embarcadero
reseller partner

Primož Gabrijelčič
http://primoz.gabrijelcic.org

MARAND
Napredna računalniška hiša

Kazalo

Vsi programi, omenjeni v tem dokumentu, so na voljo na naslovu http://17slon.com/EA/EA-UnitTesting.zip.

Uvod

Vsi programerji imamo radi dobro preizkušene programe. Če ne drugega, zato, ker potem lažje spimo, saj vemo (ali vsaj upravičeno upamo), da nas ponoči ne bo zbudil paničen klic. Malo manj radi pa pišemo teste, torej kodo, ki naše programe preizkuša. Danes vam bi radi dokazali, da je preizkušanje enot (angl. Unit Testing) koncept, ki ga ni težko vpeljati, če je koda – in, priznamo, to je lahko velik *če* – primerno organizirana. Vseeno pa mislimo, da se je vredno malo potruditi in program pripraviti na preizkušanje, saj bodo rezultati pomagali na več koncih – program bo lažje testirati, teste boste lažje napisali, pa še program bo na koncu lepše organiziran in ga bo zato lažje popravljati in spreminjati.

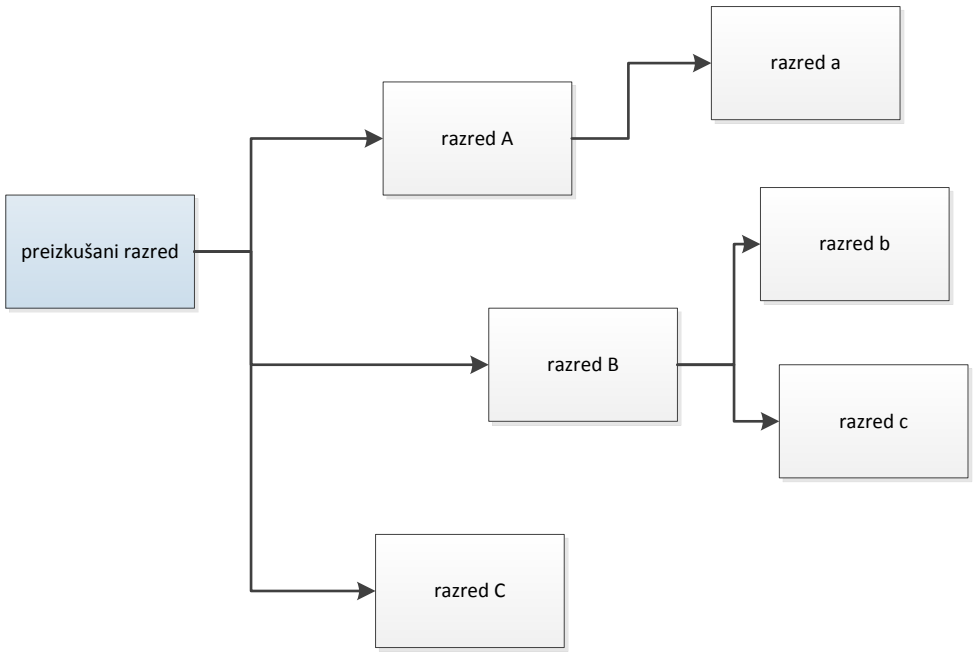
Prav pri spreminjanju obstoječe kode se pokaže prava moč samodejnega preizkušanja – najprej poženete teste in se prepričate, da je s kodo vse v redu, nato se lotite spreminjanja programa. Vsake toliko, ko ste prepričani, da je koda v dobrem, delujočem stanju, poženete teste in preverite, ali imate prav, ali pa bo treba morda spremenjeni del programa popraviti.

Poznamo več načinov preizkušanja: preizkušanje enot, integracijsko preizkušanje, regresijsko preizkušanje, preizkušanje cele aplikacije, pa še kakšen bi se našel. Danes se bomo ukvarjali predvsem s prvim, malo pa bomo zašli tudi na ostala področja. A o tem več kasneje.

Preizkušanje enot

Tema današnje delavnice je predvsem preizkušanje enot (»unit testing«). Takoj moramo opozoriti, da beseda »unit« v tem izrazu nima istega pomena, kakor v programskem jeziku Object Pascal. Ko govorimo o preizkušanju, nam »unit«, oziroma »enota«, pomeni majhen del programa, ki ga lahko samostojno preizkušamo. Običajno – oziroma vsaj pri lepo strukturiranem programu – nam enota predstavlja razred (class).

Največja težava pri preizkušanju samostojnih enot je, da so razredi v programu medsebojno odvisni. Posamezen razred je le redko tako samostojen, da ga lahko testiramo, ne da bi pri tem v igro pripeljali druge razrede.



Večinoma nas to moti, saj obnašanja teh razredov ne moremo predvideti (oziroma se ne moremo zanašati na to, da so dobro preizkušeni in brez napak), poleg tega pa raba dodatnih razredov lahko vpliva na zunanjo stanje (denimo na bazo podatkov), česar si pri testiranju enot prav gotovo ne želimo. Zato dodatne razrede, od katerih je testirani razred odvisen, zamenjamo z lažnimi (*mock, stub*). Več o tem smo zapisali v poglavju *Lažni razredi*.

Če program razširimo z dobro napisanimi preizkusi, nam to prinese več prednosti.

1. Odpornost na napake. Če preizkuse pridno poganjamo, bodo zaznali večino napak, ki bi jih v kodo vpeljali z nerodnim programiranjem.
2. Olajšano predelovanje (refactor). Pravzaprav se ta točka pokriva s prvo – kadar program predelujemo, nam dobro napisani preizkusi zaznajo večino napak, ki bi jih s predelavo dodali v kodo.
3. Testiranje robnih pogojev. V večini programov lahko enostavno preizkusimo tiste dele kode, ki so običajno v rabi. Težko pa preizkusimo kodo, ki se izvede le ob napakah ali čudnih vhodnih podatkih. S preizkušanjem enot lahko takšne dele kode dobro preizkusimo.

