti igények változására!
- Módosítani kell a terveket, az implementációt
  - Csak ha ellenőrizhetőek a változtatások
- A projekt módszertan támaszt elvárásokat
  - Csak akkor tudunk agilisan fejleszteni, ha tudunk megfelelő unit teszteket írni
  - Ez további architekturális és egyéb kritériumokat ad
  - Egy módszertan alkalmazásának van **célja**, alkalmazhatóságának vannak **feltételei/következményei**!
- Hozhatunk bármilyen döntést: **tudatosan**!

Agilis módszertan elvárásai

- Gyorsan, olcsón lefuttatható
- Hosszú távon karbantartható
- Az eredmény jól felhasználható

- Javasolt megoldás: **unit teszt**

Mi egy unit teszt?

- Egy darab kód, ami egy másik darab kódot ellenőriz

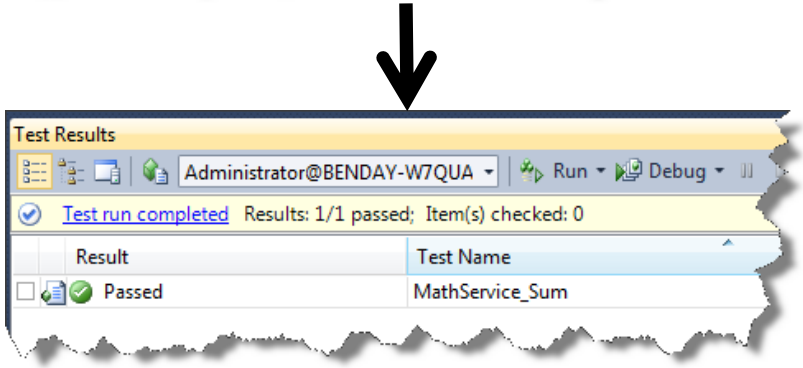
```
[TestMethod]
public void MathService_Sum()
{
    var facade = new MathService();

    int expected = 5;

    int actual = facade.Sum(2, 3);

    Assert.AreEqual<int>(
        expected, actual,
        "Got the wrong sum.");
}
```

```
public class MathService
{
    public int Sum(int value1, int value2)
    {
        return value1 + value2;
    }
}
```



A jó unit test

1. Megbízható
2. Hosszú távon futtatható, karbantartható
3. Jól olvasható
4. Egyszerű megírni
5. Gyorsan lefut
6. Automatizált és megismételhető
7. Bárki egyszerűen futtathatja
8. Izolált: jól felhasználható

1. Megbízható

- Nem indítjuk el debuggerben, hogy megnézzük, tényleg jó-e ☺
- Ha hibát jelez a teszt, akkor arra nem legyintünk ☺

- A teljes teszt halmazt tekintve: *elégséges* kódlefedettség…

A kód lefedettségről

- A bonyolult kódot fedjük le minél jobban
- A triviális kódhoz felesleges unit teszteket írni
- Kérdéses az értelme az olyan elvárásoknak, mint hogy legyen 100%-os lefedettségű a kódunk!

"Just because you have 100% code coverage doesn't mean that your code works. It only means that you've executed every line."

- Scott Hanselman (Hanselminutes interview with Scott Bellware)

2. Karbantartható teszt 1.

- Minimalizáljuk a tesztekben lévő logikát!
  - Ne használjunk véletlen számokat stb.
- Kerüljük a privát belső állapotok ellenőrzését
  - A belső működés egy refaktorálásnál megváltozhat
    - Tehetjük internallá vagy használhatunk accessort
  - Vagy tegyük publikussá: ezzel rögzítjük a szerződést amit megvalósít
- Setup metódus: ne inicializáljunk olyan objektumokat, amiket csak a tesztek egy része használ
  - A Setup meghívása rejtve van, a double-öket érdemes inkább a tesztekben létrehozni, segédmetódusokban

2. Karbantartható teszt 2.

- Izoláció: a tesztek sose épüljenek egymásra!
  - Ne hívják egymást
  - Ne építsenek egy közös állapotra
    - Mindig rollbackeljenek
- Túl specifikált tesztek elkerülése
  - Az állapot alapú teszt túl sokat feltételez a belső működésről
  - Mockot használ amikor a stub is elég volna

3. Olvashatóság

- A tesztekkel „üzenünk” a következő fejlesztő generációnak, akiknek a kódot karban kell tartani
- Úgy írjuk a teszt kódot, hogy az is könnyen olvassa, aki először látja!
- Elnevezések (teszt-szenárió-viselkedés)
- Jó assert üzenetek
  - Ne ismételjük amit a teszt keretrendszer elmond vagy amire a teszt neve utal
  - Assertbe sose tegyünk funkcióhívást!

