# Jedinično testiranje pomoću xUnit alata (u C#.NET aplikacijama)

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Nikola Kovačević, 1104/19
nik_ko_@outlook.com

15. januar 2020.

**Sažetak**

U ovom sažetom ali sveobuhvatnom radu ćemo proći kroz sve najbitnije delove jediničnog testiranja (eng. *Unit Testing*) pomoću *xUnit frameworka* na *.NET* platformi. Rad počinjemo sa navođenjem osnovnih definicija i koncepata automatskog (eng. *Automated testing*) i samim tim jediničnog testiranja, bez zalaženja u ogromnu dubinu. Zatim nastavljamo sa objašnjenjem šta su to *xUnit* alati, šta ih karakteriše i šta oni moraju da ispunjavaju da bi bili član familije tih alata. Takođe govorimo o njihovim osnovnim celinama i načinu na koji funkcionišu. I naravno, kroz tekst dajemo primere koristeći konkretan alat *nUnit* u *C#.NET* okruženju.

## Sadržaj

# 1 Uvod

Razvoj aplikacija uvek sa sobom nosi i proizvođenje grešaka koje mogu da utiču ili narušavaju specifikacije same aplikacije. Kao posledicu toga, testiranje postaje veliki deo ciklusa razvoja aplikacija (eng. *Software Development Lifecycle*) i doprinosi smanjenju i pronalaženju dobrog broja grešaka. *Jedinično testiranje* se izdvaja po tome što je veoma isprepleteno sa samom fazom implementacije rešenja. Kako bi se sam proces testiranja ubrzao i unapredio, implementiraju se razni alati za *automatsko testiranje*.

Među velikim brojem alata, *xUnit* se izdvaja kao jedan od najstarijih i najkorišćenijih. Važno je imati u vidu da postoji bitna dinstikcija između **xUnit i xUnit.NET** naziva. Kada kažemo **xUnit**, govorimo o familiji alata, o jednoj klasi, čiji se članovi zasnivaju na istim principima. Dok naziv **xUnit.NET** koristimo za konkretan primer jednog takvog alata. Za sve xUnit alate je specifično to što se **x** u nazivu zamenjuje sa početnim slovom jezika u kojem se koriste. Tako alat familije *xUnit* za Javu nazivamo *jUnit*, dok za C++ imamo *CppUnit*, za Haskell *hUnit* i slično. Neki od alata za .NET platformu (C#, F#, VB) su *nUnit* i *xUnit.NET*. Mi ćemo se fokusirati na koncepte koje implementiraju svi *xUnit* alati a kao primer kroz kodove ćemo koristiti konkretan alat **nUnit** u C#.NET.

# 2 Jedinično testiranje

Postoji ogroman broj stvari na koje treba obratiti pažnju pri pisanju testova, kao što i postoji veliki broj *obrazaca* (eng. *design pattern*) testiranja za rešavanja prepoznatljivih problema. Ovde ćemo navesti samo osnovne definicije jediničnog testiranja bez zalaženja u detalje, različite tačke gledanja (filozofije) i tehničke specifičnosti.

**Definicija 2.1** ***SUT (engl. System Under Test)*** *je naziv za komponentu ili sistem koja je cilj testiranja, odnosno objekat nad kojim vršimo testiranje. Može biti u vidu pojedinačnih metoda i klasa ili čak njihovih skupova [4].*

**Definicija 2.2** ***DOC (engl. Depended-on Component)*** *nazivamo bilo koju komponentu koja je potrebna SUT komponenti za njeno izvršavanje, odnosno testiranje. Na primer, ovo može biti kompozitni objekat unutar objekta koji se testira [4].*

**Definicija 2.3** ***Unit (Unit of Work)*** *je skup akcija koji se dešavaju između poziva nekog javnog metoda u SUT i jednog vidljivog završnog rezultata. Rezultat je vidljiv kroz javni interfejs (engl.* Application Programming Interface*) bez pristupa unutrašnjem stanju jedinice. Ovo je kod jedinice (engl.* Unit*) koju testiramo [4].*