4. Dokumentacija. Preizkusi enot dobro dokumentirajo rabo razredov, ki jih preizkušamo, zato dodatna dokumentacija dostikrat niti ni potrebna.
5. Dobra zasnova programov. Koda, ki jo je enostavno preizkušati, ima malo medsebojnih odvisnosti in je zato lažja za razumevanje in spreminjanje.

Zagovorniki preizkušanja enot trdijo, da s pisanjem preizkusov prihranite čas, ki bi ga sicer porabili za razhroščevanje v celi življenjski dobi kode.

Kdaj naj bi preizkuse pravzaprav poganjali? Vsekakor med samim programiranjem, kar je tudi osnova principa TDD, o katerem bomo spregovorili nekaj besed v nadaljevanju. V večini podjetij, ki uporabljajo tak način dela, zahtevajo tudi, da programer požene preizkuse za spremenjeno kodo, preden jo pošlje v sistem za vodenje verzij (version control system, na primer Subversion ali Git). Primerno je tudi, da preizkuse vključimo v prevajalni strežnik (build server), če uporabljamo sistem stalne integracije (continuous integration), vsekakor pa jih moramo pognati kot del postopka, ki izdela končno verzijo programa (release).

TDD

Preiskušanje enot je centralni del načina programiranja, ki ga imenujemo *Test Driven Development*, oziroma na kratko TDD. Pri tem načinu obrnemo vrstni red in napišemo preizkus pred kodo, ki bi jo radi preizkusili. Ta preizkus seveda ne uspe (saj kode, ki naj bi jo preizkusil, še ni), zato v naslednjem koraku napišemo še ustrezen del programa. Nato preizkus še enkrat poženemo in preverimo, če je uspel. Tako nadaljujemo, dokler del programa, ki ga razvijamo, ni dokončan.

Razvojni proces TDD se tako vrti v enostavni zanki:

```
repeat
  WriteATest();
  WatchItFail();
  WriteTheCode();
  WatchTestPass();
until false;
```

Kako pisati dobre preizkuse

Pisanje dobrih preizkusov je umetnost, ki se jo najlažje naučimo v praksi. Vseeno pa lahko zapišemo nekaj vodil.

- Dober preizkus preizkuša čim manjši kos funkcionalnosti.
- Dober preizkus vsebuje le en test pravilnosti (*ali je dobljeni rezultat pravilen?*).
- Vedno raje napišemo še en preizkus, kot da bi v obstoječi preizkus dodajali nova preverjanja pravilnosti.
- Če v preizkusu nastopata stavka *if* ali *case*, je to običajno znak, da je treba preizkus razbiti na več manjših preizkusov.
- Preizkusi naj bodo napisani po vzorcu: pripravi preizkus, pokliči preizkušano kodo, preveri rezultat.
- Najprej vedno preizkusimo primere, ki bodo v rabi najpogostejši, šele nato se posvetimo robnim primerom.
- Vedno preverimo vse primere, ki v kodi sprožijo pričakovano izjemo (exception).
- Poskusimo preveriti vse možne poti skozi preizkušano kodo. Pri tem nam lahko pomaga odprtokodno orodje *Delphi Code Coverage* (povezavo najdete v Virih).
- Preizkusi naj bodo neodvisni en od drugega in od vrstnega reda izvajanja.

Zapomnite si tudi maksimo: »Če je preizkušanje težko izpeljati, nekaj delate narobe.« V večini primerov gre za slabo zasnovo programa (prepleteni razredi, koda, ki izvaja več nepovezanih akcij), ki zaplete preizkušanje.

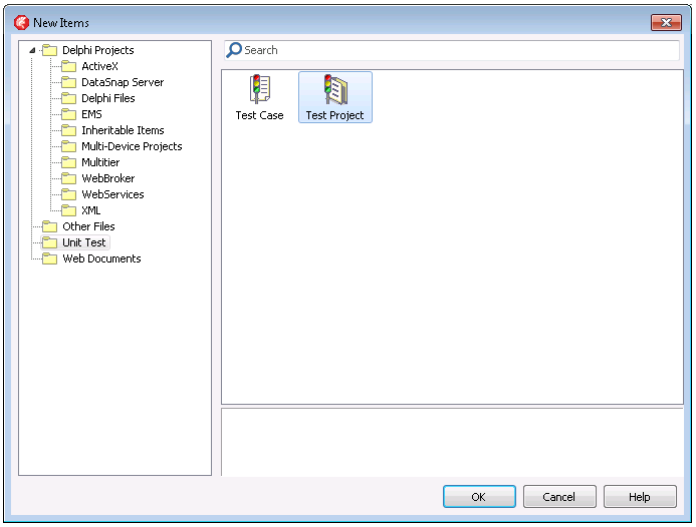
Orodja

Preizkušanje enot bi pravzaprav lahko zasnovali »iz nič«, na osnovi lastne kode, a je običajno lažje uporabiti eno od obstoječih ogrodij za preizkušanje. V Delphiju običajno uporabljamo DUnit, DUnit2 ali novejši DUnitX, omenili pa bomo tudi dodatek TestInsight, ki ogrodju DUnitX izredno izboljša uporabnost in ga naredi za ta trenutek najboljše preizkusno ogrodje.

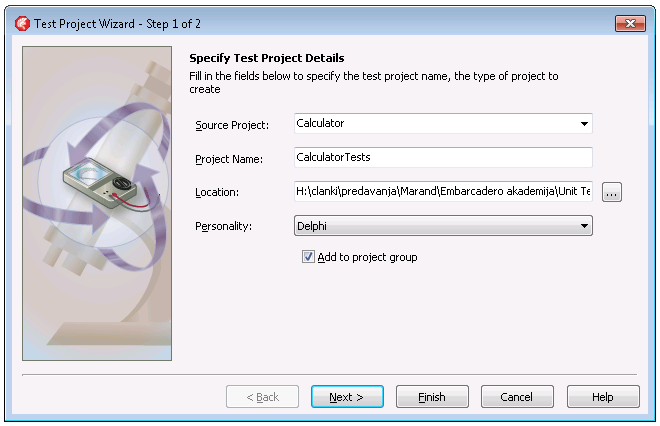
DUnit

DUnit je starosta orodij tega tipa za Delphi. Čeprav gre za staro in ne preveč posodabljano ogrodje, ima pred konkurenco veliko prednost – na vaš računalnik se bo namestil skupaj z Delphijem (če te opcije med namestitvijo niste ugasnili).

