

Introducere

Test-Driven Development (TDD)

Reguli [Kent Beck]

- *Never write a single line of code unless you have a failing automated test*
- *Eliminate duplication*

Manifesto for Agile Software Development

We are uncovering better ways of developing software
by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

[beck-manifesto]

=> Agile ideals are inherently goal-oriented

Tipuri de teste automate în TDD

Testele programatorilor

- *un fel de* testare a componentelor – scop diferit
- de preferat a fi scrise în același limbaj ca și codul sursă

Testele clienților

- specifică funcționalitatea de care clientul are nevoie
- numite teste de acceptanță
- scrise într-un limbaj pe care clientul îl înțelege -> clientul scrie teste - Framework for Integrated Test (FIT)
<http://fit.c2.com>

Red/Green/Refactor [Kent Beck]

Procesul de implementare a fiecărui test

1. Scrierea testului
2. Compilarea testului - obligatoriu trebuie să „pice” testul (pt. că nu este nimic implementat, nu există încă funcțiile apelate)
3. Implementarea minimală a codului sursă – cât să compileze
4. Rularea testului – testul trebuie să „pice”
5. Implementarea minimală a codului sursă – cât să „treacă” testul
6. Rularea testului – testul trebuie să „treacă”
7. Refactorizare – pt. claritatea codului și eliminarea duplicității
8. Repetarea pasului 1

Red/Green/Refactor - utilitate

- pași cât mai mici
- cu cât pasul e mai mic, cu atât mai ușor de găsit o eroare
- testele produc feedback imediat – nici nu mai e nevoie de debugger (rularea pas cu pas a programului) – se va ști locul în care s-a produs defectul
- siguranță în dezvoltare

Exemplu implementare TDD [James Newkirk]

Problemă

Implementarea unei structuri de date de tip stivă, care să permită următoarele operații: Push, Pop, Top și IsEmpty.

Lista testelor

- Crează *Stivă* și verifică *IsEmpty=true*
- Adaugă un obiect în stivă și verifică *IsEmpty=false*
- Adaugă un obiect, scoate obiectul din stivă și verifică *IsEmpty=true*
- Adaugă un obiect în stivă, memorează, scoate obiect și verifică dacă obiectele sunt identice
- Adaugă trei obiecte în stivă, memorează, apoi scoate trei obiecte din stivă și verifică dacă sunt scoase în ordinea corectă
- Scoate obiect dintr-o stivă goală
- Adaugă un obiect, apelează *Top* și verifică dacă *IsEmpty=false*
- Adaugă un obiect, memorează, apelează *Top* și verifică dacă cele două obiecte sunt identice
- Apelează *Top* pe o stivă goală

Alegerea primului test

- Cel mai simplu
- Cel mai apropiat de esența problemei

Test1: Crează unui obiect de tip stivă și verifică dacă *IsEmpty=true*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void Empty()
    {
        Stack stack = new Stack();
        Assert.IsTrue(stack.IsEmpty);
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public class Stack
{
    private bool isEmpty = false;

    public bool IsEmpty
    {
        get
        {
            return isEmpty;
        }
    }
}
```

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private bool isEmpty = false;

    public bool IsEmpty
    {
        get
        {
            return true;
        }
    }
}
```

Test2: Adaugă un obiect în stivă și verifică dacă *IsEmpty=false*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushOneTest()
    {
        Stack stack = new Stack();
        stack.Push("first element");
        Assert.IsFalse(stack.IsEmpty,
            "After Push, IsEmpty should be
false");
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public void Push(object element)
{
}
```

Nunit: .PushOneTest : After Push, IsEmpty should be false

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public void Push(object element)
{
    IsEmpty=false;
}
```

PAS 4. Refactorizare

```
[TestFixture]
public class StackFixture
{
    private Stack stack;

    [SetUp]
    public void Init()
    {
        stack = new Stack();
    }

    [Test]
    public void EmptyTest()
    {
        Assert.IsTrue(stack.IsEmpty);
    }

    [Test]
    public void PushOneTest()
    {
        stack.Push("first element");
        Assert.IsFalse(stack.IsEmpty,
            "After Push, IsEmpty should be false");
    }
}
```

