Unit testovi

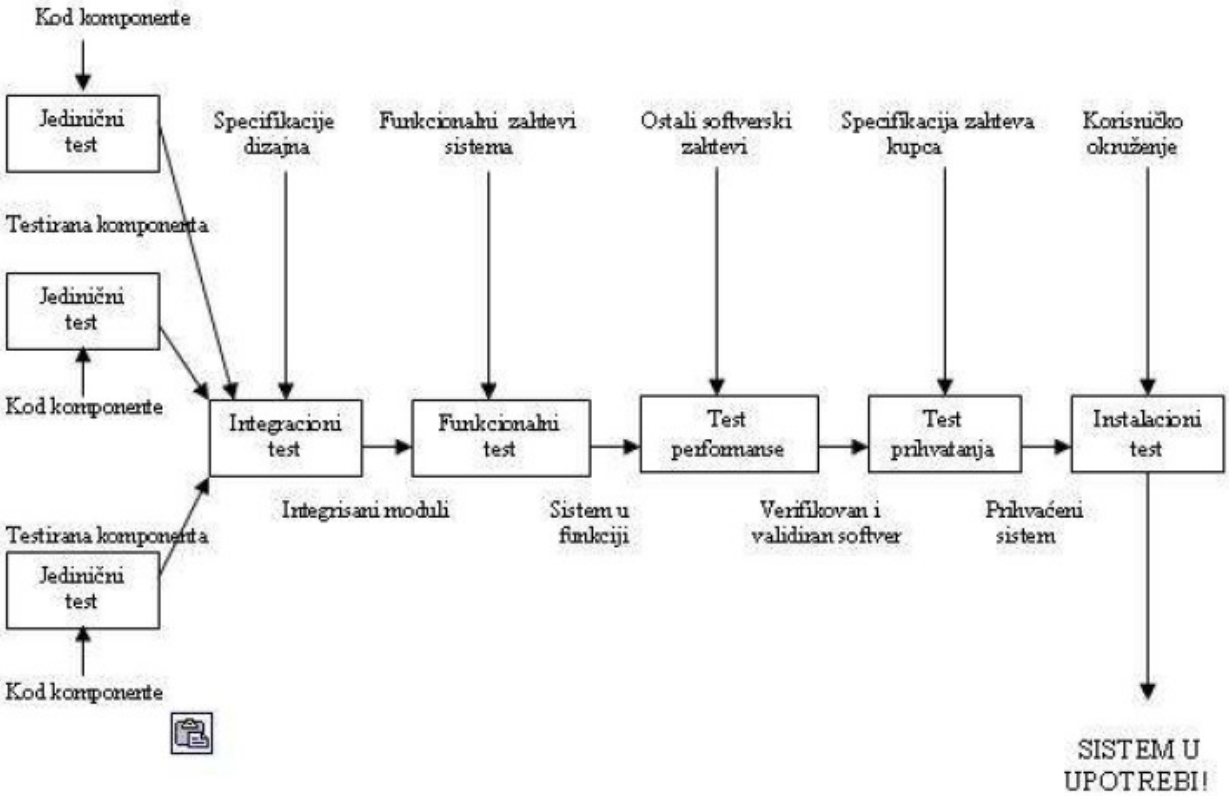
Razvoj softvera vođen testovima (test-driven development - TDD)

Mnogi programeri posmatraju testiranje kao dokaz da njihovi programi ispravno rade. Ali, zapravo, programi se testiraju da bi se dokazalo postojanje grešaka. Pošto je cilj da se greška otkrije, test se smatra uspešnim samo ako se greška otkrije ili ako dođe do otkaza u toku testiranja.