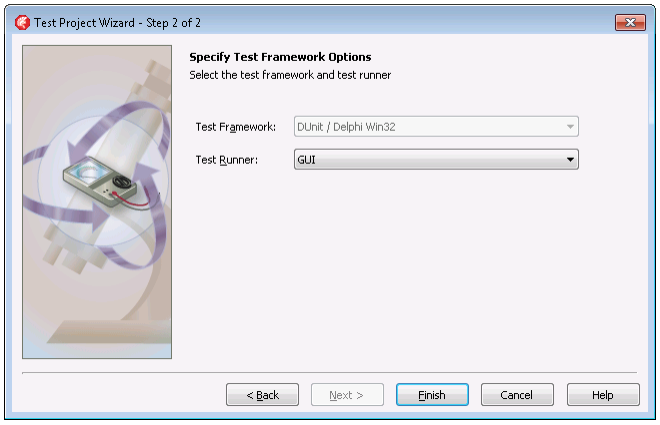
Preizkušanja z DUnitom se najlažje lotimo tako, da odpremo projekt, ki vsebuje *enoto*, ki bi jo radi preizkusili, ter v menuju izberemo *File, New, Other, Unit Test, Test Project.*



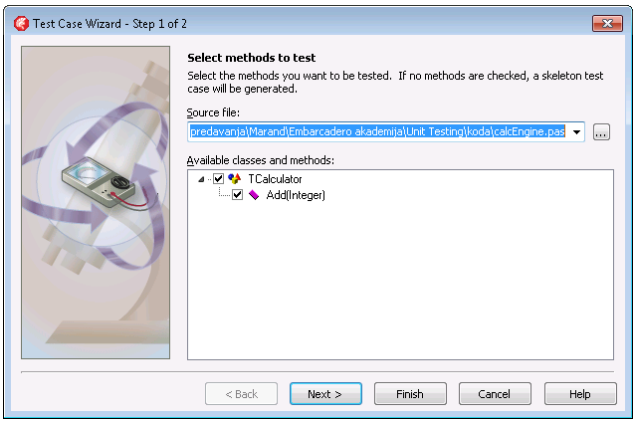
Določimo ime projekta in njegovo lokacijo. Priporočamo, da preizkusni projekt dodate v projektno skupino skupaj z originalnim programom.



Nato določite še okolje, v katerem se bo izvajal testni program. Na voljo imate dve okolji - grafično (GUI) in tekstovno (Console).



Nastal bo prazen projekt, ki še ni zelo uporaben. Vanj moramo dodati še enega ali več *primerov* (Test Case). Načeloma za vsako enoto, ki jo želimo preizkušati, naredimo en testni primer. Najlažji način je, da v urejevalniku odpremo kodo, ki vsebuje razred, ki bi ga radi preizkusili, ter iz menuja izberemo *File, New, Other, Unit Test, Test Case.* Čarovnik bo analiziral kodo ter prikazal razrede in metode, ki jih lahko testiramo (se pravi razrede, ki so definirani v *interface* delu ter metode, ki imajo vidljivost vsaj *public*).



Ko kliknemo *Finish*, čarovnik zgradi novo enoto (tokrat ta izraz uporabljamo v pomenu Object Pascala), ki vsebuje razred, v katerega dodajamo preizkuse.

```
type
  // Test methods for class TCalculator
  TestTCalculator = class(TTestCase)
  strict private
    FCalculator: TCalculator;
  public
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestAdd;
  end;
```

Ta razred je registriran v ogrodje v inicializacijski kodi enote.

```
initialization
  // Register any test cases with the test runner
  RegisterTest(TestTCalculator.Suite);
end.
```

Kadar preizkuse dodajate sami, brez čarovnika, nikakor ne smete pozabiti na to registracijo, sicer ogrodje vaših preizkusov ne bo zaznalo.

Vsaka metoda z vidljivostjo *published* (v našem primeru metoda *TestAdd*, ki jo je dodal že čarovnik) predstavlja en preizkus. Za vsak preizkus izvede ogrodje DUnit naslednji postopek:

- Naredi nov primerek razreda TestTCalculator.
- Požene metodo SetUp, v kateri si pripravimo okolje za preizkušanje.
- Požene preizkusno metodo (na primer *TestAdd*).
- Če metoda *uspe* (o tem več kasneje), jo označi kot *uspešno*. Če metoda javi napako, jo označi kot *neuspešno*.
- Požene metodo TearDown, v kateri počistimo preizkusno okolje.
- Uniči primerek razreda, ki ga je ustvaril v prvem okolju.
- Na ustrezen način (grafično, konzola) prikaže rezultat preizkusa.

Koda, ki jo pripravi čarovnik, je enostavna in ne vsebuje pravega preizkusa. Metoda TestAdd sicer pokliče metodo, ki jo želimo preizkusiti, a ne preveri rezultata.

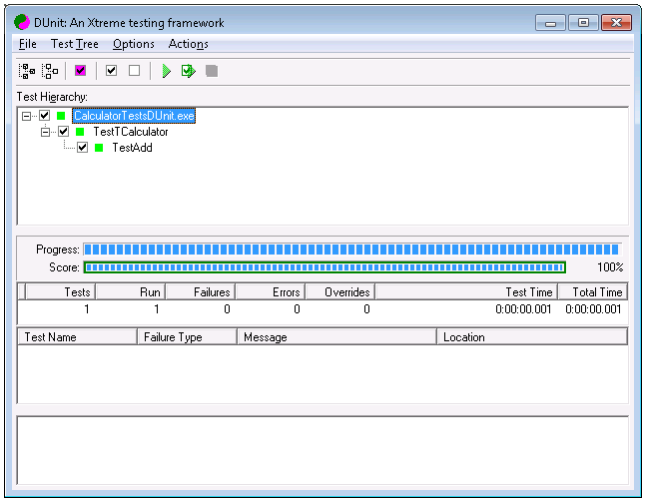
```
procedure TestTCalculator.SetUp;
begin
  FCalculator := TCalculator.Create;
end;

procedure TestTCalculator.TearDown;
begin
  FCalculator.Free;
  FCalculator := nil;
end;

procedure TestTCalculator.TestAdd;
var
  ReturnValue: Integer;
  b: Integer;
  a: Integer;
begin
  // TODO: Setup method call parameters
  ReturnValue := FCalculator.Add(a, b);
  // TODO: Validate method results
end;
```