Test3: Adaugă un obiect în stivă, scoate obiect și verifică dacă *IsEmpty=true*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PopTest()
    {
        stack.Push("first element");
        stack.Pop();
        Assert.IsTrue(stack.IsEmpty,
            "After Push - Pop, IsEmpty should be
true");
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public void Pop()  
{  
}
```

Nunit: .PopTest : After Push - Pop, IsEmpty should be true

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public void Pop()  
{  
    isEmpty = true;  
}
```


Test4: Adaugă un obiect în stivă, memorează obiect, scoate obiect și verifică dacă obiectele sunt identice

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushPopContentTest()
    {
        int expected = 1234;
        stack.Push(expected);
        int actual = (int)stack.Pop();
        Assert.AreEqual(expected, actual);
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

```
public object Pop()  
{  
    isEmpty = true;  
    return null;  
}
```

Nunit: .PushPopContentTest : System.NullReferenceException :
Object reference not set to an instance of an object.

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private bool isEmpty = true;
    private object element;

    public bool IsEmpty    {
        get{
            return isEmpty;
        }
    }

    public void Push(object element)    {
        this.element = element;
        isEmpty = false;
    }

    public object Pop()    {
        isEmpty = true;
        object top = element;
        element = null;

        return top;
    }
}
```

PAS 4. Refactorizare

```
public class Stack
{
    private object element;

    public bool IsEmpty
    {
        get
        {
            return (element == null);
        }
    }

    public void Push(object element)
    {
        this.element = element;
    }

    public object Pop()
    {
        object top = element;
        element = null;

        return top;
    }
}
```

Test5: Adaugă trei obiecte în stivă, memorează obiecte, scoate obiecte și verifică dacă obiectele sunt scoase în ordinea inversă adăugării

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
public void PushPopMultipleElementsTest()
{
    string pushed1 = "1";        stack.Push(pushed1);
    string pushed2 = "2";        stack.Push(pushed2);
    string pushed3 = "3";        stack.Push(pushed3);
    string popped = (string)stack.Pop(); Assert.AreEqual(pushed3, popped);
    popped = (string)stack.Pop();  Assert.AreEqual(pushed2, popped);
    popped = (string)stack.Pop();  Assert.AreEqual(pushed1, popped);
}
```

PAS 2. Codul compilează. Nunit → fail(red)

```
Nunit: .PushPopMultipleElementsTest :  
  expected:<"2">  
    but was:<(null)>
```

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public class Stack
{
    private ArrayList elements = new ArrayList();

    public bool IsEmpty
    {
        get
        {
            return (elements.Count == 0);
        }
    }

    public void Push(object element)
    {
        elements.Insert(0, element);
    }

    public object Pop()
    {
        object top = elements[0];
        elements.RemoveAt(0);
        return top;
    }
}
```

Test6: Scoate obiect dintr-o stivă goală (*IsEmpty=true*)

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    [ExpectedException(typeof(InvalidOperationException))]
    public void PopEmptyStackTest()
    {
        stack.Pop();
    }
```


PAS 2. Codul compilează. Nunit → fail(red)

Nunit: .PopEmptyStackTest : Expected: InvalidOperationException
but was ArgumentOutOfRangeException

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public object Pop()
{
    if(IsEmpty) throw
new InvalidOperationException("cannot pop an empty
stack");

    object top = elements[0];
    elements.RemoveAt(0);
    return top;
}
```

Test7: Adaugă un obiect, apelează *Top* și verifică dacă *IsEmpty=false*

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushTopTest()
    {
        stack.Push("42");
        stack.Top();
        Assert.IsFalse(stack.IsEmpty);
    }
```

PAS 2,3. Implementarea codului pentru compilare. Nunit → success(green)

```
public object Top()  
{  
    return null;  
}
```

Test8: Adaugă un obiect, memorează, apelează *Top* și verifică dacă obiectele sunt identice

PAS 1. Scrierea testului. Nu va trebui să compileze

```
[Test]
    public void PushTopContentCheckOneElementTest()
    {
        string pushed = "42";
        stack.Push(pushed);
        string topped = (string)stack.Top();
        Assert.Equals(pushed, topped);
    }
```

PAS 2. Implementarea codului pentru compilare. Nunit → fail(red)

Nunit: PushTopContentCheckOneElement :
expected:<"42">
but was:<(null)>

PAS 3. Implementarea minimală a codului pentru Nunit → success(green)

```
public object Top()  
{  
    return elements[0];  
}
```