Test Driven Development je tehnika razvoja softvera koja podrazumeva često testiranje napisanog kôda. Test se piše pre kôda, a nakon toga se piše izvorni kôd koji treba da zadovolji test. Refaktorisanjem se taj kôd dalje prečišćava i pojednostavljuje, ali osnovno je da test koji se jednom verifikovao mora da se iznova verifikuje, pri svim sledećim izmenama. To je tzv. jedinično testiranja (unit test).
Postoje i funkcionalni testovi kroz test primere, odnosno test slučajeve (test case) koje izvodi sam klijent. Testovi korisnosti i korisničkog interfejsa se takođe izvode kod klijenta. TDD strategija je počela da privlači pažnju početkom 2000-te, kao aspekt Ekstremnog programiranja.
Postoje razni aspekti TDD razvoja, kao što su na primer principi „Keep It Simple, Stupid“ (KISS), i „You Ain’t Gonna Need It“ (YAGNI). Fokusirajući se samo na kôd koji mora da zadovolji test, dizajn može biti jasniji i čistiji nego što se to obično može postići nekom drugom metodom.
TDD zahteva da programeri prvo napišu test case koji ne prolazi, kako bi bili sigurni da test zaista radi ispravno i da može da uhvati grešku.

Faze testiranja

Kada se razvija veliki sistem, testiranje zahteva više faza.



Prvo se zasebno testira svaka programska komponenta, nezavisno od ostalih delova sistema. Takvo testiranje modula (**jedinično testiranje, unit test**) proverava da li pojedinačne komponente ispravno funkcionišu sa svim očekivanim tipovima ulaza, u skladu sa dizajnom komponente.
Jedinično testiranje treba vršiti u kontrolisanom okruženju tako da tim za testiranje može komponenti koja se testira da predaje unapred definisan skup podataka, i da posmatra izlazne akcije

i rezultate. Takođe, tim za testiranje proverava unutrašnju strukturu podataka, logiku i granične uslove za ulazne i izlazne podatke.

Kada se završi jedinično testiranje skupa komponenti, sledi provera da li su interfejsi između komponenti pravilno definisani i realizovani. **Integraciono testiranje** je postupak provere da li sistemske komponente sarađuju kao što je opisano u specifikacijama dizajna sistema i programa.

Funkcionalnim testiranjem se proverava da li integrisani sistem zaista izvršava funkcije opisane u specifikaciji zahteva. Kao rezultat dobija se funkcionalni softverski sistem.
Funkcionalni test predstavlja poređenje sistema koji se gradi sa funkcijama zahteva kupca.

Testiranjem performanse sistem se poredi sa ostatkom hardverskih i softverskih zahteva. Kada se to testiranje uspešno obavi u stvarnom radnom okruženju kupca, dobija se validiran sistem.

U sledećem koraku, zajedno sa kupcem se obavlja završni **test prihvatanja (acceptance test)**, u kojem se proverava usklađenost sistema sa opisom zahteva kupca. Kada se obavi završni test, prihvaćeni sistem se instalira u okruženje u kome će se koristiti. Konačni **instalacioni test** potvrđuje da li sistem i dalje ispravno funkcioniše.

Automatizovani alati za testiranje (JUnit, Nunit)

Današnji softverski sistemi su veoma velike i kompleksne aplikacije. Zato je važno koristiti alate za automatizovano izvršavanje testova, jer omogućavaju veoma veliki broj slučajeva za testiranje, bez kojih sistem bi bio površno testiran.
Alati za izvršavanje testova mogu da se integrišu sa drugim alatima radi izgradnje okruženja za sveobuhvatno testiranje. Alati se često povezuju sa bazom podataka za testiranje, mernim alatima, alatima za analizu koda, editorima teksta i alatima za simulaciju i modelovanje da bi se automatizovao što je moguće veći deo procesa testiranja.

