Laborator 2.1

Subiect: Test driven development si Unit testing - prezentare si demo in Visual Studio 2022

1. Prezentare Test Driven Development

Demo: metoda de adunare a doua numere; scrierea unei clase Vector de intregi (tablou de elemente) ce suporta operatii de baza (instantiere, adunare, produs scalar): se dezvolta in stil TDD un proiect de tip Class Library.

Sursa: Dan Bunea blog

Nota: in cele ce urmeaza se face referinta la NUnit, un framework de testare larg raspandit ce nu este inclus in Visual Studio. Abstractie facand de elementele de sintaxa, discutia este valabila si pentru TDD in Visual Studio.

<citat>

Test Drive Development este una dintre cele mai cunoscute practici agile. Din pacate, nivelul intelegerii corecte asupra ce inseamna TDD este scazut si se pare ca sunt o serie de neintelegeri si folosiri gresite a tehnicii.

Cele mai mari confuzii cand vine vorba de TDD

Cei care nu stiu prea multe despre practicile agile sau sunt incepatori au tendinta sa creada ca TDD inseamna fie sa ai teste automate pentru codul scris fie ca TDD este o metoda de testare, si nu de dezvoltare de cod. Sa le luam pe rand:

1. Test Driven Development este pentru testeri

Confuzia poate veni si din nume: Test Driven Development care pentru unii programatori inseamna a testa codul prin teste automate si care ar trebui sa fie treaba tester-ilor (QA). Cuvantul test din nume pare sa duca la confuzii in cazul multor dezvoltatori, care automat spun ca TDD nu este pentru ei ci pentru testeri. Titlul totusi ar trebui sa fie destul de clar: **teste automate care influenteaza (drive) dezvoltarea codului.**

2. Eu folosesc TDD pentru ca am teste automate pentru cod

Incepatorii in metodologiile (nu ca as fi eu ultra avansat) au tendinta sa creada ca a avea teste automate care acopera partial sau integral codul inseamna ca ei folosesc TDD. Din pacate TDD este o tehnica care spune ca ***testele se scriu in paralel cu codul***, **de aceea daca aceste teste sunt scrise dupa, nu este TDD**. Totusi, daca aceste teste automate exista si chiar daca sunt scrise dupa ce codul a fost dezvoltat, e un lucru bun, si poate influenta dezvoltarea macro a proiectului pe viitor, ceea ce inseamna intr-un fel tot TDD dar la un nivel mai inalt: la nivel de proiect.

Teoria

TDD este o metoda de dezvoltare de cod, tintind programatorii si nu testerii, care foloseste teste automate care sunt scrise inaintea codului, pentru a influenta modul in care codul este scris. La sfarsit codul scris "trece" testele, ceea ce inseamna ca el a fost influentat de teste, fiind scris pentru ca testele sa poata "merge". Pentru ca tehnica este amplu descrisa in literatura de specialitate,(in romana mai putin) o sa incerc sa nu dezvolt prea tare subiectul. TDD se mai numeste si "red-green-factor", "test a little, code a little" sau "test, fail, code, pass" si consta intr-o serie de pasi care trebuie urmati pentru a obtine codul pe care-l dorim. Deci este o tehnica de dezvoltare a codului, una diferita destul de mult de cele "clasice". Pasi:

Test: Scrie un test mic (unul nu mai multe)

1. **Compile: Fa codul sa compileze si codul scris sa faca testul sa "fail"**
2. **Fail: ruland testul, acesta da "failed"**
3. **Code: scrie codul , doar suficient incat sa treaca testul, *nu mai mult***
4. **Pass: ruleaza testul, ar trebui sa dea "succeeded". Daca nu, revino la pasul 4.**
5. **Refactor: fa refactoring la codul scris pana acuma. (si cel de test si codul propriu-zis)**
6. **Pass again. Daca "pica" revino la 4**
7. **Repeat: Mergi la pasul 1 , adauga un nou test, fa-l sa treaca si fa asta de suficiente ori incat sa obtii codul dorit.**

Acesta este o lista simplificate de pasi. De fapt primul pas, este gandirea la ce se cere iar al doilea dezvoltarea primului test. TDD este un proces iterativ care presupune scrierea unui test, si a colului pentru el si repetarea fenomennului de suficiente ori pana cand toate testele care ar trebui sa le treaca codul nostru, sunt trecute.