4. Egyszerű megírni

- Tudatosan kialakított környezet
  - Architektúra: DI
  - Infrastruktúra: CI
  - Idő: becslés
- Segédosztályok, keretrendszerek, …
  - Automatizáltan futtatható, bárki által
  - Összegzés, hibák részletei
  - Osztálykönyvtár attribútumokkal, segédmetódusokkal, ellenőrző funkciókkal
- Rendelkezünk megfelelő teszt adatokkal

5. Gyorsan lefut

- Megfelelő technológia használata
- A lassú külső komponensek leválasztása, helyettesítése
- Kategorizálás: egység / integrációs / UI tesztek

8. Izoláció

- **Egyetlen funkciót** teszteljünk
- **Izoláltan:** minél kevesebb külső függőséggel
- Minden esetet külön
- Különálló futtatás - tipikus felépítés (xUnit)
  1. Setup - előkészítés
  2. Exercise - futtatás
  3. Verify - ellenőrzés
  4. Teardown - takarítás

Integrációs teszt – unit teszt

- Ha teszt nem felel meg az előző listának, akkor az jó eséllyel inkább integrációs teszt

- „*Integration testing* means testing two or more dependent software modules as a group.”

Mikor töröljünk ki egy tesztet?

- Szinte soha, a tesztek jelentik a biztonsági hálót
  - Duplikátumok (pl több fejlesztő készítette)
  - Változó követelmények
- Ha a teszt hibás, javítsuk ki
  - Ellenőrizzük, hogy jelez, ha hibás a kód
  - Ellenőrizzük, hogy lefut ha minden rendben van
- A tesztek javítása sok időt vehet igénybe
- Nagyon fontos az olvasható teszt!

Okok, amiért nem szoktunk unit-tesztelni…

- „A unit tesztek nehezek”
- „A unit tesztek azoknak kellenek, akik bénák”
- „A projekt nem elég nagy, hogy kelljen unit teszt”
- „Csak én dolgozom a projekten”
- „A Megrendelő nem fizeti ki a unit teszteket”
- „Szorít a határidő, nincs időnk a unit tesztekre”

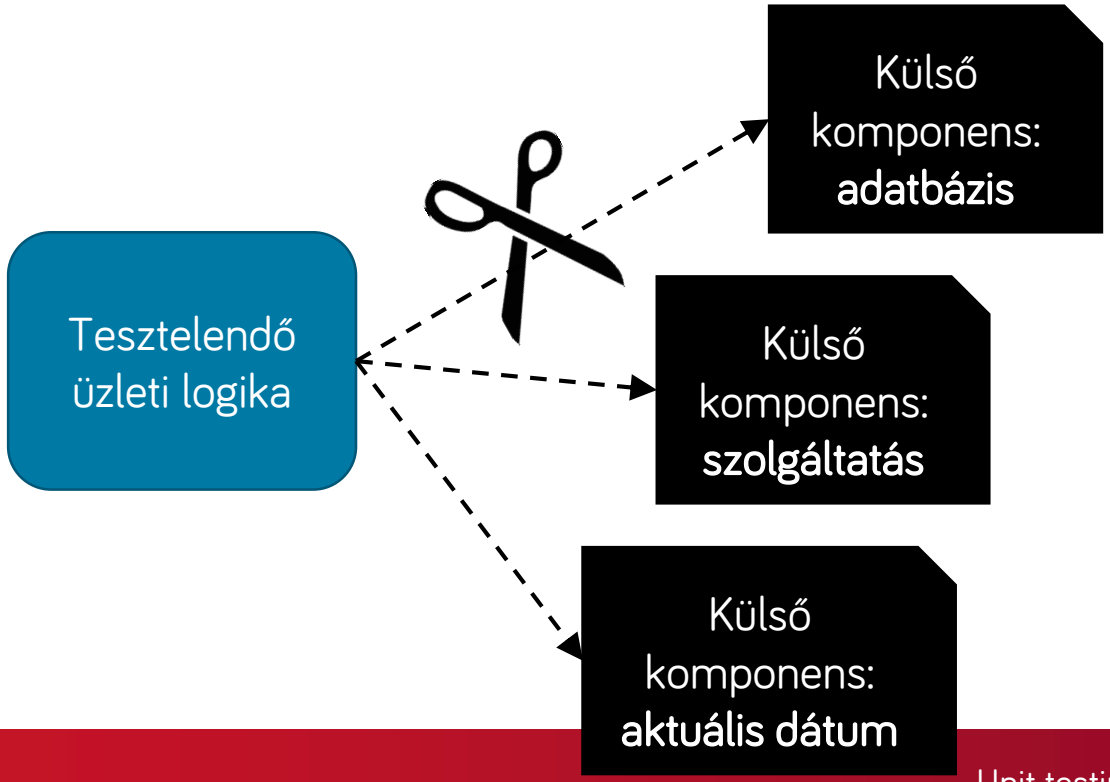
Függőségek kezelése

- **Cél**: a tesztelt kód legyen minél kisebb, minél kevesebb külső függőséggel
- **Megoldás**: minden felhasznált külső modult (=komponenst/függőséget) helyettesítsünk be fix, a teszt által megadott implementációval!