Slednjo lahko z majhno spremembo (označeno spodaj) spremenimo v pravo preizkusno metodo.

```
procedure TestTCalculator.TestAdd;
var
  ReturnValue: Integer;
  b: Integer;
  a: Integer;
begin
  a := 2; b := 2;
  ReturnValue := FCalculator.Add(a, b);
  CheckEquals(4, ReturnValue);
end;
```

Nastaviti moramo torej začetne podatke, poklicati preizkušano kodo, ter na koncu preveriti vrnjeno vrednost. Metoda CheckEquals (del razreda TTestCase, iz katerega izhaja naš testni razred TestTCalculator) preveri, ali sta parametra enaka, ter ustrezno nastavi stanje preizkusa (uspešen/neuspešen).

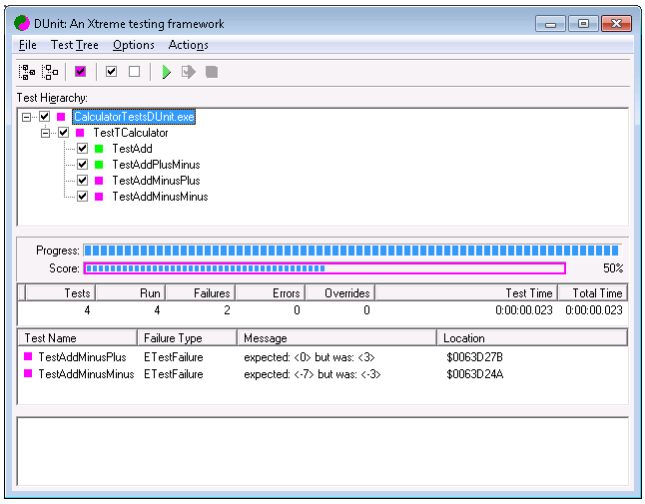
Ko tak program poženemo, se znajdemo pred grafičnim okoljem, v katerem lahko izberemo preizkuse, ki bi jih radi pognali. Trenutno imamo le enega, zato se s tem ne ukvarjamo, temveč pritisnemo F9 (Run) ali kliknemo zeleno puščico v orodjarni, da poženemo preizkus. Ker uspe, dobimo zelen (uspešen) status.



Na podoben način lahko dodamo še več preizkusov, recimo take, ki uporabljajo tudi negativna števila, pa take, ki dajo rezultat 0 (kar lahko na nek način pojmujemo kot robni primer) itd.

```
procedure TestTCalculator.TestAddMinusMinus;
begin
  CheckEquals(-7, FCalculator.Add(-4, -3));
end;

procedure TestTCalculator.TestAddMinusPlus;
begin
  CheckEquals(0, FCalculator.Add(-3, 3));
end;

procedure TestTCalculator.TestAddPlusMinus;
begin
  CheckEquals(0, FCalculator.Add(3, -3));
end;
```

Z nekaj sreče bo rezultat še vedno zelen, kadar pa kateri od preizkusov ne uspe, se obarva rdeče.



V takem primeru pogledamo, zakaj preizkus ni uspel, popravimo kodo (ali morda preizkus – tudi tam se lahko zmotimo), ter ponovno poženemo program ter ponovimo preizkušanje.

TTestCase

Razred TTestCase (pravzaprav njegov prednik TAbstractTest) nam nudi množico podprogramov, s katerimi lahko preverjamo pravilnost rezultatov. Prepoznamo jih po tem, da se njihovo ime začne s *Check*. Nekateri so na voljo v različnih variantah parametrov (overload). Skrajšan seznam metod, ki jih lahko uporabite, si lahko ogledate na naslednji strani.

```
procedure Check(condition: Boolean; msg: string = '');
procedure CheckTrue(condition: Boolean; msg: string = '');
procedure CheckFalse(condition: Boolean; msg: string = '');
procedure CheckEquals(expected, actual: extended; msg: string = ''); overload;
procedure CheckEquals(expected, actual: extended; delta: extended; msg: string =
'');  overload;
procedure CheckEquals(expected, actual: integer; msg: string = ''); overload;
procedure CheckEquals(expected, actual: Cardinal; msg: string = ''); overload;
procedure CheckEquals(expected, actual: int64; msg: string = ''); overload;
procedure CheckEquals(expected, actual: uint64; msg: string = ''); overload;
procedure CheckEquals(expected, actual: string; msg: string = ''); overload;
procedure CheckEquals(expected, actual: RawByteString; msg: string = ''); overload;
procedure CheckEqualsMem(expected, actual: pointer; size:longword; msg:string='');
procedure CheckEquals(expected, actual: Boolean; msg: string = ''); overload;
procedure CheckEqualsBin(expected, actual: longword; msg: string = ''; digits:
integer=32);
procedure CheckEqualsHex(expected, actual: longword; msg: string = ''; digits:
integer=8);
procedure CheckEquals(expected, actual: TCharArray; msg: string = ''); overload;
procedure CheckNotEquals(expected, actual: integer; msg: string = ''); overload;
procedure CheckNotEquals(expected, actual: Cardinal; msg: string = ''); overload;
procedure CheckNotEquals(expected, actual: int64; msg: string = ''); overload;
procedure CheckNotEquals(expected: extended; actual: extended; delta: extended = 0;
msg: string = ''); overload;
procedure CheckNotEquals(expected, actual: string; msg: string = ''); overload;
procedure CheckNotEqualsMem(expected, actual: pointer; size:longword;
msg:string='');
procedure CheckNotEquals(expected, actual: Boolean; msg: string = ''); overload;
procedure CheckNotEqualsBin(expected, actual: longword; msg: string = ''; digits:
integer=32);
procedure CheckNotEqualsHex(expected, actual: longword; msg: string = ''; digits:
integer=8);
procedure CheckNotEquals(expected, actual: TCharArray; msg: string = ''); overload;
procedure CheckNotNull(obj :IUnknown; msg: string = ''); overload;
procedure CheckNull(obj: IUnknown; msg: string = ''); overload;
procedure CheckSame(expected, actual: IUnknown; msg: string = ''); overload;
procedure CheckSame(expected, actual: TObject; msg: string = ''); overload;
procedure CheckNotNull(obj: TObject; msg: string = ''); overload;
procedure CheckNull(obj: TObject; msg: string = ''); overload;
procedure CheckException(AMethod: TTestMethod; AExceptionClass: TClass; msg: string
= '');
procedure CheckEquals(expected, actual: TClass; msg: string = ''); overload;
procedure CheckInherits(expected, actual: TClass; msg: string = ''); overload;
procedure CheckIs(AObject: TObject; AClass: TClass; msg: string = ''); overload;
```