**Definicija 2.4** ***Jedinični test (engl. Unit test)*** *je automatizovani deo koda koji testira jedinicu (engl.* Unit*)) tako što proverava da li je krajni rezultat rada te jedinice jednak nekoj pretpostavci. Krajni rezultat je najčešće objekat koji SUT vraća a nekad može biti i provera ponašanja (engl.* behavior*) [2].*

Ono što je zanimljivo za jedinično testiranje je to što u većini slučajeva testove piše sam programer koji uporedo razvija *SUT*. Jedinično testiranje pripada grupi *dinamičkog testiranja*, što znači da se testira rad komponente za vreme izvršavanja (engl. *runtime*). Pored toga, jedinični testovi su u potpunosti automatizovani (engl. *Fully automated*). Odnosno,

čitav test implementiramo kao kod (uglavnom na jeziku koji je korišćen za razvoj aplikacije). Test izvršava čitav ciklus od postavljana okruženja za testiranje, pozivanja ili izvršavanja jedinice pa do poređenja rezultata sa pretpostavkom (engl. *assumption*). Takođe imaju osobine samoprovere (engl. *self-checking*), što znači da sam test vrši proveru svoje uspešnosti.

Kod objektno-orijentisanog programiranja predmet testiranja su uglavnom klase i metode. Poželjno je za svaki slučaj koji proveravamo napraviti poseban test kako bi se lakše izolovao problem, ukoliko on nastane. Testove često grupišemo na osnovu nekih konvencija, zavisno od situacije. Mnoga radna okruženja (engl. *Integrated Development Environment (IDE)*) pružaju podršku u vidu grafičkog interfejsa (engl. *Graphical User Interface* ) ili kroz interfejs komandne linije (eng. *Command Line Interface*). Vredan spomena je i ***razvoj vođen testovima*** (engl. *Test-driven development (TDD)*) gde jedinično testiranje igra ključnu ulogu.

## 2.1 Prednosti i ciljevi

Nabrojaćemo i objasniti ciljeve koje automatsko i jedinično testiranje treba da postignu:

- Poboljšanje kvaliteta (engl. *Quality Assurance*) nastaje jer nam pisanje testova daje uvid u šta $SUT$ treba da radi i pre same implementacije testirane komponente. Samo pisanje testova nas tera na analizu testirane jedinice što doprinosi boljem razvoju [2].
- Sprečavanje bagova pre njihovog pojavljivanja. Pored toga, zbog lokalizacije testova, grešku je dosta lakše precizirati i izdvojiti jer su testovi izvršeni na jedinicama koje su kratke i jednostavne [4].
- Testovi su vid dokumentacije jer nose dosta informacija sa sobom o tome kako $SUT$ funkcioniše [4, 2].
- Testovi treba da budu jednostavni za pokretanje. Pored toga, potrebno je ispuniti osobine potpune automatizacije (engl. *fully atomated*), samoprovere (engl. *self-checking*) i ponovljivosti (engl. *test repeatability*) [2].
- Testovi treba da budu jednostavni za pisanje i održavanje, ekspresivni i modularni. Poželjno je i da budu što više međusobno isključivi u slučajevima koje pokrivaju.
- Pisanje testova ne bi trebalo da menja $SUT$ iako je poželjno imati u vidu testabilnost (engl. *testability*) pri dizajnu aplikacije [2].

Pored toga, nabrojaćemo i neke direktne posledice ovakvog vida testiranja:

- Omogućavaju otklanjanje bagova u hodu u veoma ranoj fazi razvoja aplikacije.
- Nakon refaktorizacije (engl. *Refactorization*) dovoljno je pokrenuti ponovo testove čime ćemo proveriti da li smo greškom izmenili ponašanje sistema [2].
- Ovakav vid testiranja donekle zahteva bolju arhitekturu aplikacije. Na primer, da bi jedinično testiranje bilo moguće, potrebno je obezbediti sve $DOC$ objekte koje $SUT$ zahteva pre njegovog pokretanja. Često se DOC objekti apstrahuju i koriste se njihove lažne (engl. *dummy, mock*) implementacije pri testiranju. [2, 4].
- Omogućava razvoj vođen testovima (engl. *Test-driven development*).

