Poglavlje 10

Obrada postojećeg programskog koda i testiranje jedinica

U Delphiju 2005, pored mnogih novih osobina, postoje dve nove tehnologije koje su tokom poslednjih nekoliko godina postale značajne za programere. Jedna od tih tehnologija je preispitivanje postojećeg programskog koda (engl. refactoring), dok je druga inkluzija dva okruženja za testiranje jedinica. Iako ove dve tehnologije nisu tesno povezane, one se koriste zajedno i deo su istog pogleda na programiranje koji možemo opisati pomoću dva termina: agilne metodologije i ekstremno programranje (XP).

Pošto su to u programiranju relativno novi koncepti, naročito za Delphi programere, ja sam, pored toga što ću objasniti njihovo praktično korišćenje u Delphijevom IDE-u, napisao teorijski deo. Takođe, napisao sam kratak uvod u agilni i XP svet, uglavnom dajući linkove za relevantne izvore. Nemojte mnogo brinuti zbog teorijskog dela. U većem delu teksta ovog poglavlja objašnjavam Delphijeve osobine i pokazujem praktične primere.

Osnovni razlog zbog kojeg ove dve tehnologije nisam objasnio u Poglavlju 1, u kojem sam objašnjavao IDE, jeste to što ipak morate imati malo više znanja o programskom jeziku i bibliotekama nego što sam u Poglavlju 1 pretpostavio da imate.

Pored opšte diskusije agilnih metodogloija i ekstremnog programiranja, u ovom poglavlju se posebno bavim preispitivanjem postojećeg programskog koda u Delphiju, objašanjavam ostale pomagače za IDE i razmatram testiranje jedinica u Delphiju pomoću jedinica DUnit i NUnit.

Više od alata za brzo pavljenje programa

Delphi je nastao kao alat za brzo pravljenje aplikacija (Rapid Application Development - RAD), odnosno kao vizuelno okriženje, ali istovremeno u potpunosti podržava objektno-orijentisano programiranje. Možete u potpunosti zaboraviti vizuelne dizajnere i pisati programski kod i koristiti sve OOP pristupe. To nije uvek najbrži način za pravljenje aplikacije, ali će struktura programa svakako biti fleksibilnija i robusnija bez obzira na izmene.

OOP pristupi programiranju su tokom poslednjih nekoliko godina sve popularniji, ali je nedavno bilo dosta rasprava o nekim pristupima programiranju koji se često zasnivaju na OOP programiranju ili su mu slični. Najvažniji pristup koji je sličan OOP programiranju jeste korišćenje šablona, značajan industrijski napredak koji je zahvatio i Delphi. U nekoliko časopisa i web sajtova ćete naći tekstove o ovoj temi, ali ja ih, na žalost, u ovoj knjizi neću razmatrati.

Pretpostavke tradicionalnih modela programiranja obuhvataju značajnu ulogu analiziranja i strukture u programiranju, i potrebu za veoma formalnim postupkom programiranja. Ekstremno programiranje i agilne metodologije menjaju ulogu programera, smeštajući ih u središte postupka pravljenja softvera. Agilne metodlogije otklanjaju obavljanje zamornih postupaka.

Postoje brojne knjige posvećene toj temi, a rezime na jednoj stranici knjige svakako nije dovoljan, te ću ja pokušati da izdvojim neke elemente i da iskažem svoje utiske.

U narednim odeljcima ću objasniti tehnike koje naizgled nisu u međusobnoj vezi, ali za koje smatram da ih treba pomenuti, čak i ako nemate dovoljno vremena da ih sve detaljno obradite. Pružiću vam reference na web sajtove i knjige u kojima ima više informacija.

Agilne metodologije

Ukoliko su dizajniranje i pisanje programskog koda dva važna koraka za pravljenje programa, onda postoji mnogo više koraka koji se tiču analiziranja i testiranja aplikacije. Na početku je pravljenje softvera bio kreativan posao. U poslednjoj dekadi i za veće projekte, formalni postupak programiranja i metodologija su postali norma. Međutim, ne samo da su programeri morali da se nose sa tim dodatnim formalnim koracima, već su metodolozi iznašli da su kontraproduktivne.