- **Meglepetés**: Ez nagyon komoly architekturális követelményeket állít a rendszerrel és a felhasznált komponensekkel szemben!

Interfész alapú programozás

- Elősegíti a modularitást, laza csatolást
- Lehetővé teszi helyettesek használatát
- **Egyszerűen jó dizájn!**



Inversion of Control (IoC)

- Általános fogalom
- Bármire utalhat, amikor a vezérlés „megfordul”
  - > esemény vezérelt programozás (windows)
- Nem a komponens vezényel, hanem Őt használják fel, hívják meg, irányítják
- Változtatják meg a viselkedését …

Függőségek, laza csatolás

- A kódban felhasznált komponensek konfigurációjának/létrehozásának és felhasználási helyének szétválasztása!
  - Például DBConnection objektum, connection string, …
- Megoldás: ezek az információk nincsenek belekódolva a tartalmazó osztályokba
  - Például így: conn = new SqlConnection(…);
- Az osztály kódjának függetlenítése a felhasznált komponensek *implementációjától*

Dependency Injection (DI)

- Szoftver tervezési minta: a függőségek átadásra kerülnek a függő objektumba
  - Az átadás lehet manuális vagy automatikus (injection)
- Laza csatolás
- A minta elemei
  - A függőség implementációja
  - A függő/kliens objektum
  - Az interfész
  - (injektor objektum)

„DI is a 25-dollar term for a 5-cent concept”

- Dependency injection means giving an object its instance variables. Really. That's it.

http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html

Első lépés: interfész

- Minden külső függőség funkciót írjuk le egy interfésszel
  - Explicit (deklaratív) a kapcsolat (szerződés) leírása
  - Cserélhető az implementáció
  - A lecserélés helye: „*seam*”
- A különböző megoldások az implementáció létrehozásának helyét, konfigurálását mondják meg
  - Például megkaphatja a konstruktorban…

Szolgáltatás interfész

```
/// <summary>
/// External dependency behind an interface.
/// </summary>
17 references | Albert István | 1 author, 1 change
public interface IBankInfoProvider
{
    /// <summary>
    /// Returns the name of the bank based on the first three
    /// eg. 117... -> OTP
    /// </summary>
    /// <param name="giroCode">First three digit of giro accou
    /// <returns>The name for the bank.</returns>
    9 references | Albert István | 1 author, 1 change
    string GetBankName( string giroCode );
}
```

Konstruktor alapú megoldás

```
/// <summary>
/// .ctor injection for testing.
/// </summary>
/// <param name="bankInfoProvider"></param>
4 references | 0/4 passing | Albert István | 1 author, 1 change
public AccountService( IBankInfoProvider bankInfoProvider )
{
    if( bankInfoProvider == null ) throw new ArgumentNullExcept
    this.bankInfoProvider = bankInfoProvider;
}
```

A konstruktor alapú megoldásról…

- Előnye, hogy jól láthatóak a kötelező függőségek
  - A property injection használható az opcionális függőségek megadására
- Sok paraméter lehet
  - Áthidalható, ha egyetlen paramétert kap, amiben minden szükséges függőséget átad a hívó – de akkor elveszítjük az deklaratív függőség leírást
- Új függőség új paramétert jelent, ami miatt az összes unit teszt módosítandó
  - Vagy újabb .ctor-t kell bevezetni ami hívja a többit
- IoC konténerrel érdemes használni…

Property alapú megoldás

```
/// <summary>
/// Used for property injection.
/// </summary>
1 reference | 0/1 passing | Albert István | 1 author, 1 change
public IBankInfoProvider BankInfoProvider
{
    get { return bankInfoProvider; }
    set { bankInfoProvider = value; }
}
```

A property alapú megoldásról…

- A függőségeket nem a .ctorban, hanem külön propertykben adja meg a hívó kód
- Nem explicit a függőségek leírása
  - Egy új függőség bevezetését az összes meglévő unit tesztben kézzel, hibajelzés nélkül kell végig vezetni
- Nem látni jól, hogy melyek a külső függőségek
  - Az osztálynak lehet egy csomó egyéb property-je, nem tudni, melyiket kell beállítani…
- Tipikusan opcionális függőségek esetén lehet jobb megoldás

További dimenziók

- Eddig **kívülről** állítottuk be **explicit** módon a függőségeket
  - A függőségek jelzése volt **deklaratív** vagy **„imperatív”**