## 2.2 Ograničenja

Kao što to uvek biva, svaku prednost prate neke mane ili ograničenja o kojima treba voditi računa. Ovakve zamke (engl. *pitfalls*) su odgovornost programera da ih izbegne ili minimalizuje. Navodimo neka ograničenja automatskog testiranja [2]:

- *Osetljivost ponašanja* se javlja usled menjanja ponašanja testiranog sistema. Kao posledicu ima da neki već napisani testovi neće proći i da se moraju ponovo implementirati.
- *Osetljivost interfejsa* se javlja pri testiranju logike kroz korisničke interfejse koje je veoma teško simulirati i veoma često podložno promenama.
- *Osetljivost podataka* se javlja kada testovi zavise od nekih spoljašnih podataka. Testovi mogu biti neotporni na njihovo menjanje.

Jedinično testiranje takođe ima ograničenja koja su specifična za ovaj vid testiranja [2]:

- Testovi se fokusiraju na jednostavne i pojedinačne celine sistema. Za testiranje njihovog rada kao jedan kompleksni sklop neophodno je koristiti neku drugu vrstu testiranja (npr. integraciono).
- Javlja se ogroman broj testova jer je potrebno ispitati veliki broj kombinacija ulaza i izlaza.
- Problem sa nitima (engl. *threads*) i nedeterminističkim kodom.
- Potrebna je postavka okruženja kako bi se simuliralo ponašanje testirane jedinice u celokupnom sistemu.
- Problem sa kontrolom toka (engl. *control flow*).

# 3 xUnit familija okvira (framework family)

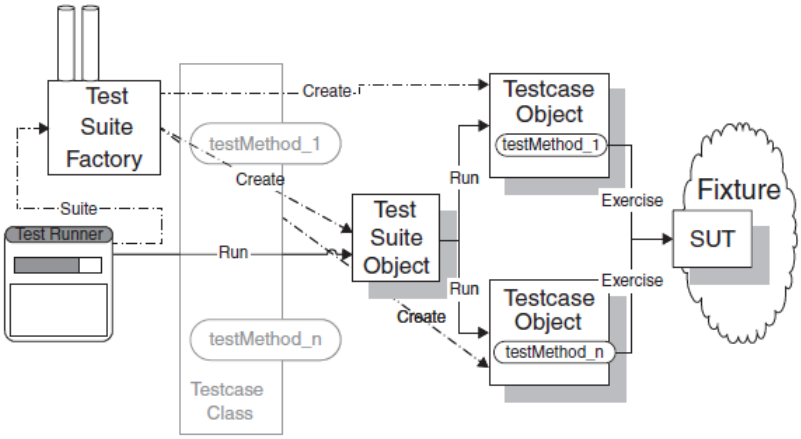
xUnit okviri (engl. *frameworks*) potiču od *sUnit* okvira, koji je dizajnirao Kent Bek 1998. godine. Napravljen je sa cijljem da olakša automatsko testiranje tako što omogućuje testiranje individualnih komponenti izdvojenih od ostatka aplikacije, lako pokretanje i efikasno izvršavanje testova [2].

## 3.1 Definicije i koncepti

Objasnićemo osnovnu struktura ovog alata primenjenog u objektnom-orijentisanom jeziku. Alat se sastoji od svega nekoliko zasebnih komponenti koji su međusobno povezani poznatim *projektnim obrascima* (engl. em design patterns)[3, 2]. Svi *xUnit* alati prate sličnu arhitekturu čiji je dijagram prikazan na sledećoj slici:

Pored *SUT* termina za koji smo već uveli definiciju, definišemo i ostale komponente sa dijagrama:

**Definicija 3.1** *Test slučaj (engl.* Test case*) definišemo kao skup akcija koje se izvršavaju da se proveri funkcionalnost neke jednice. Sadrži sve korake testa, pripremu podataka i preduslova, kao i proveru rezultata nakon izvršavanja [2].*




Slika 1: xUnit arhitektura [2].