Exemplu practic
Sa zicem ca avem un sistem de procesare a comenzilor, cerut de un client si eu sunt rugat sa scriu partea unde clientilor li se da un discount, daca comanda are o valoare mai mare decat o valoare definita. In XP (Extreme Programming) acesta ar fi un task la un "user story".
In acest moment, eu nu am absolut nici o linie de cod scrisa. Prima data ma gandesc ca ar trebui sa am o clasa Order, care ar trebui sa aiba mai multe OrderLine. Ok, sa scriu un test care face totalul unui Order, care nu are nici un OrderLine:

```
[TestFixture]
public class OrderTests
{
   [Test]
   public void EmptyOrder()
   {
      Order order = new Order();
      Assert.AreEqual(0,order.Total);
   }
}
```

Dar eu nu am clasa Order. La pasul 2 ar trebui sa fac codul sa compileze in asa fel incat testul sa nu treaca.Ok, la treaba:

```
public class Order
{
   public decimal Total
   {
      get{throw new NotImplementedException();}
   }
}
```

Compilez, si testul pica: (vezi: http://www.geocities.com/danbunea/articles/dotnet/tdd/image001.jpg)
Se pare ca pasii 1-3 au fost simpli, sa ma mut la pasul 4, scrisul codului care sa treaca codul. S-ar putea sa va surprinda:

```
public class Order
{
   public decimal Total
   {
      get{return 0;}
   }
}
```

Intentionat am facut codul sa returneze 0, ceea ce face ca codul meu sa nu fie bun, dar face testul sa treaca ceea ce este suficient acum. Testul imi spune ce cod sa scriu, nu scriu nimic "de la mine"

Acum pasul 5, sa vedem: http://www.geocities.com/danbunea/articles/dotnet/tdd/image002.jpg

Wow, se pare ca am implementat codul care sa faca primul test sa treaca, sa vedem acum daca este necesar vreun refactoring. Pana acum nu vad nevoia nici unui refactoring, asa ca revenim la pasul 1 incepand cel de-al doilea ciclu. Un test la total pentru o comanda cu cateva randuri.

```
[TestFixture]
public class OrderTests
{
   [Test]
   public void EmptyOrder()
```

```
    {
        Order order = new Order();
        Assert.AreEqual(0,order.Total);
    }

    [Test]
    public void TwoOrderLinesTotal()
    {
        Order order = new Order();

        OrderLine ol1 = new OrderLine();
        ol1.Product = "Laptop";
        ol1.Quantity = 1.0;
        ol1.Price = 1000.0;

        OrderLine ol2 = new OrderLine();
        ol2.Product = "Monitor";
        ol2.Quantity = 2.0;
        ol2.Price = 200.0;

        order.AddOrderLine(ol1);
        order.AddOrderLine(ol2);

        Assert.AreEqual(1400.0, order.Total);
    }
}
```

Sa-l facem sa compileze si testul sa pice, adaugand OrderLine si metoda AddOrderLine:

```
public class OrderLine
{
    public string Product
    {
        get {throw new NotImplementedException();}
        set { }
    }
    public decimal Quantity
    {
        get { throw new NotImplementedException(); }
        set { }
    }
    public decimal Price
    {
        get { throw new NotImplementedException(); }
        set { }
    }
}
```

Si:

```
public class Order
{
    public decimal Total
    {
        get{return 0;}
    }
```

```
        public void AddOrderLine(OrderLine ol)
        {
        }
    }
```

Sa vedem acum: http://www.geocities.com/danbunea/articles/dotnet/tdd/image003.jpg