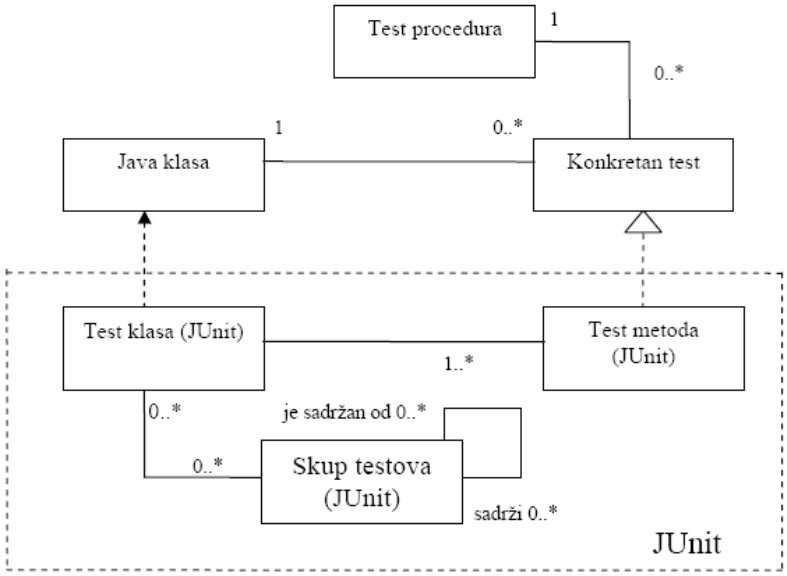
Međutim, utvrđivanje da greška postoji ne znači da je greška i pronađena. Zato će automatizovano testiranje zahtevati i ljudski trud u traženju izvora za nastali problem.

JUnit je open source okvir za testiranje, napisan za Java programski jezik. Međutim, postao je toliko popularan da je preveden i prilagođen za korišćenje u drugim programskim jezicima: C# (NUnit), Python (PyUnit), Fortran (fUnit), C++ (CPPUnit). Pomoću ovih alata, moguće je automatizovano izvršavati jedinične testove iz IDE razvojnog okruženja.

JUnit je integrisan u mnoga razvojna okruženja (Eclipse, NetBeans, JBuilder, JDeveloper, Sun Java Enterprise Studio) . Prednost integrisanog JUnita je u tome što je olakšan rad pri testiranju koda jer se vrši preko bogatog grafičkog interfejsa, a mnoge operacije, kao što je na primer kompajliranje testova i izvršavanje testova, su automatizovane.

JUnit je nastao na osnovu originalnog okruženja za testiranje u SmallTalku, koje je napisao Kent Beck (SUnit). Nakon toga su Kent Beck i Erich Gamma napravili JUnit za J2SE i J2EE platformu.

Junit omogućava kreiranje testova za bilo koju Java klasu, odnosno za bilo koju njenu metodu.



Svaka Java klasa može da se testira, ali ne moraju sve. Kada se testira klasa, koristi se jedna ili više test procedura. Test procedura je nezavisna od klase, pa može ista test procedura da se primeni na različitim klasama. Kad se test procedura primeni na određenu klasu, onda nastaje konkretan test ili više konkretnih testova. Konkretan test je u vezi sa samo jednom procedurom i sa samo jednom klasom.

U JUnit-u, svaki konkretan test može da se implementira kao jedna test metoda.
JUnit test metoda je Java metod čije ime počinje sa „test“ (na primer: testIzracunajRacun) i pripada JUnit test klasi (TestCase).
Svaka test klasa može da ima jednu ili više test metoda, a test metoda može da pripada samo jednoj test klasi.
Svaka test klasa odgovara samo jednoj Java klasi, odnosno onoj koja se testira. Test klase mogu da se organizuju u veće celine – skupove testova (TestSuite).
Svaki skup sadrži jednu ili više test klasa, a može da sadrži i jedan ili više drugih skupova. Na taj način se može pozvati izvršavanje svih testova, nekih testova koji pripadaju određenom skupu ili pojedinačnih testova.

JUnit može da se koristi sve vreme tokom razvoja softvera, jer se prvo piše test, a posle toga kodira, odnosno pišu se jedinični testovi kojima se testira svaka metoda. Korišćenjem Junita kreira se baza testova, a testovi se izvode organizovano i detaljno. Tako se stvara sigurnost da sve što je implementirano je potpuno funkcionalno.