1. *A függőségeket **belülről** is beállíthatja a osztály*
   - *Például ha az osztály mondja meg a létrehozás paramétereit stb.*
2. A függőségek létrehozása lehet **implicit** (automatikus)
   - Egy külső komponens (IoC container) felel azok létrehozásáért…

Factory metódus

```
/// <summary>
/// Could be used for production code.
/// </summary>
4 references | 0/4 passing | Albert István | 1 author, 1 change
public AccountService()
{
    // uses a local virtual factory method
    bankInfoProvider = this.CreateBankInfoProvider();
}

/// <summary>
/// Could be extended in unit tests.
/// </summary>
/// <returns>An IBankInfoProvider implementation.</returns>
3 references | Albert István | 1 author, 1 change
protected virtual IBankInfoProvider CreateBankInfoProvider()
{
    return new ExternalBankInfoProvider();
}
```

Factory minták

- **Factory method (simple factory)**: általában *egy* metódus implementáció, *egy* objektumot hoz létre a paraméterek alapján
  - Például: WebRequest . Create(URI)
- **Absztrakt factory minta**: Több factory leszármazott van, amelyek közül az egyik kiválasztott tipikusan több *kapcsolódó* objektumot hoz létre
  - Kiemelhető közös inicializációs logika az ős factoryban
  - A konkrét típusokhoz tartozó specifikumok a leszármazottakban
- **Cél**: a kliensről leválasztani a kompozíciós logikát

Factory minta

```
blic class BankInfoProviderFactory

    /// <summary>
    /// Default ctor to use production co
    /// </summary>
    3 references | 0/2 passing | Albert István | 1 author, 1 change
    public BankInfoProviderFactory()
    {
        BankInfoProviderType = typeof( ExternalBankInfoProvider );
    }

    /// <summary>
    /// Set by tests to use stub.
    /// </summary>
    4 references | 0/2 passing | Albert István | 1 author, 1 change
    public Type BankInfoProviderType { get; set; }


    /// <summary>
    /// Creates a BankInfoProvider instance.
    /// </summary>
    /// <returns>Returns a new instance of IBankInfoProvider instan
    2 references | Albert István | 1 author, 1 change
    public IBankInfoProvider Create()
    {
        return (IBankInfoProvider)Activator
            .CreateInstance( BankInfoProviderType );
```

```
/// <summary>
/// Uses hardwired factory or ServiceLocator to cre
/// </summary>
1 reference | 0/1 passing | Albert István | 1 author, 1 change
public AccountService( BankInfoProviderFactory fac
{
    this.bankInfoProvider = factory.Create();
}
```

esting, Dependency Injection

A tiszta factory minták hátrányai

- Tipikusan nem újrafelhasználhatók
  - A típusok és kiválasztó logika bele van kódolva a factory implementációba
  - A létrehozó kód nehézkesen általánosítható, módosítható, kiterjeszthető
- A legtöbb implementáció fordítás idejű típusinformációra épít
- A polimorfikus factory implementációk közös alaposztályra vagy interfészre alapoznak – de ezeket körülményes mindig implementálgatni
- Nehezebb tesztelni

Service Locator minta

- Egyetlen (singleton) objektum, ami tartalmazza az összes szükséges szolgáltatást
  - Inkább meglévő (létrehozott) objektum*okat* ad vissza – egy regisztrációs adatbázisból (registry)
  - A szolgáltatások létrehozása lehet statikus (belekódolt), dinamikus (futásiőben lehet konfigurálni mire van szükség) stb.
  - De akár használhat Dependency Injection-t is

Service Locator

```
[Test]
0 references | Albert István | 1 author, 1 change
public void GetBankName_OTP_withservicelocator()
{
    // ARRANGE
    ServiceLocator.Instance.Register<IBankInfoProvider>(
        () => new StubBankInfoProvider() );
    AccountService srv = new AccountService( false );

    // ACT
    var name = srv.GetBankName(
        new Account { AccountNumber = AN_long, ID = -1 } );

    // ASSERT
    Assert.That( name, Is.EqualTo( "OTP" ) );
}
```

```
/// <summary>
/// Uses hardwired factory o
/// </summary>
2 references | 0/2 passing | Albert István |
public AccountService( bool useFactoryNotServiceLocator )
{
    // we don't care if it's a stub or not
    if( useFactoryNotServiceLocator )
    {
        var factory = new BankInfoProviderFactory();
        this.bankInfoProvider = factory.Create();
    }
    else
        this.bankInfoProvider = ServiceLocator
            .Instance.Resolve<IBankInfoProvider>();
}
```

Service Locator mint anti-pattern (?)