Zbog toga su se krajem 1990-ih godina počele pojavljivati nove metodologije (odnosno, postupci u pravljenju softvera). Mada postoji nekoliko različitih pristupa, uobičajeno je da nove metodologije daju prednost jednostavnosti (u svetu svakodnevnih promena zahteva) i korisnicima. Zbog toga se za takve metodologije sve više koristi termin agilne metodologije (engl. agile methodologies). Taj termin je postao veoma važan kada je grupa programera 2001. godine udružila napore i stvorila Agile Manifesto (pronaći ćete ga na web sajtu `http://agilemanifesto.org`).

Jedan od najboljih izvora o agilnim metodologijama i predlozima, sa desetinama linkova ka knjigama i web stranicama, jeste dokument Martina Faulera (Martin Fowler), koji se nalazi na adresi:

`www.martinfowler.com/articles/newMethodology.html.`

Ja ne nameravam da se detaljno bavim raznim agilnim metodologijama, jer one nisu tema ove knjige. U narednom odeljku ću ukratko opisati jednu od agilnih metodologija, jer ta metodologija promoviše preispitivanje postojećeg programskog koda i testiranje jedinica.

Ekstremno programiranje

Ideja ekstremnog programiranja (XP, nema veze sa oznakom Windows XP, koja je nastala od Windows eXperience) jeste da programeri imaju istaknutu ulogu, jer oni obično imaju male radionice, a veoma retko velike kompanije. Mada se XP može posmatrati kao skup najboljih postupaka koji vam mogu pomoći prilikom biranja modela za pravljenje softvera, pobornici ekstremnog programiranja ga smatraju složenom metodologijom, to jest, osmišljenim postupkom za pravljenje softvera.

XP su Kent Bek (Kent Beck) i Vord Kaningem (Ward Cunningham) formalno predstavili 1986. godine kao alternativan način za pravljenje softvera. XP se ne odnosi samo na programiranje (pisanje programskog koda) već na ceo postupak pravljenja softvera, od analiza zahteva do testiranja.

Neke od osnovnih pretpostavki su osposobljavanje programera, uključivanje korisnika u tim menadžera i programera koji pravi softver, i fleksibilnost prilikom izmena zahteva. XP je okrenut komunikaciji, jednostavnosti, razmeni poruka i ohrabrivanju. XP projekti se zasnivaju na čestim poboljšanjima aplikacije. Programeri rade u parovima i od samog početka testiraju programski kod. Postoji mnogo XP tehnika, uključujući:

Zahtevi korisnika Zahteve pišu korisnici i predstavljaju zamenu za dokumente sa formalnim zahtevima, jer sadrže kratak opis onoga što aplikacije treba da ispuni. Ti zahtevi su napisani tako da ne sadrže tehničke termine.

Planiranje verzije Planiranje se obavlja tako što se procenjuje vreme koje je potrebno za realizaciju svakog zahteva korisnika i kasnije se dele na iteracione planove. XP je zapravo iterativan postupak programiranja, a zahtev korisnika treba podeliti u najmanje desetak iteracija. Sveukupni napredak u pravljenju aplikacije se meri praćenjem iteracija.

Male verzije Ove verzije se često prave i omogućavaju da ih korisnik isproba i da timu programera pruži značajne informacije.

Jednostavna struktura je bolja od složene strukture. Mada svi to znaju, neke metodologije favorizuju veliku, složenu strukturu pre nego što se počne sa pisanjem programskog koda. Ključ jednostavne strukture jeste da se u osnovni sistem ne ugrađuju nepotrebne mogućnosti.

CRC kartice (Class, Responsibilities i Collaboration kartice) se na objektno-orijentisan način koriste prilikom struktuiranja sistema. Svaka kartica predstavlja klasu, koja je opisana skupom zadataka i spiskom pridruženih klasa.

Nemilosrdno preispitivanje postojećeg programskog koda Znači da postojeći programski kod poboljšavate i prepravljate čak i kada pravilno radi (nasuprot pravilu da programski kod koji radi ne treba menjati), kako bi se unapredila sveukupna struktura, kako bi programski kod postao razumljiviji i kako bi se njime lakše upravljalo. (Preispitivanje postojećeg programskog koda ću opisati u posebnom odeljku u ovom poglavlju.)

Programiranje u parovima Ideja je da dva programera uvek pišu programski kod i da naizmenično koriste tastaturu. Par programera su zajedno autori programskog koda (još jedna od osnovnih ideja u ekstremnom programiranju), tako da projekat ne zavisi od jednog programera. Osim toga, nasuprot onome što se može pomisliti, na ovaj način se pisanje programskog koda može značajno ubrzati, naročito kada se piše složen programski kod.