Preizkušanje izjem

Če želimo preizkusiti vejo kode, ki sproži izjemo (exception), uporabimo lastnost ExpectedException in ji nastavimo razred izjeme, ki naj bi jo preizkušena koda sprožila.

```
procedure TestTTimerPlan.TestItemExceptionHigh;
var
  value: Integer;
begin
  FTimerPlan.Parse('1 2 3');
  ExpectedException := Exception;
  value := FTimerPlan[3];
end;
```

V tem primeru ne potrebujemo klica ene od metod *Check*. Če bo dostop do *FTimerPlan[3]* sprožil izjemo Exception, bo preizkus uspel, sicer pa ne.

DUnit2

DUnit2 je odprtokodni projekt, ki je nameraval oživiti zastareli DUnit, a je tudi na njemu aktivnost že pred leti zamrla. Vseeno pa je v to ogrodje dodano nekaj uporabnih lastnosti.

DUnit2 je večinoma združljiv z ogrodjem DUnit in definira enake razrede ter metode (in seveda dodaja nove). Zato lahko testni program zgradite kar s čarovnikom za DUnit, ki ga vsebuje Delphi, nato pa popravite pot (search path), kjer Delphi išče izvorno kodo.

Za razliko od ogrodja DUnit se tu konstruktor in destruktor preizkusnega razreda (potomca TTestCase) poženeta le enkrat. Za vsak preizkus pa se še vedno pokličeta metodi SetUp in Teardown. Obnašanje bi lahko ponazorili z naslednjo psevdokodo:

```
testCase := TMyTestCase.Create;
try
  for i := FirstTest to LastTest do begin
    testCase.Setup;
    try
      RunTestCase(testCase, i);
    finally testCase.Teardown; end;
  end;
finally testCase.Free; end;
```

Orodje za poganjanje zna testirati tudi »puščanje« pomnilnika (po potrebi lahko to preizkušanje izklopimo).

Status preizkusov lahko izvozimo v XML.

Dodali so obsežno ogrodje, s katerim lahko preizkušate obnašanje celega programa (klikate gumbe, vpisujete besedila v vnosna polja ...).

DUnitX

Povsem sveži projekt DUnitX se preizkušanja loteva drugače. Preizkusni razred izhaja iz poljubnega drugega razreda, testne primere opišemo z atributi, namesto CheckXXXX pa uporabljamo Assert.XXXX.

Testiranje metode TCalculator.Add bi v ogrodju DUnitX napisali tako:

```
  [TestFixture]
  TestTCalculator = class(TObject)
  strict private
    FCalculator: TCalculator;
  public
    [Setup]
    procedure SetUp;
    [TearDown]
    procedure TearDown;
    [Test]
    [TestCase('A','1,2,3')]
    [TestCase('B','3,-3,0')]
    [TestCase('C','-3,3,0')]
    [TestCase('D','-4,-3,-7')]
    procedure TestAdd(const AValue1, AValue2, AResult: Integer);
  end;
```

Atribut [TestFixture] določi testni razred. Še vedno pa ga moramo registrirati v inicializacijski sekciji.

```
initialization
  TDUnitX.RegisterTestFixture(TestTCalculator);
end.
```

Atributa [Setup] in [TearDown] določata metodi za pripravo in čiščenje preizkusa. Same preizkusne metode pa opišemo z atributom [Test]. Ni več potrebno, da bi bile te metode *published*.

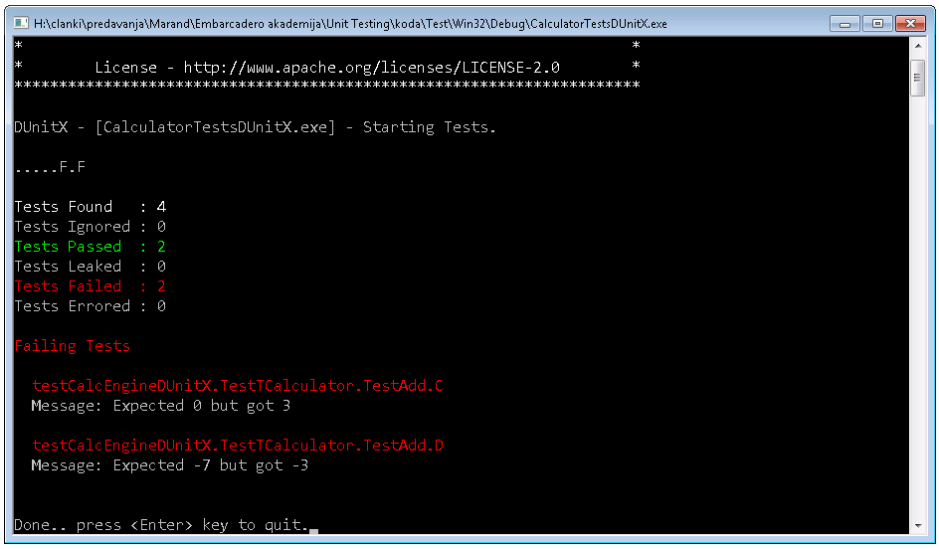
Metodam lahko dodamo tudi testne primere (atribut [TestCase]), kar lahko močno skrajša čas programiranja. Podamo jih v tekstovni obliki, ogrodje pa jih pretvori v obliko, ki jo pričakuje testna metoda (v našem primeru v tri celoštevilske vrednosti).