Test implementiramo kao jednu metodu ***Test method*** gde sam proces testiranja izvršavamo kroz 4 faze pozivanjem te metode. Za vreme izvršavanja, za svaku napisanu metodu koja predstavlja test se kreira *Testcase Object* koji ima ulogu komandnog objekta (engl. *Command object, Command Design pattern*) [2]. Komandni objekat obuhvata test i sve komponente neophodne za njegovo izvršavanje [3].

```
1000 [Test]
     public void Test42(){
1002   //Prepare
       //Exercise
1004   //Verify
       //Teardown
1006 }
```

Listing 1: nUnit testove prepoznaje preko atributa.

**Definicija 3.2** ***Test Suite*** *je skup testova, često grupisanih sa nekim smislom (npr. svi testovi jedne klase mogu biti u jednom Test Suite objektu) [4, 2].*

Test Suite implementiramo kao klasu *Testcase Class* u kojoj se nalazi skup metoda koje predstavljaju testove. Za vreme izvršavanja kreira se *Test Suite Object* u kojem su agregirani svi *Testcase* objekti (kompozitni projektni obrazac) [3, 2].

**Definicija 3.3** ***Test Fixture*** *označava sve što je neophodno da bismo pokrenuli neki test nad SUT objektom [2].*

Obično se pod *Test Fixture* podrazumevaju instanca koju testiramo i svi njeni *DOC* objekti koji su neophodni za njeno pokretanje. Često se *Test Fixture* uzima kao sinonim za *Testcase Class* u slučajevima kada svi testovi jedne klase (jednog *Suite* objekta) dele istu *Test Fixture* instancu (ovo je slučaj kod **nUnit** alata) [1].

```
1000 [TestFixture]
public class Tests
1002 {
   [Test]
1004   public void Test0(){
   }
1006   ................
   [Test]
1008   public void Test42(){
   }
1010 }
```

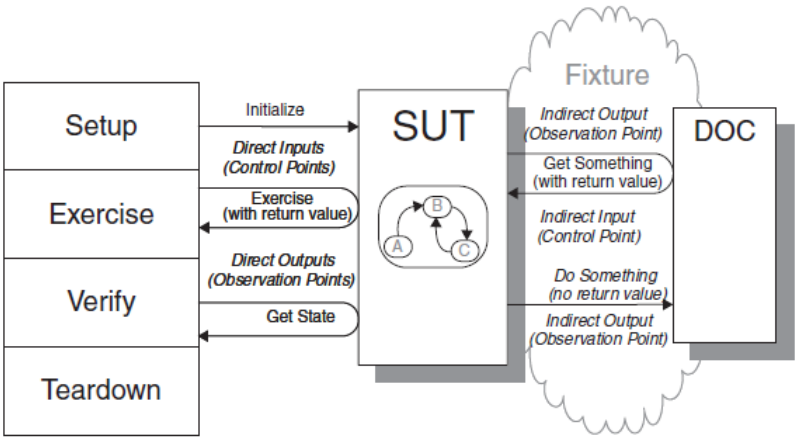
Listing 2: nUnit prepoznaje Test Fixture/Suite preko atributa [1]

**Definicija 3.4** ***Test Suite Factory*** *je klasa koja ima ulogu pronalaženja i kreacije testova na zahtev Test Runner objekta [3, 2].*

**Definicija 3.5** ***Test Runner*** *pokreće testove kroz objekte koje mu factory pruža i obrađuje njihove rezultate [2].*

Bitno je uzeti u obzir da i Test Suite i Testcase klase implementiraju isti interfejs za komandni obrazac (engl. *Command pattern*). Taj interfejs nameće klasama da implementiraju jednu metodu **run()** koja izvršava koju god komandu komandni objekat predstavlja. Ovim postižemo da *Test Runner* testove izvršava jednostavnim pozivanjem *run()* metode, bez ikakvog znanja na koji način se oni izvršavaju. Takođe dobijamo mogućnost izvršavanja većeg broja testova pomoću samo jedne akcije. To postižemo pozivanjem *run()* nad *Test Suite* objektom čime se iteriraju svi agregirani *Testcase* objekti nad kojima se takođe poziva *run()* [3, 2].