E timpul sa facem testul sa treaca. Adaugam un pic de cod, vedem daca trece. Daca nu , mai adaugam putin pana cand putem trece la pasul urmator, toate testele trecand cu succes. Codul devine:

```
    public class Order
    {
        private IList orderLines = new ArrayList();

        public decimal Total
        {
            get
            {
                decimal sum = (decimal)0;
                foreach(OrderLine ol in this.orderLines)
                {
                    sum += ol.Total;
                }
                return sum;
            }
        }

        public void AddOrderLine(OrderLine ol)
        {
            this.orderLines.Add(ol);
        }
    }
```

si OrderLine:

```
    public class OrderLine
    {
        private decimal quantity;
        private decimal price;
        private string product;

        public string Product
        {
            get { return product; }
            set { product = value; }
        }
        public decimal Quantity
        {
            get { return quantity; }
            set { quantity = value; }
        }
        public decimal Price
        {
            get { return price; }
            set { price = value; }
```

```
    }

    public decimal Total
    {
        get
        {
            return this.quantity * this.price;
        }
    }
}
```

http://www.geocities.com/danbunea/articles/dotnet/tdd/image004.jpg

Wow, am facut si al doilea test, in 5 min, sa vedem ce refactorings ar fi necesare. Codul pare ok, dar se pare ca avem cod duplicat (initializarea Order). O sa mutam acest cod in SetUp, care oricum e rulat inainte de fiecare metoda de test (test case).

```
[TestFixture]
public class OrderTests
{
    Order order = null;

    [SetUp]
    public void SetUp()
    {
        order = new Order();
    }

    [Test]
    public void EmptyOrder()
    {
        Assert.AreEqual(0,order.Total);
    }

    [Test]
    public void TwoOrderLinesTotal()
    {
        OrderLine ol1 = CreateOrderLine("Laptop",1,1000);
        OrderLine ol2 = CreateOrderLine("Monitor", 2, 200);

        order.AddOrderLine(ol1);
        order.AddOrderLine(ol2);

        Assert.AreEqual(1400.0, order.Total);
    }

    private OrderLine CreateOrderLine(string product, decimal quantity, decimal price)
    {
        OrderLine ol1 = new OrderLine();
        ol1.Product = product;
        ol1.Quantity = quantity;
        ol1.Price = price;
        return ol1;
    }
}
```

Sa vedem daca mai ruleaza (testele daca mai trec):
http://www.geocities.com/danbunea/articles/dotnet/tdd/image005.jpg

Se pare ca da asa ca tocmai am terminat al doilea ciclu, si avem un cod aproape bun testat destul de bine. Sa vedem ce putem face la partea cu discountul, printr-un nou ciclu, asa ca adaugam un nou test case:

```
[TestFixture]
public class OrderTests
{
   Order order = null;

   [SetUp]
   public void SetUp()
   {
      order = new Order();
   }

   [Test]
   public void EmptyOrder()
   {
      Assert.AreEqual(0,order.Total);
   }

   [Test]
   public void TwoOrderLinesTotal()
   {
      OrderLine ol1 = CreateOrderLine("Laptop",1,1000);
      OrderLine ol2 = CreateOrderLine("Monitor", 2, 200);

      order.AddOrderLine(ol1);
      order.AddOrderLine(ol2);

      Assert.AreEqual(1400.0, order.Total);
   }

   [Test]
   public void Discount10PercentOver2000()
   {
      IRule discountRule = new DiscountRule(10, 2000);

      OrderLine ol1 = CreateOrderLine("Laptop", 1, 1000);
      OrderLine ol2 = CreateOrderLine("Monitor", 2, 200);
      OrderLine ol3 = CreateOrderLine("MimiMac", 2, 500);

      order.AddOrderLine(ol1);
      order.AddOrderLine(ol2);

      order.ApplyBusinessRule(discountRule);

      Assert.AreEqual(2400.0, order.Total);
      Assert.AreEqual(2160.0, order.TotalAfterDiscount);

   }

   private OrderLine CreateOrderLine(string product, decimal quantity, decimal price)
```

```
    {
        OrderLine ol1 = new OrderLine();
        ol1.Product = product;
        ol1.Quantity = quantity;
        ol1.Price = price;
        return ol1;
    }
}
```

Acum dupa ce facem codul sa compileze si testul sa crape. Merge, acum facem codul sa treaca. Totul este bine ar trebui sa treaca. Ooooo!!! se pare ca nu, ceva nu e in regula.