Preizkusno metodo TestAdd bi z ogrodjem DUnitX napisali tako:

```
procedure TestTCalculator.TestAdd(const AValue1, AValue2, AResult:
Integer);
begin
  Assert.AreEqual(AResult, FCalculator.Add(AValue1, AValue2));
end;
```

Pri izdelavi testnega programa si lahko pomagamo s čarovnikom (*File, New, Other, Delphi Projects, DUnitX Project*), testne enote in razrede pa moramo izdelati sami.

Za poganjanje je na voljo le konzolno okolje.



Lepše in bolj uporabno okolje za poganjanje dobimo z dodatkom TestInsight, ki ga opisujemo v nadaljevanju.

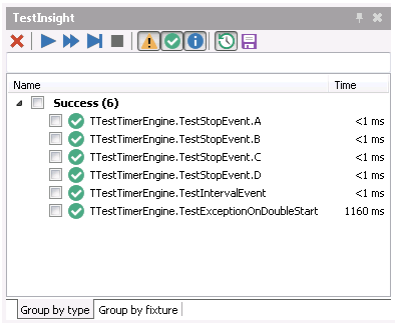
Preizkušanje izjem

Pričakovane izjeme v ogrodju DUnitX lovimo s klicem metode *Assert.WillRaise*, ki pričakuje kot prvi parameter proceduro brez parametrov. Običajno na tem mestu uporabimo anonimno proceduro, ki kliče kodo, ki bi jo radi testirali. Kot drugi parameter navedemo razred izjeme, ki jo pričakujemo.

```
procedure TTestTimerEngine.TestExceptionOnDoubleStart;
begin
  FEngine.Plan.Parse('1');
  FEngine.Start;
  Assert.WillRaise(
    procedure
    begin
      FEngine.Start;
    end,
    Exception
  );
end;
```

TestInsight

Povsem svež dodatek ogrodju DUnitX je razširitev TestInsight, ki Delphi razširi z oknom TestInsight.



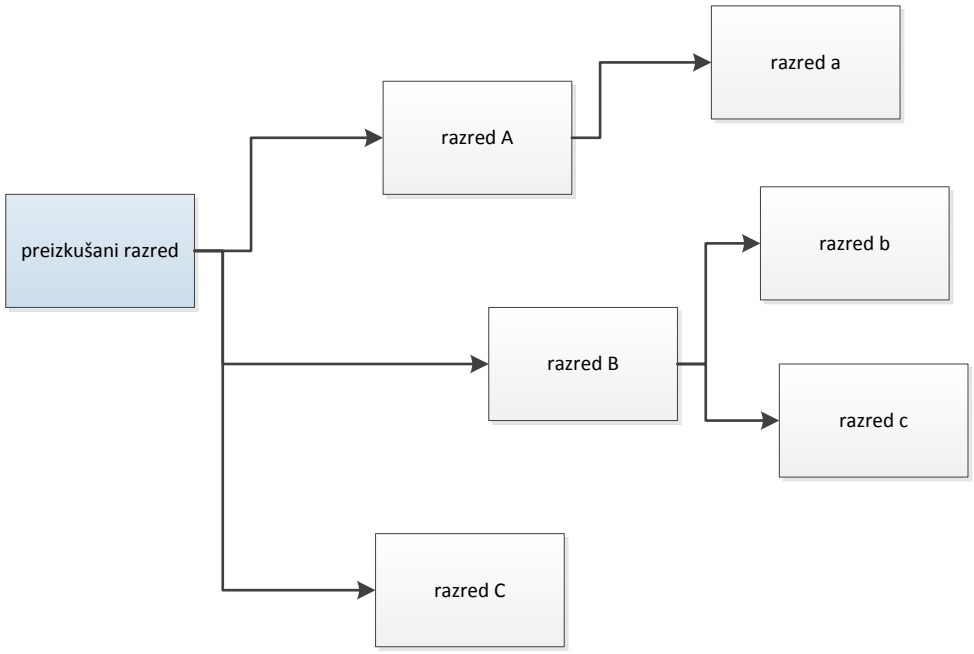
V tem oknu vidimo rezultate preizkusov, lahko jih na enostaven način poženemo, najlepše pa je, da lahko nastavimo, da se preizkusi poženejo samodejno, čim smo nekaj časa neaktivni. Po vsaki spremembi kode bo razširitev malo počakala, nato poskusila prevesti program in če bo prevajalnik uspešno opravil nalogo, se bodo preizkusi pognali.

V praksi to pomeni, da se lahko osredotočimo na pripravo preizkusov in testnih primerov, TestInsight pa jih bo samodejno in brez našega posredovanja poganjal in prikazoval rezultate.

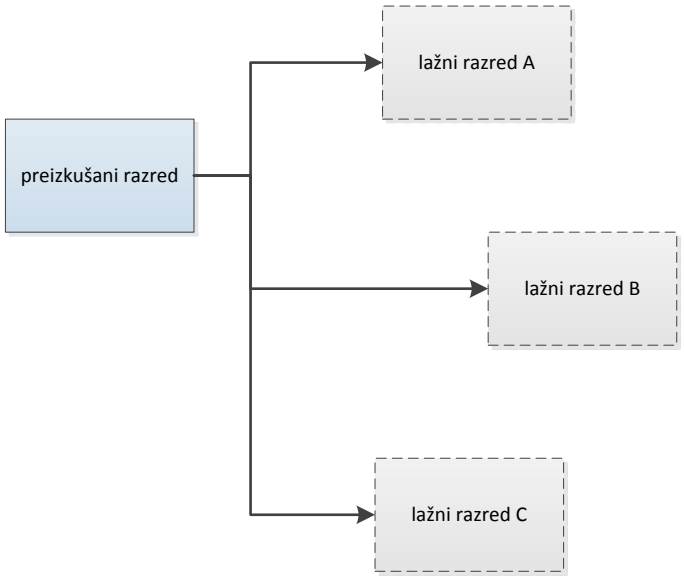
TestInsight zna sodelovati tudi z drugima dvema ogrodjema za preverjanje enot (DUnit, DUnit2).