# 4 Implementacija testova




Slika 2: Izvršavanje testa [2].

Kod same implementacije testova, unutar svake metode koja predstavlja test, izdvajaju se 4 uzastopne faze:

- Instanciranje i podešavanje *Test Fixture* objekta (*Setup*).

- Izvršavanje testa nad *SUT* objektom, uglavnom kroz neki javni interfejs (*Exercising tests*).
- Poređenje rezultata sa očekivanjima (*Veryfing results*).
- Uništavanje *Test Fixture* objekata (*Teardown phase*).

Kao primer navodimo jednostavan kod. Faza podešavanja se svodi na kreaciju objekta koji testiramo. Izvršavamo metodu klase koju testiramo i poredimo njen rezultat sa pretpostavkom.

```
1000 public interface Cpu {
       int calculate();
1002 }

1004 public class LazyCpu : Cpu
     {
1006   public int calculate(){
         var date = DateTime.Now.AddYears(7500000);
1008     SleepUntil(date);
         return 42;
1010   }
     }
1012
     //Klasa koja se testira
1014 public class DeepThoughtComputer
     {
1016   private Cpu cpu { get; set; }

1018   public int MeaningOfLifeTheUniverseAndEverything {
         get {
1020      return cpu.calculate();
          }
1022   }

1024   public DeepThoughtComputer(){
         cpu = new LazyCpu();
1026   }

1028   public DeepThoughtComputer(Cpu cpu) {
         this.cpu = cpu;
1030   }
     }
1032
     ..................
1034
     [TestFixture]
1036 public class Tests
     {
1038   [Test]
       public void Test42()
1040   {
         var computer = new DeepThoughtComputer();
1042
         Assert.That(computer.MeaningOfLifeTheUniverseAndEverything,
1044             Is.EqualTo(42));
       }
1046 }
```

Listing 3: Primer testiranja.

## 4.1 Instanciranje i podešavanje

U zavisnosti od vrste *Test Fixture* objekta, zavisiće i izvođenje ove faze. *Test fixture* može biti [2]:

- ***Svež (engl. Fresh)***, što znači da se pre svakog pokretanja testa kreira nova instanca a nakon svakog testa ta instanca se uništava.

- ***Trajan (engl. Persistent)***, što znači da se koristi ista instanca kroz više testova. Instanca može da prolazi kroz promene kako se testovi smenjuju.

Ovu fazu možemo implementirati na početku svakog testa u metodi a možemo je i izdvojitu u zasebnu metodu *Testcase* klase. U zavisnosti da li želimo da koristimo metodu pre svakog testa ili samo jednom pre pozivanja bilo kojeg testa unutar klase, imamo različite atribute:

```
[TestFixture]
public class Tests
{
  [SetUp]
  public void InitBeforeEachTest(){ /* ... */ }
  [TestFixtureSetUp]
  public void InitOnce(){ /* ... */ }

  [Test]
  public void Test42() { /* ... */ }
}
```

Listing 4: Setup metod

## 4.2 Izvršavanje testa i rezultat

Nakon svih postavki i kreiranja potrebnih komponenti (*Fixture*) objekata izvršavamo testiranje. Činimo to jednostavnim pokretanjem *SUT* objekta, uzimanjem rezultata pokretanja i poređenjem rezultata sa očekivanjem. Većina *xUnit* alata ima ugrađene funkcije za verifikaciju, odnosno poređenje rezultata.