Prednosti testiranja koda korišćenjem JUnit-a su sledeće:

- JUnit je veoma jednostavan za korišćenje, a testovi se pišu veoma lako.
- Čuva se baza testova, tako da se može menjati, dopuniti i ponovo izvršavati, što je važno, jer se može proveriti da li je program zadržao svu funkcionalnost koju je imao ranije, odnosno vrši se regresiono testiranje (izvršavaju se svi postojeći testovi) i da li novi deo funkcioniše kako bi trebalo (napišu se i dodaju novi testovi u bazu i izvrše).
- Pri izvršavanju, testovi napisani u JUnit-u, vrše automatsku proveru rezultata i prijavljuju greške.
- Testovi se mogu organizovati u hijerarhijski povezane celine, što je važno kada je broj testova relativno veliki.
- Svaki programer može lako da nauči da piše testove, jer se pišu u Javi u formi običnih

metoda.

Osnovni ciklus slučaja testiranja

Svaki slučaj testiranja prati sledeći osnovni ciklus:

petlja kroz metode za testiranje:
```
{
  setUp(); // izvršava se logika za inicijalizaciju
  doTests(); // izvršava se metod za testiranje
  tearDown(); // izvršava se logika za završetak testa
}
```

Svaki jedinični test treba da sledi sledeće korake:
1. Treba da bude potklasa klase **junit.framework.TestCase.**
2. U metodu **setUp()** treba dodati neki kod za podešavanje.
3. Treba definisati test. Svaki test treba da kao povratni tip ima void (odnosno da ne vraća ništa), a njegovo ime treba da počinje sa test, na primer testMaxSize(), testConvertAmount(), testAdd() itd.
4. Resurse koje smo zauzeli preko funkcije setUp() osloboditi preklapanjem metoda tearDown().
5. Definisati skup testova, ako je potrebno da se testovi izvršavaju zajedno.

TestCase klasa je osnovna klasa za testiranje. Sadrži pomoćne metode za kreiranje i pokretanje testova. Kada se izvrši TestCase, automatski se pokreću sve metode klase koja počinje sa test. JUnit koristi refleksiju da odredi koje testove da pokrene, olakšavajući na taj način dodavanje novih metoda za testiranje u postojeći slučaj testiranja. Refleksija je proces kojim program može da uoči i modifikuje svoju strukturu i ponašanje.
TestCase uključuje i metode kojima se implementira „fixture“. „Fixture“ je artifakt u odnosu na koga se pokreću testovi. To nije nešto što se testira, nego sredstvo (resource) koje je potrebno da bi se test pokrenuo. Na primer, ako se testira pristup bazi podataka (database access), potreban je Connection objekat da bi se pristupilo tabelama baze. Ne testira se sama konekcija, nego je konekcija potrebna pre nego što se napiše ostatak testa. Slučaj testiranja će uključiti Connection kao „fixture“.
Metod setUp() framework automatski poziva pre nego što se pokrene svaki test.
Slično, tearDown() metod se poziva nakon što se pokrene svaki test. Na primer, setUp() uspostavlja konekciju na bazu, a tearDown() zatvara tu konekciju.

Primer : Testiranje klase SimpleTest

```
package junit.samples;
import junit.framework.*;

public class SimpleTest extends TestCase{
   protected int value1;
   protected int value2;

   public SimpleTest(String name) {super (name);}

  protected void setUp(){
      value1=2;
      value2=3;
   }
```

```
   public static TestSuite(){
      return new TestSuite(SimpleTest.class);
   }

public void testAdd(){
   double result=value1+value2;
     // forced failure result ==5
     assertTrue(result==6);
}

public void testDivideByZero(){
   int zero=0;
   int result=8/zero;
}

public void testEquals(){
  assertEquals(12, 12);
  assertEquals(12L, 12L);
  assertEquals(new Long(12), new Long(12));
  assertEquals("Size", 12, 13);
  assertEquals("Capacity", 12.0, 11.99, 0.0);
}

}
```

Napomena:
assertEquals() – upoređuje na jednakost dve vrednosti. Test prolazi ako su te dve vrednosti jednake.

assertTrue() - procenjuje parametar, tj. boolean promenljivu. Test prolazi ako je vrednost boolean promenljive true.