- A függőségek nem jelennek meg explicit módon
  - A kódban imperatív módon van benne, hogy a Locatort milyen paraméterekkel hívja meg
- Új függőség bevezetése
  - Egyszerű a felhasználási helyen
  - Lefordul a kód
  - Az összes unit test hibás (lehet) ami futás időben derül ki

A Service Locator előnyei

- Új függőség bevezetése egyszerűbb, kevesebb módosítással jár a tesztelendő kódban
- Kisebb kód, mintha rengeteg paramétert kéne a konstruktorokban átadogatni
  - Ráadásul ilyenkor a felső szintű osztályok olyan paramétereket kapnak, amikhez semmi közük
  - A hívási hierarchia mélyén lévő osztályok könnyen tudják elérni a szükséges függőségeket
  - Az implicit függőségi hibák kiszűrésére megoldás a unit testing
  - Példa: egy DAL rétegben lévő SQL logger

DI vs ServiceLocator

- Service Locator esetén a függőség elkérése explicit (imperatív)
  - Pl könnyű rá breakpointot tenni…
  - Nehezebb látni a függőségeket
  - Extra függőség van a ServiceLocatoron – nehezebb újrafelhasználhatóság
- A dependency injection deklaratív
  - Nehezebb debuggolni, átlátni a létrehozás folyamatát
  - Könnyebben láthatóak a függőségek (pl ctor)

A két minta egyfajta összevetése

- Az egyik példányosítva csak egy adott típusú (vagy abból leszármazott) objektumot ad vissza
- A másik példánya viszont több típusút tud előállítani

Abstract Factory	Service Locator

`public interface IFactory<T>`
`{`
`    T Create(object context);`
`}`
	`public interface IServiceLocator`
	`{`
	`    T Create<T>(object context);`
	`}`

A két minta másfajta összevetése

- A factory által visszaadott példány mindenképpen új
  - Így annak a hívó a tulajdonosa
- A ServiceLocator viszont nem biztos, hogy új példányt hoz létre
  - Ilyen értelemben IoC jellegű megoldás
  - A kapott példányt lehet, hogy mások is használják (singleton stb)

A bemutatott megoldások áttekintése

1. A külső komponenst interfész választja le, de a provider létrehozása bedrótozott
2. A service (opcionális) konstruktor paraméterben kapja meg a providert
3. A service propertyben kapja meg a providert
4. Local factory method
5. A service factory osztályt használ a példányosításhoz, a factory testreszabható
6. A service Locatort használ a példány létrehozásához

További dimenziók

- Eddig **kívülről** állítottuk be **explicit** módon a függőségeket
  - A függőségek jelzése volt **deklaratív** vagy **„imperatív”**

1. A függőségeket **belülről** is beállíthatja az osztály
   - Például ha az osztály mondja meg a létrehozás paramétereit stb.
2. *A függőségek létrehozása lehet **implicit** (automatikus)*
   - *Egy külső komponens (IoC/DI container) felel azok létrehozásáért…*

IoC / DI konténer

- Objektum gráfok létrehozása automatikusan
  - Nem new-val, a pontos típust kiírva
- Függőségek automatikus, rekurzív kielégítése
  - Például constructor injection
- Az interfészhez tartozó konkrét típusokat deklaratívan adhatjuk meg a konténer példányon

```
var c = new Container();
c.Register<IClient, SomeClient>();
c.Register<IService, SomeService>();

// somewhere else
IClient client = c.Resolve<IClient>();
```

DryIoC

```
using( var container = new Container() )
{
    // ARRANGE
    container.Register<IBankInfoProvider, StubBankInfoProvider>();
    // selecting the .ctor is necessary only when there are multiple .ctors.
    container.Register<AccountService, AccountService>(
        withConstructor: type => type.GetConstructor(
            new[] { typeof( IBankInfoProvider ) } ) );


    AccountService srv = container.Resolve<AccountService>();

    // ACT
    var name = srv.GetBankName( new Account { AccountNumber = AN_long, ID = -1 } );

    // ASSERT
    Assert.That( name, Is.EqualTo( "OTP" ) );
}
```

DI fogalmak

- **Resolution**: a konténeren hívott Resolve metódus, ami visszaadja a létrehozott objektumgráfot
- **Resoltuion root**: a létrehozott gráf gyökere
- **Injection**: a függő objektumok létrehozása és átadása
  - Konstruktorban: javasolt
  - Propertyben: kerülendő, de támogatott

Életciklus támogatás

- A DI konténer szabályozza a létrehozott objektumok életciklusát
  - „normál” módon viselkedő objektumok
  - Singleton minta
  - Szálanként egyetlen objektum létrehozása
  - UoW támogatása – például HTTP kérésenként egy objektum létrehozása

Singleton minta

- A regisztrációnál adható meg az életciklus modell

```
c.Register<IClient, SomeClient>();
c.Register<IService, SomeService>(Reuse.Singleton);

// consuming part still the same, win!
IClient client = c.Resolve<IClient>();

// let's check
var anotherClient = c.Resolve<IClient>();
Assert.AreSame(anotherClient, client);
```