*nUnit* nudi funkciju *Assert.That* kojoj prosleđujemo dobijeni rezultat i objekat ograničenja (engl. *constraint object*). Time proveravamo da li prosleđeni rezultat zadovoljava dato ograničenje [1].

```
[Test]
public void Test42()
{
  var computer = new DeepThoughtComputer();

  Assert.That(computer.MeaningOfLifeTheUniverseAndEverything,
                       Is.GreaterThan(42));
}
```

Listing 5: Poređenje 2 cela broja.

Dok se prethodni test zasniva na poređenju jednog rezultata, određeni kod možemo testirati i time da li on baca neki izuzetak (engl. *exception*):

```
[Test]
public void Test42()
{
  /* .... */

  Assert.Throws<ArgumentException>(SomeMethod);
}
```

Listing 6: Exception

## 4.3 Čišćenje

Fazu čišćenja (engl. *Teardown phase*) možemo izdvojiti u metod koji ćemo označiti atributom. U zavisnosti od toga da li želimo da se čišćenje izvršava posle svakog testa ili samo jednom imamo dve vrste atributa [1]:

```
[TestFixture]
public class Tests
{
  [TearDown]]
  public void CleanupAfterEachTest(){ /* ... */ }
  [OneTimeTearDown]
  public void CleanupOnce(){ /* ... */ }

  [Test]
  public void Test42() { /* ... */ }
}
```

Listing 7: Teardown metod.

## 4.4 Lažni objekti

Lažiranje objekata je još jedna veoma bitna radnja koja se odvija u fazi pripreme testa. Zapravo, veoma često nije praktično ili nemamo na raspolaganju *DOC* (deo aplikacije koji je potreban delu koji testiramo) pa možemo napraviti takozvani **lažni** objekat. Aplikaciju koju pravimo ne smemo menjati zbog testiranja ali je poželjno imati u vidu testabilnost pri njenom dizajniranju.

Tako na primeru našeg koda, metoda klase koju testiramo se jako dugo izračunava. Međutim, imamo mogućnost da to ubrzamo prosleđivanjem lažnog *DOC* objekta koji nije sastavni deo prave aplikacije.

```
public class InfinitelyFastTestCpu : Cpu
{
  public int calculate(){
    return 42;
  }
}

[TestFixture]
public class Tests
{
  [Test]
  public void Test42(){
  var computer = new DeepThoughtComputer(new InfinitelyFastTestCpu
     ());

    Assert.That(computer.MeaningOfLifeTheUniverseAndEverything,
            Is.EqualTo(42));
  }
}
```

Listing 8: Lažan objekat.

Zato što je aplikacija dizajnirana tako da se komponente oslanjaju na apstrakcije i *umetanje zavisnosti (engl. Dependency injection)*, imamo dosta veću slobodu pri testiranju [3].

Ovo je tema za sebe i veoma često se koriste dodatni alati za pravljenje ovakvih objekata. Postoji i više vrsta ovakvih *DOC* objekata u zavisnosti od njihove uloge (*stub, mock, fake, double*) [2].

# 5 Zaključak

Obradili smo arhitekturu *xUnit* alata i osnovne konkretne operacije koristeći *nUnit framework*. Stvorili smo neku opštu sliku kako jedan test i njegovo izvršavanje izgledaju. Iako, na prvi pogled, testiranje može da izgleda jednostavno i da se svodi na pisanje *Assert* funkcija, to u praksi uopšte nije slučaj. Na realnim projektima treba da se donese veliki broj

teških odluka, najčešće u vezi sa arhitekturom projekta i testiranja. Literatura koja se koristila kao reference za ovaj tekst pruža veliki broj rešenja i projektnih obrazaca za razne situacije u praksi jediničnog testiranja, te se preporučuje kao odličan izvor informacija za dalje istraživanje.

# Literatura


[1] nUnit Official Documentation. on-line at: https://github.com/nunit/docs/wiki/NUnit-Documentation.

[2] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[3] Gang of Four. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] Roy Osherove. *The Art of Unit Testing*. Manning Publications, 2009.