Lažni razredi

Na začetku smo omenili, da je največji problem preizkušanja notranja zgradba programov. Bolj ali manj vsak razred v programu je odvisen od drugih razredov, ti spet od drugih in tako dalje. Običajno so nekateri od teh razredov povezani z zunanjim svetom (datoteke, socketi, podatkovne baze), kar nam oteži pripravo testnih primerov.



Pri preizkušanju enot ta problem običajno zaobidemo, tako da prave razrede nadomestimo z *lažnimi* – z razredi, ki se na zunaj obnašajo kot pravi, vsebujejo pa le malo ali nič programske kode. Seveda taki lažni razredi tudi niso odvisni od drugih razredov.



Dependency Injection

Pri tem nam pomaga programski pristop, imenovan *dependency injection*, ki je del širšega principa *inversion of control*. Za kaj pravzaprav gre?

Pri »običajnem« programiranju razred zgradi primerke vseh odvisnih razredov, ki jih potrebuje za svoje delo. Razred bi si tako na primer zgradil primerek razreda za pisanje v dnevnik (log) ter ga uporabljal med svojim delom. Kadar je razred napisan na ta način, težko posežemo v njegovo obnašanje ter mu namesto dnevniškega razreda podtaknemo razred, ki se obnaša na enak način, le da sporočila meče stran, namesto da bi jih zapisoval v dnevnik.

Zato po principu *inversion of control* logiko programa postavimo na glavo (jo »obrnemo«) ter primerek razreda za zapisovanje v dnevnik zgradimo zunaj našega razred, ter razredu le nastavimo lastnost, ki kaže na dnevnik. Temu določanju »od zunaj« pravimo tudi *dependency injection*, oziroma »vstavljanje odvisnosti«, saj primerke razredov, od katerih je naš razred odvisen, vstavimo vanj »od zunaj«.

Definicija takšnega razreda bi se lahko na primer glasila tako:

```
type
  ILogger = interface
    procedure Log(const msg: string);
  end;

  TTestedClass = class
  strict private
    FLogger: ILogger;
  public
    property Logger: ILogger read FLogger write FLogger;
  end;
```

Definiramo vmesnik, ki ga mora implementirati razred za zapisovanje v dnevnik (ILogger). Naš razred (TTestedClass) nato le izpostavi lastnost tega tipa, ter se ne ubada z njeno inicializacijo. Koda, ki bo kreirala instanco razreda TTestedClass, ji bo tudi nastavila lastnost Logger na pravo vrednost.

V testnem programu lahko nato izdelamo razred (*lažni* razred), ki implementira vmesnik ILogger, njegova metoda Log pa ne počne nič, nato pa ga uporabimo za testiranje. A v resnici si lahko z odprtokodnim ogrodjem Delphi Mocks to delo še bistveno olajšamo.

Delphi Mocks

Projekt Delphi Mocks je nastal z namenom olajšanega pisanja preizkusov, v katerih potrebujemo lažne primerke razredov. Podpira dva različna tipa lažnih razredov – TStub le implementira nek vmesnik, vse njegove metode pa so prazne (ne naredijo nič), TMock pa zna tudi preverjati, ali so bile njegove metode klicane na pravi način ter v primeru nepravilne rabe javi napako (izjemo), ki jo testno ogrodje prikaže programerju.

TStub

Denimo, da testiramo metodo DoSomething našega razreda, ki uporablja lastnost Logger, kot smo jo definirali zgoraj.

```
function TTestedClass.DoSomething(const msg: string): string;
begin
  Logger.Log(msg);
  Result := msg + '!';
end;
```

Če v preizkusnem primeru razred le zgradimo, ter pokličemo DoSomething, bo zgornja koda javila napako *Access Violation*, ker lastnost Logger ne bo nastavljena (enaka bo *nil*).

```
procedure TMockDemoTest.SetUp;
begin
  FTested := TTestedClass.Create;
end;

procedure TMockDemoTest.TestStub;
begin
  CheckEqualsString('Hello!', FTested.DoSomething('Hello'));
end;
```

V testni metodi TestStub zato definiramo spremenljivko tipa TStub<ILogger>, ki bo samodejno izdelala »prazen« primerek tega vmesnika.

```
procedure TMockDemoTest.TestStub;
var
  logger: TStub<ILogger>;
begin
  logger := TStub<ILogger>.Create;
  FTested.Logger := logger;
  CheckEqualsString('Hello!', FTested.DoSomething('Hello'));
end;
```

V nekaj vrsticah lahko tako definiramo prazen razred, ki implementira poljubno kompleksen vmesnik, ter ga uporabimo za preizkušanje drugega razreda.

TMock

Še bolj zanimiva je raba lažnih razredov TMock. Tem lahko tudi nastavimo, da preverjajo obnašanje preizkušane kode.

Recimo, da bi radi preverili, ali se ob klicu DoSomething natanko enkrat pokliče Logger.Log. Z uporabo TMock bi to dosegli na zelo enostaven način:

```
procedure TMockDemoTest.TestMock;
var
  logger: TMock<ILogger>;
begin
  logger := TMock<ILogger>.Create;
  logger.Setup.Expect.Once('Log');
  FTested.Logger := logger;
  CheckEqualsString('Hello!', FTested.DoSomething('Hello'));
  logger.Verify;
end;
```

Preverimo lahko celo, da se metoda Log pokliče s točno določenim parametrom:

```
procedure TMockDemoTest.TestMock;
var
  logger: TMock<ILogger>;
begin
  logger := TMock<ILogger>.Create;
  logger.Setup.Expect.Once.When.Log('Hello');
  FTested.Logger := logger;
  CheckEqualsString('Hello!', FTested.DoSomething('Hello'));
  logger.Verify;
end;
```