Egyéb újrafelhasználási modellek

- **Transient**: nincs újrafelhasználás
- **Singleton**: a konténer életciklusával együtt él, dispose-olódik a konténer megszűnésekor
- **InResolutionScope**: egy gráfon belül ugyanaz a példány kerül használatra egy típusból
- Saját reuse támogatása: **IReuse** interfészen keresztül

Dryloc - scope

- Külön konténer objektum, ami követi a benne InCurrentScope módon létrehozott objektumokat
  - Egyébként hivatkozik az eredeti konténerre: regisztrációk stb.
- A benne lévő objektumok megszűnnek amikor ez a konténer is

```
container.Register<Car>(Reuse.InCurrentScope);
// container.Resolve<Car>(); // throws ContainerException here because t
using (var scopedContainer = container.OpenScope())
{
    var car = scopedContainer.Resolve<Car>();
    car.DriveMeToMexico();
}
// Disposable car will be disposed here together with opened scope in sc
```

Késleltetett példányosítás

- Factory metódust kérünk a konténertől
- Lazy<T> vagy Func<T> létrehozása esetén a példány nem jön létre, csak amikor azt elkérjük
- Így áttehető a létrehozás pillanata a megfelelő scope-ba

```
container.Register<Car>(Reuse.InCurrentScope);
var carFactory = container.Resolve<Func<Car>>(); // Does not throw.
using (var scopedContainer = container.OpenScope())
{
    var car = carFactory();
    car.DriveMeToMexico();
}
```

Scope-ok

- A scope-ok automatikusan egymásba ágyazódnak, a konténer követi őket
- Hány aktív scope van? Választható:
  - Szálanként – ThreadScopeContext (default)
  - Aszinkron – ExecutionFlowContext
  - Http kérésenként: HttpContextScopeContext
  - Saját – IScopeContext interfész

Egyebek

- Factory delegate-ek támogatása regisztrálásnál
  - Statikus típusokhoz
  - Dinamikus/futásidőben létrehozott típusokhoz
- MEF attribútum szimuláció

Tipikus hibák

- Túl sok konstruktor paraméter
  - Sérti a „Single Responsibility” elvet
  - „God object” anti-minta gyanús
  - Ha mégis kell: készíthetünk wrappert

DI az ASP.NET Core-ban

- Életciklus vezérlés
  - Transient: mindig új objektum
  - Scoped: kérésenként egy új objektum
  - Singleton: egyetlen példány
- A beépített DI konténer is lecserélhető

```
// Register application services.
services.AddScoped<ICharacterRepository, CharacterRepository>();
services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
services.AddTransient<OperationService, OperationService>();
```

A beépített DI konténer lecserélése

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    // Add other framework services

    // Add Autofac
    var containerBuilder = new ContainerBuilder();
    containerBuilder.RegisterModule<DefaultModule>();
    containerBuilder.Populate(services);
    var container = containerBuilder.Build();
    return new AutofacServiceProvider(container);
}
```

Mockolás és hibaforrások

- Direkt integrációs tesztelés – **nem tudni hol a hiba**
  - Kód + adatbázis
  - Én kódom + Te kódod
- Integrációs teszt mock objektumokkal – **pontosan tudni, hol a hiba**
  - Kód + mock adatbázis
  - Én kódom + Te mockkolt kódod
  - Én mockkolt kódom + Te kódod
- A mockok egyértelműsítik hogy mit tesztelünk
  - Mint a tudományos módszereknél: csak egy paramétert változtassuk, minden más legyen fix

Állapot tesztelés: fekete doboz

- A külső függőségeket behelyettesítettük stubokkal
  - Egyszerű, logika nélküli osztályok fix válaszokkal
- **A teszt sikeressége a tesztelt objektum állapotán múlik**
  - Gyakran külön metódusok (accessor osztályok) kellenek a belső állapot lekérdezéséhez
- Néha nem elég: például ha az eredmény egy hívás egy másik objektumba, nem pedig a belső állapot megváltozása

Vislkedés tesztelés: fehér doboz

- A felhasznált komponenseket megfelelő sorrendben és paraméterekkel hívta-e meg
  - A hivatkozott komponensek mockolva vannak
- Tipikusan mockokkal dolgozik
- Mock (test spy): olyan objektum, ami a tesztelt kód (CUT) interakciója alapján dönti el, hogy a teszt sikeres-e
- **A teszt sikerességét a mock osztály dönti el!**

A „test double”-ök terminológiája

Elnevezés	Magyarázat
Dummy object	Sose használt, paraméterként átadogatott objektum
Fake object	Működő, de leegyszerűsített implementáció, például memória adatbázis valódi helyett stb.
Stub	Beégetett visszatérési értékeket ad a teszt során. Inkább állapot verifikációra szolgál. Néha extra metódusokkal rendelkezik az állapot lekérdezéséhez.
Mock object	A valódi osztály működését imitálja. Lényegi különbség van a setup és verifikációs fázisban, egyébként ugyanúgy többnyire fix értékeket ad vissza. Nem állapot verifikációval dolgozik.
Mole	Tetszőleges metódus hívás elirányítható. Futtató környezet (CLR) szintű megoldás.

Mocking keretrendszerekről általában

- Interfész / alaposztály helyettesítés
  - Tipikusan Castle DynamicProxy-t használnak a futás idejű osztály létrehozásra
- Viselkedés beállítás
  - Klasszikus: Setup, …
  - Record / replay minta (moq nem támogatja)
- Futtatás
- Ellenőrézs

moq keretrendszer

- Jelenleg az egyik legkényelmesebb, legtöbbet használt és ajánlott, aktívan fejlesztett keretrendszer (2017)
- Erősen típusos: linq kifejezések stringek helyett
- Nincs record/replay minta
- Egyszerű, gyorsan tanulható
- Korszerű nyelvi elemek használata

Moq használata