U datoteci SimpleTest.java nalazi se sledeće:
· Referencira se biblioteka iz okvira JUnit;
· Daje se ime klasi za testiranje (SimpleTest). Ta klasa treba da bude potklasa klase TestCase;
· Veši se preklapanje metode setUp() i podešavanje inicijalne vrednosti za promenljive u testu;
· Vrši se preklapanje osnovnog skupa testova i vraća novua istanca za taj slučaj testiranja;
· Klasa za testiranje implementira test "dodavanje brojeva", testAdd(). Generiše se poruka ako sabiranje ne uspe. Ovde se javlja greška radi demonstracije.
· Pravi se izuzetak, tako što se u metodi testDivideByZero() vrši deljenje sa nulom. Nastaje greška, što je suprotno u odnosu na neuspeh.
· Metoda testEquals() proverava jednakosti i vraća poruku o neuspehu.

Izvršavanjem SimpleTest-a dobija se izveštaj o 1 grešci i 2 otkaza
==========

Prvi NUnit alati obezbedivali su osnovne klase iz kojih su se razvijale klase za testiranje. Međutim, ovi sistemi imaju dosta ograničenja prilikom razvoja koda za testiranje zato što veliki broj programskih jezika (Java i C#) omogucava samo jednostruko nasleđivanje. Ovo znači da je refaktoring test koda nemoguc bez upoznavanja komplikovanih hijerarhija nasleđivanja.

.NET uvodi novi koncept programiranja koji rešava ovaj problem: atributi.
Atributi omogućavaju dodavanje meta-podataka kodu. Atributi ne uticu na izvršavanje samog koda, oni se najcešce koriste za dokumentovanje koda, ali moguce je koristiti ih kako bi dodali

informacije o .NET asembliju.

Primena atributa predstavlja osnovu za rad NUnit 2.0. Aplikacija Test Runer skenira kompajlirani kod kako bi našla attribute koji ukazuju koje su klase i metode kreirane za testiranje. Da bi se zatim refleksijom izvršile te metode. Nije potrebno izvlaciti test klase iz osnovne klase, vec treba koristiti prave atribute. NUnit omogucava dodavanje velikog broja atributa koji se koriste prilikom kreiranja Unit testova. Oni se koriste kako bi definisali test uredaje, test metode, setUp i tearDown metode. Postoje i atributi koji ukazuju na ocekivane greške ili koji ukazuju na test koji ce biti izostavljen.

Atribut TextFixture

Atribut TestFixture se koristi kako bi ukazao da klasa sadrži test metodu. Prilikom izvršavanja aplikacije za testiranje, aplikacija skenira kod kako bi pronašla test klase. Naredni kod ilustruje korišcenje ovog atributa.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;
     [TestFixture]
public class SomeTests{}
}
```

Klase koje koriste atribut TestFixture moraju imati public konstruktor koji nema ulaznih atributa i vraca void.

Atribut Test

Atribut Test se koristi kako bi ukazao da aplikacija koja pokrece test treba da izvrši metodu koja je oznacena ovim parametrom. Potrebno je da ova metoda bude javna, nema povratnu vrednost i ne zahteva ulazne parametre. Metoda koja ne zadovoljava ove uslove nece biti prikazana u Test Runner GUI i nece biti izvršena prilikom pokretanja testa.