http://www.geocities.com/danbunea/articles/dotnet/tdd/image006.jpg

http://www.geocities.com/danbunea/articles/dotnet/tdd/image007.jpg

Avem un bug in test. Am uitat sa adaugam cea de-a 3-a linie la comanda. Schimbam codul testului, o adaugam si rulam testul din nou. Datorita relatiei apropiate intre test si cod, bugurile fie dintr-o parte fie din cealalta pot fi detectate cu usurinta.

http://www.geocities.com/danbunea/articles/dotnet/tdd/image008.jpg

Se pare ca avem o serie de teste simple care ne-au indrumat procesul de dezvoltare a unei bucati de cod. S-ar mai putea adauga si altele dar, cred ca acestea sunt suficiente pentru a face o scurta demonstratie asupra ceea ce inseamna TDD. Aceste teste ne vor ajuta in continuare, deoarece vor putea detecta instant cand ceva a afectat in mod negative codul pe care il testeaza.
</citat>

2. Unit testing cu Visual Studio 2022

Documentatie: Microsoft Visual Studio 2010 Unleashed, cap 9 (in special: ordonarea si organizarea testelor)
http://www.geekzone.co.nz/vs2008/4819 (VS 2008, dar se aplica si la 2019)
http://stephenwalther.com/blog/archive/2008/03/20/tdd-test-driven-development-with-visual-studio-2008-unit-tests.aspx (VS 2008, dar se aplica si la 2019)

Lista de asertiuni ce pot fi folosite:
http://stephenwalther.com/blog/archive/2008/03/20/tdd-test-driven-development-with-visual-studio-2008-unit-tests.aspx, sectiunea Creating Test Assertions.

Teme:

1. Folosind Test Driven Development, sa se implementeze o clasa Stiva (clasa generica sau cu componente intregi). Sa se scrie teste pentru a verifica:
   a. daca la initializare stiva raporteaza 0 elemente continute
   b. daca incercare de scoatere din stiva duce la aruncare de exceptie utilizator MyStackUnderflowException, folosind atributul ExpectedExceptionAttribute
   c. daca dupa un sir de operatii stabilite de utilizator (mix de extrageri si adaugari) stiva are un anumit numar de elemente – clasa Assert
   d. daca dupa un sir de operatii stabilite de utilizator (mix de extrageri si adaugari) stiva are un anumit continut – incercati clasa CollectionAssert sau extrageti rand pe rand elementele si comparati-le cu cele asteptate.

e. daca impuneti o limita maxima pentru stiva, testate daca incercarea de a pune mai multe elemente decat limita maxima duce la aruncarea unei exceptii utilizator MyStackOverflowException.
f. Setati ca pe diferite metode de test sa fie pusi ca si posesori diferiti programatori fictivi. Setati si o descriere scurta pentru fiecare metoda de test.

Daca pe statia locala nu este instalat Visual Studio 2022, se lucreaza pe masina virtual de pe d:\ si se lanseaza local folosind vmware player.

2. De multe ori, codul "mostenit" nu a fost scris in spiritul TDD. Se poate totusi sa se creeze si apoi sa se dezvolte un schelet de teste pornind de la codul scris, dand click dreapta pe codul ce se vrea a avea teste generate si apoi "Create unit tests...". Scrieti o clasa de tip coada generica, fara a folosi TDD sau unit testing si apoi generati teste si schimbati codul de testare cu unul asemanator cu cel de la punctul 1.

~~**3.** Se pot testa si metodele private ale unei clase. Creati o clasa simpla cu metoda statica si testati i functionalitatea, conform http://stephenwalther.com/blog/archive/2008/03/20/tdd-test-driven-development-with-visual-studio-2008-unit-tests.aspx, sectiunea "**Testing Private Methods, Properties, and Fields**". Remarcati codul care seteaza testul ca fiind inconcluziv, manevra folosita pentru a semnala testele ce nu sunt implementate complet.~~ Manevra nu mai este posibila incepand cu VS 2012.

4. Cititi despre Unit testing de la adresa http://en.wikipedia.org/wiki/Unit_testing.