```
[Test]
0 references | Albert István | 1 author, 1 change
public void GetBankName_OTP_withmoq_2()
{
    // ARRANGE
    // ennek alapján már a strict is elég, hiszen megadjuk,
    // hogy pontosan milyen paraméterekkel hívják meg a metódust
    var provider = new Mock<IBankInfoProvider>( MockBehavior.Strict );
    // 117 -> OTP
    provider.Setup( p => p.GetBankName("117") ).Returns( "OTP" );
    var srv = new AccountService( provider.Object );

    // ACT
    var name = srv.GetBankName( new Account { AccountNumber = AN_long, ID = -1 } );

    // ASSERT

    // ellenőrzi, hogy a metódus ezzel a paraméterrel egyszer meg lett hívva
    provider.Verify( p => p.GetBankName( "117" ), Times.Once );

    // az alábbit nem nagyon van értelme ellenőrizni, hiszen ezt mi adjuk vissza a m
    // legfeljebb akkor van értelme, ha azt szeretnénk tudni, hogy tényleg visszajön
    Assert.That( name, Is.EqualTo( "OTP" ) );
}
```

Mock<T> központi osztály

- A T típus példányát fogjuk mockolni
- Setup metódus: melyik metódus milyen bemenetre milyen választ adjon
- A T típusú mockolt példány az Object propertyn keresztül érhető el

```
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
```

```
// throwing when invoked
mock.Setup(foo => foo.DoSomething("reset")).Throws<InvalidOperationException>();
mock.Setup(foo => foo.DoSomething("")).Throws(new ArgumentException("command");


// lazy evaluating return value
mock.Setup(foo => foo.GetCount()).Returns(() => count);
```

Hívások ellenőrzése

```
mock.Verify(foo => foo.Execute("ping"));

// Verify with custom error message for failure
mock.Verify(foo => foo.Execute("ping"), "When doing operation X, the service shou

// Method should never be called
mock.Verify(foo => foo.Execute("ping"), Times.Never());

// Called at least once
mock.Verify(foo => foo.Execute("ping"), Times.AtLeastOnce());

mock.VerifyGet(foo => foo.Name);

// Verify setter invocation, regardless of value.
mock.VerifySet(foo => foo.Name);

// Verify setter called with specific value
mock.VerifySet(foo => foo.Name ="foo");

// Verify setter with an argument matcher
mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5, Range.Inclusive));
```

Input intervallumok kezelése

- Fluent syntax

```
// any value
mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);


// matching Func<int>, lazy evaluated
mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);


// matching ranges
mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns(true);
```

Propertyk, stubként

- Klasszikus stubként működik
  - > Vagy: mock.SetupAllProperties

```
// start "tracking" sets/gets to this property
mock.SetupProperty(f => f.Name);

// alternatively, provide a default value for the stubbed property
mock.SetupProperty(f => f.Name, "foo");


// Now you can do:

IFoo foo = mock.Object;
// Initial value was stored
Assert.Equal("foo", foo.Name);

// New value set which changes the initial value
foo.Name = "bar";
Assert.Equal("bar", foo.Name);
```

Propertyk mockolása

- Visszatérési érték megadása
- Setter hívás ellenőrzése

```
mock.Setup(foo => foo.Name).Returns("bar");


// auto-mocking hierarchies (a.k.a. recursive mocks)
mock.Setup(foo => foo.Bar.Baz.Name).Returns("baz");

// expects an invocation to set the value to "foo"
mock.SetupSet(foo => foo.Name = "foo");