```
namespace UnitTestingExamples
{
   using System;
   using NUnit.Framework;
      [TestFixture]
    public class SomeTests
   {
      [Test]
       public void TestOne()
       {
          // Do something...
       }
    }
}
```

Atributi SetUp & Teardown

Prilikom sastavljanja testova, često se nailazi na slučajeve u kojima je pre i nakon svakog testa potrebno pozvati veliki broj funkcija. Ovim funkcijama se najcešce kreiraju zavisni objekti (npr. konekcija ka bazi podataka).

Jedno rešenje ovog problema je kreiranje privatne metode koja se poziva iz svih test metoda. Drugo rešenje je korišcenje atributa Setup and Teardown. Ovi atributi govore da ce se funkcija izvršiti pre (SetUp) ili nakon (Teardown) svake test metode u okviru klase koja je oznacena

atributom Test Fixture.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;
   [TestFixture]
  public class SomeTests
  {
     private int _someValue;
      [SetUp]
     public void Setup()
     {
       _someValue = 5;
     }
      [TearDown]
    public void TearDown()
    {
       _someValue = 0;
    }
     [Test]
    public void TestOne()
     {
     // Do something...
      }
  }
}
```

Atribut ExpectedException

Atribut ExpectedException se koristi kada je potrebno proveriti da li program vraća grešku određenog tipa.

Alternativa korišcenju ovog atributa bilo bi kreiranje try..catch bloka u okviru kojeg se postavlja logicka vrednost određene promenjive.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;
  [TestFixture]
   public class SomeTests
   {
     [Test]
     [ExpectedException(typeof(InvalidOperationException))]
     public void TestOne()
     {
       // Do something that throws an InvalidOperationException
      }
    }
}
```

Test ove metode će biti uspešan samo ako program vraca grešku tipa InvalidOperationException.
U situacijama kada program vraca više od jednog tipa greške potrebno je koristiti poslednju grešku

u steku. Test može testirati samo jednu stvar. Druga specificnost je da atribut nema informaciju o nasledivanju. Ako program vraća grešku ciji tip nasleduje klasu InvalidOperationException, test nece biti uspešan.

Atribut Ignore

Atribut Ignore koristi se u slučajevima kada nije potrebno testirati određenu test metodu. Ovaj atribut se koristi kada je potrebno privremeno skloniti test.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;
   [TestFixture]
  public class SomeTests
  {
    [Test]
    [Ignore("We're skipping this one for now.")]
    public void TestOne()
    {
       // Do something...
     }
   }
}
```

Klasa NUnit Assertion

NUnit osim atributima koji se koriste kako bi identifikovali testove u kodu, ima i klasu Assertion. Ova klasa sadrži static metode koje se mogu koristiti za testiranje pretpostavki odn. uporedivanje dobijenih i očekivanih rezultata.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;
   [TestFixture]
  public class SomeTests
  {
    [Test]
    public void TestOne()
    {
      int i = 4;
      Assertion.AssertEquals( 4, i );
     }
   }
}
```

Izvršavanje testova

NUnit dolazi sa dve razlicite aplikacije za pokretanje testova: Windows GUI aplikacijom i konzolnom XML aplikacijom.
Za pokretanje GUI aplikacije neophodno je navesti lokaciju na kojoj se nalaze skup unit testova odredenog asemblija. Skup testova je biblioteka klasa ili izvršnih programa koja sadrži atribut Test Fixtures. Prilikom pokretanja testova u aplikaciji se može grafički videti hijerarhija test klasa i odgovarajucih test metoda koji pripadaju odgovarajucoj klasi.

Za pokretanje svih testova potrebno je odabrati opciju Run. Ako je potrebno testirati samo metode odgovarajuce klase Test Fixture ili samo jednu test metodut, potrebno je dvaput kliknuti na odgovarajucu klasu/metodu.

Ako je potrebno automatizovati proces testiranja tako da se dobije izlaz koji se može postaviti na web sajt ili neko drugo mesto gde mogu pristupiti razvojni tim, menadžment ili korisnici, tada GUI aplikacija nije najbolje rešenje. Konzolna aplikacija NUnit 2.0 uzima lokaciju bibiloteke klasa kao argument komandne linije i kao rezultat vraca XML. Odgovarajucim XSLT transformacijama moguce je konvertovati XML u HTML ili neki drugi format.