Programiranje uslovljeno testiranjem označava testiranje jedinica (testiranje na nivou klasa/metoda), pisanje probnog programskog koda pre nego što se napiše programski kod aplikacije, jer je ovakvo testiranje robusnije i korisnije prilikom pisanja boljeg programskog koda (kasnije ću vam ovo detaljnije objasniti).

Testiranje podobnosti se zasniva na zahtevima korisnika i proverava da li su ti zahtevi potpuno i korektno implementirani. Testiranje je deo provere kvaliteta (engl. quality assurance - QA) i može ga obavljati poseban tim. Rezultati provere kvaliteta se objavljuju i ključni su za procenu napretka projekta.

Stalna integracija programskog koda koji pišu parovi programera znači da se programski kod često integriše, ali tako što se integriše jedan po jedan programski kod tima programera, a nakon svake integracije se testira celokupan programski kod.

Drugi važni elementi su standardi pisanja programskog koda (kako bi programeri lakše razmenjivali programski kod), optimizacija na kraju pisanja programskog koda (samo nakon provere brzine aplikacije), nema prekovremenog rada, kolektivnog vlasništva nad programskim kodom... međutim, ekstremno programiranje je mnogo više od ovoga što sam pomenuo.

Postoje mnogi tekstovi o ekstremnom programiranju, uključujući knjigu Kenta Beka "Extreme Programming Explained" (Addison-Wesley, 1999), i nedavno objavljenog drugog izdanja (Addison-Wesley, 2004). Uvod u ekstremno programiranje ćete naći na web sajtu `www.extremeprogramming.org` i na web sajtu `www.xprogramming.com`.

Neki od pobornika ekstremnog programiranja kažu da ne možete upotrebiti samo neke od tehnika, već morate upotrebiti sve tehnike ekstremnog programiranja kako biste uspeli da ostvarite projekat. Ja mislim da žele izbegavanje odgovornosti. Smatram da su tehnike ekstremnog programiranja interesantne, ali ne mislim da su najbolje, niti da se moraju doslovno primenjivati. Projekti koje pravi jedan programer i projekti za koje je angažovan veliki broj programera su veoma različiti. Takođe, alati za programiranje, ciljni operativni sistem, udaljenost između programera i mnogi drugi faktori se moraju uzeti u obzir prilikom izbora metodologije pravljenja softvera.

Pobornicima agilnih metodologija često smeta popularnost ekstremnog programiranja jer je umanjen značaj drugih interesantnih pristupa programiranju. To je delimično zbog svega što čini Kent Bek, ali najviše zbog suštine ekstremnog programranja. Ekstremno programiranje se vatreno promoviše, a pobornici i protivnici vode žestoke rasprave. Metodologije za pravljenje softvera su obično suvoparne.

Mnogo toga se može napisati o agilnim metodologijama i ekstremnom programiranju. Ja u ovoj knjizi pažnju želim da usmerim na specifične oblasti, na primer, testiranje i preispitivanje postojećeg programskog koda. Razlog je jednostavan: u Delphiju 2005 postoji direktna podrška, a ja sam smatrao da moram da objasnim te termine, kako biste mogli da razumete o čemu pišem.

Preispitivanje postojećeg programskog koda

Kada pišete nov programski kod, kako treba da se odnosite prema postojećem programskom kodu programa koji pravite? Kao što sam već rekao, u industriji softvera postoji tendencija da ne treba dirati ništa što pravilno radi. Međutim, kada u klasu dodajete nove metode, ne možete izbeću da s vremena na vreme menjate strukturu klase, i da, na kraju, izmenite odnose između različitih klasa.

Preispitivanje (engl. refactoring) programskog koda je u stvari obrada postojećeg programskog koda. Osnovna ideja je da programski kod preispitujete (proveravate) kada se promeni interna struktura jedne ili više klasa, a da se pri tom ne promeni ponašanje programa. Druga ideja je da čak i kada pravite velike izmene u programskom kodu, preispitujete postojeći programski kod, tako da svako preispitanje programskog koda ima ograničen obim. Vi želite da program pravilno radi ne samo kada ugradite ceo skup transformacija, već i nakon svake ugrađene transformacije.