Metoda Expect vsebuje nekaj metod, s katerim na enostaven način določimo, kolikokrat naj bi se med testiranjem poklicala metoda vmesnika, ki ga oponašamo, vsako pa lahko uporabimo na dva načina – tako da samo preverimo, ali se je poklicala, ali pa tako, da preverimo tudi parametre.

```
  IExpect<T> = interface
    ['{8B9919F1-99AB-4526-AD90-4493E9343083}']
    function Once : IWhen<T>;overload;
    procedure Once(const AMethodName : string);overload;

    function Never : IWhen<T>;overload;
    procedure Never(const AMethodName : string);overload;

    function AtLeastOnce : IWhen<T>;overload;
    procedure AtLeastOnce(const AMethodName : string);overload;

    function AtLeast(const times : Cardinal) : IWhen<T>;overload;
    procedure AtLeast(const AMethodName : string;
      const times : Cardinal);overload;

    function AtMost(const times : Cardinal) : IWhen<T>;overload;
    procedure AtMost(const AMethodName : string;
      const times : Cardinal);overload;

    function Between(const a,b : Cardinal) : IWhen<T>;overload;
    procedure Between(const AMethodName : string;
      const a,b : Cardinal);overload;

    function Exactly(const times : Cardinal) : IWhen<T>;overload;
    procedure Exactly(const AMethodName : string;
      const times : Cardinal);overload;

    function Before(const AMethodName : string) : IWhen<T>;overload;
    procedure Before(const AMethodName : string;
      const ABeforeMethodName : string);overload;

    function After(const AMethodName : string) : IWhen<T>;overload;
    procedure After(const AMethodName : string;
      const AAfterMethodName : string);overload;
  end;
```

Delphi Mocks je odlična knjižnica, zahteva pa, da je koda, ki jo želimo preizkušati, primerno pripravljena – torej tako, da podpira vstavljanje odvisnosti.

Različni nivoji preizkušanja

Za konec omenimo še različne načine preizkušanja programov, ki jih utegnete srečati v praksi.

Preizkušanje enot smo že omenili. Bistveno za ta nivo preizkušanja je preizkušanje *neodvisnih* koščkov kode.

Integracijsko preizkušanje je podobno, le da pri njem preizkušamo kodo, ki je odvisna od druge (realne) kode oziroma od zunanjih dejavnikov (baza podatkov). Tudi takšnega preizkušanja se lahko lotimo z omenjenimi orodji, le da bomo več časa porabili za pripravo testnega okolja.

Regresijsko preizkušanje se ubada predvsem s problemom »vračanja« napak, ki smo jih že enkrat odpravili, v kodo. Če ne gre za napake uporabniškega vmesnika (za te potrebujemo aplikacijsko preizkušanje), se je regresijskega preizkušanja najbolje lotiti kar na način preizkušanja enot. Uporabimo zelo enostavno pravilo – vsakič, ko uporabnik javi napako, in jo uspemo ponoviti, naprej napišemo preizkus, ki to napako ponovi, ter jo šele zatem odpravimo. Preizkus seveda pustimo v paketu preizkusov, ki jih redno poganjamo.

Aplikacijsko preizkušanje preizkuša program na nivoju uporabniškega vmesnika (klikanje po menujih, vnosnih poljih, gumbih, itd). Če uporabljate ogrodje VCL, lahko uporabite praktično katerokoli testno okolje za Windows (zanimiv je morda TestComplete), lahko pa tudi zgradite test kar znotraj programa z ogrodjem DUnit2. Če uporabljate ogrodje FireMonkey, pa ste zaenkrat obsojeni na ročno klikanje po programih.

Viri

Unit Testing

http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Unit_Tests

Unit Testing Overview

http://docwiki.embarcadero.com/RADStudio/XE7/en/Unit_Testing_Overview

How to Write Good Unit Tests

https://developer.salesforce.com/page/How_to_Write_Good_Unit_Tests

Test Driven Development

Test Driven Development in Delphi: The Basics

http://www.codeproject.com/Articles/508680/TestplusDrivenplusDevelopmentplusinplusDelphi-apl

DUnit

http://dunit.sourceforge.net/

OpenCTF

https://mikejustin.wordpress.com/2008/06/22/openctf/

https://sourceforge.net/projects/openctf/

DUnit2

http://members.optusnet.com.au/mcnabp/index.html

http://members.optusnet.com.au/mcnabp/Projects/DUnit2/DUnit2Description.html

http://dunit2.sourceforge.net/

DUnitX

https://www.finalbuilder.com/resources/blogs/postid/697/introducing-dunitx

https://github.com/VSoftTechnologies/DUnitX

https://www.finalbuilder.com/resources/blogs/postid/697/introducing-dunitx

http://www.finalbuilder.com/Resources/Blogs/PostId/702/dunitx-has-a-wizard

https://plus.google.com/communities/110602661860791972403

TestInsight

http://delphisorcery.blogspot.com/2015/02/testinsight-unit-testing-like-pro.html

Mocks

Introducing Delphi Mocks

https://www.finalbuilder.com/resources/blogs/postid/417/introducing-delphi-mocks

Delphi Mocks: The Basics

http://www.nickhodges.com/post/Delphi-Mocks-The-Basics.aspx

Mocking Multiple Interfaces

https://www.finalbuilder.com/resources/blogs/postid/716/mocking-multiple-interfaces-delphi-mocks

Mocks Aren't Stubs

http://martinfowler.com/articles/mocksArentStubs.html

Inversion of Control

Inversion of Control

http://en.wikipedia.org/wiki/Inversion_of_control

Dependency Injection

http://en.wikipedia.org/wiki/Dependency_injection

TestComplete

http://smartbear.com/product/testcomplete/overview/

Delphi Code Coverage

https://code.google.com/p/delphi-code-coverage/

The Delphi Unit Test project

http://www.nickhodges.com/post/The-Delphi-Unit-Test-Project.aspx

https://bitbucket.org/NickHodges/delphi-unit-tests

Video

Delphi Unit Testing

https://www.youtube.com/watch?v=nm_yq9jYckc

Unit Testing in Delphi

https://www.youtube.com/watch?v=xUUC15RbiaQ