// or verify the setter directly
mock.VerifySet(foo => foo.Name = "foo");
```

A mock viselkedése

- **Strict** (true mock) mód: mindig kivételt dob, ha úgy hívják meg, ahogy nem lett előre specifikálva
- **Loose**: nem dob kivételt, mindig alapértelmezett értékeket ad vissza (0, null, üres tömb stb)
- Megadható, hogy az alaposztályt hívja meg, ha nincs a viselkedés felüldefiniálva (CallBase)

- A viselkedés a konstruktorban adható meg

Rekurzív mockok

- Ha kell, a mock által visszaadott objektumok automatikusan mockok lesznek

```
var mock = new Mock<IFoo> { DefaultValue = DefaultValue.Mock };
// default is DefaultValue.Empty

// this property access would return a new mock of IBar as it's "mock-able"
IBar value = mock.Object.Bar;

// the returned mock is reused, so further accesses to the property return
// the same mock instance. this allows us to also use this instance to
// set further expectations on it if we want
var barMock = Mock.Get(value);
barMock.Setup(b => b.Submit()).Returns(true);
```

Egyéb támogatás

- Mock factory – közös defaultok stb
- Protected tagok
- Események támogatása
  - Például hívás hatására
- Callbackek támogatása
- Több interfész implementálása

nUnit keretrendszer

- JUnitból nőtte ki magát
- Nyílt forráskódú
- Élő közösség
- Jól integrálódik a Visual Studio-ba

NUnit teszt leírók

- Alap attribútumok
  - TestFixture
  - Test
  - Ignore
  - Category

```
1  namespace TestingIntroduction
2  {
3      using NUnit.Framework;
4
5      [TestFixture]
6      public class BasicAttributeTests
7      {
8          [Test]
9          public void PassingTest()...
13
14         [Test, Ignore("I am ignored.")]
15         public void IgnoredTest()...
19
20         [Test]
21         public void FailingTest()...
25
26         [Test, Category("MyCategory")]
27         public void CategorizedTest()...
31     }
32 }
```

SetUp & TearDown

- Tesztenként
- Test fixture-önként

```
namespace NUnit.Tests
{
  using System;
  using NUnit.Framework;

  [TestFixture]
  public class SuccessTests
  {
    [SetUp] public void Init()
    { /* ... */ }

    [TearDown] public void Cleanup()
    { /* ... */ }

    [Test] public void Add()
    { /* ... */ }
  }
}
```

Paraméterezett tesztek

- TestCase
  - Result
  - TestName
- Theory
  - DataPoint
  - Általános állítások a rendszerrel kapcsolatban

```
[TestCase(12,3, Result=4)]
[TestCase(12,2, Result=6)]
[TestCase(12,4, Result=3)]
public int DivideTest(int n, int d)
{
  return( n / d );
}
```

```
public class SqrtTests
{
    [Datapoints]
    public double[] values = new double[] { 0.0, 1.0, -1.0, 42.0 };

    [Theory]
    public void SquareRootDefinition(double num)
    {
        Assume.That(num >= 0.0);

        double sqrt = Math.Sqrt(num);

        Assert.That(sqrt >= 0.0);
        Assert.That(sqrt * sqrt, Is.EqualTo(num).Within(0.000001));
    }
}
```

További attribútumok

- Localization
  - Culture
  - SetCulture
- Labeling
  - Description
  - Suite
  - Property
- Parametric
  - Attributes
    - Values
    - ValueSource
    - Range
  - Combinatorial
  - Pairwise
  - TestCaseSource
- Timing (async)
  - MaxTime
  - Timeout
- Threading
  - RequiresMTA
  - RequiresSTA
  - RequiresThread
- Others
  - Explicit
  - RequiredAddIn
  - Suite
  - ExpectedException
  - Platform

NUnit ellenőrzés szintaktika

- Classic Assertions
- Fluent Constraints
- AssertionHelper
  - > Expect
- Constraints
  - > Is, Has, Throws

```
[TestFixture]
public class BasicAssertionTests : AssertionHelper
{
    [Test]
    public void ClassicAssertion_Less()
    {
        var a = 2;
        var b = 1;

        Assert.Less(b, a);
    }

    [Test]
    public void FluentConstraints_LessThan()
    {
        var a = 2;
        var b = 1;

        Assert.That(b, Is.LessThan(a));
    }

    [Test]
    public void ExpectFluentConstraint_LessThan()
    {
        var a = 2;
        var b = 1;

        Expect(b, Is.LessThan(a));
    }
}
```

Összefoglalás

- A módszertan és architektúra szorosan összefüggnek
  - A verifikáció és validáció sok szinten megjelenhet a projektben
- A mai módszertanok még mindig nagyon törékenyek
  - Szerződés -> elvárások a módszertannal szemben
  - Módszertan -> elvárások az architektúrával szemben
  - Archtektúra -> elvárások a technológiával szemben
  - A technológiák csak részben felelnek meg
  - Párhuzamosan megjelennek az eszközök is…

Kérdések?

Albert István
ialbert@aut.bme.hu