Pored toga što opisuje pristup i što vam daje smernice za rad, preispitivanje programskog koda vam pomaže da tim izmenama programskog koda date smisleno ime, onako kako vam šabloni struktura pomažu da programskom kodu date ime. Preispitivanje programskog koda se obavlja kada se taj programski kod održava, a ne samo kada dobijete projekte ili programski kod koji su napisali drugi programeri. Naravno, preispitivanje programskog koda se ne može obaviti kada počnete da pišete program. Više informacija ćete pronaći u knjizi Marina Faulera (Martin Fowler) "Refactoring: Improving the Design of Existing Code" (Addison-Wesley, 1999).

Postoje razna preispitivanja programskog koda. Primeri preispitivanja programskog koda koji ne postoje u Delphiju su sakrivanje metoda, kada metod pretvarate u metod tipa private ukoliko se ne poziva izvan klase, očuvanje celih objekata, sugeriše da objekte prosleđujete kao parametar umesto da prosleđujete nekoliko njegovih polja, ili zamena izuzetaka testiranjima, što znači da treba ukloniti izuzetke koji se mogu inicirati pogrešnim parametrima metoda i zameniti ih običnim uslovnim iskazima. Najsvežiji spisak (koji pravi Martin Fauler) ćete pronaći na adresi:

```
www.refactoring.com/catalog/index.html
```

Kao što ćete uskoro videti, u Delphiju se primenjuju samo neku postupci za preispitivanje postojećeg programskog koda, sam termin označava programere koji nisu deo tima koji preispituje postojeći programski kod.

Testiranje jedinica

Testiranje jedinica je interesantna tehnika za proveru imeplentacije klase koju koriste programeri koji korste druge alate i programske jezike. Testiranje jedinica nije proveravanje kako će aplikacija izgledati već proveravanje pozivanja metoda.

Testiranje jedinica ima dva cilja. Testiranjem jedinica se proverava da li se programski kod koji ste napisali pravilno izvršava i da li će se pravilno izvršavati kada dodate nov programski kod ili kada izmenite postojeći (kada preispitujete programski kod). Međutim, testiranje jedinica se koristi kako bi se unapred detaljno definisali zahtevi, naročito kada prihvatite ideju ekstremnog programiranj da se jedinice testiraju pre nego što ih ugradite u aplikaciju.

Nemojte preterivati sa testiranjem, to jest, nemojte postavljati uslove za svaku mogućnost, već se usredsredite na osnovnu funkcionalnost i napravite testiranja za greške koje ste uočili prilikom pisanja programskog koda.

Ideja ekstremnog programiranja i ostalih agilnih metodologija jeste da prihvatite izmene, pa je zbog toga testiranje novog i postojećeg programskog koda veoma važno. Na samo da testiranje mora biti uspešno pre objavljivanja svake vezije aplikacije, već mora biti uspešno pre nego što nov programski kod ugradite u aplikaciju, tako da svaka nova verzija aplikacije uspešno prođe testiranje.

Kombinovanje preispitivanja programskog koda i testiranja jedinica

Treba da znate da se preispitivanje programskog koda i testiranje jedinica obavlja zajedno. Kada preispitujete programski kod koji su napisali drugi programeri morate biti sigurni da nećete promeniti ponašanje tog programskog koda već da ćete promeniti samo način na koji se efekat ostvaruje. Tome služi testiranje jedinica.

Ukoliko testiranje jedinica postojećeg programa obavljate pre nego što počnete sa menjanjem programskog koda, slobodno možete preispitivati programski kod jer ćete tako proveriti da li u postojeći programski kod unosite greške ili stare greške izlaze na površinu. Ukoliko u postojećem programu ne postoji podrška za testiranje, a planirate da preispitate programski kod (zato što vam pristiže nov programski kod), prvo treba da napravite jedinice kako biste proverili šta se pomoću programskog koda obavlja i kako biste razumeli njegovu strukturu. Tek kada uobiličite okruženje za testiranje treba da počnete sa proveravanjem programskog koda.

Ja neću koristiti preispitivanje programskog koda i testiranje jedinica dok u ovoj knjizi budeom objašnjavao kako ih u Delphiju možete koristiti, jer ću se držati jednostavnih primera.

Preispitivanje programskog koda u Delphiju 2005

Iako postoje dodaci koji obezbeđuju slične osobine, u Delphiju 2005 u IDE-u po prvi put postoji mogućnost preispitivanje programskog koda. Preispitivanje programskog koda u Borlandovom Developer Studiu se zasniva na kompletnoj analizi izvornog programskog koda koje se obavlja pomoću .NET tehnologije CodeDOM, pa zbog toga, ukoliko uklonite podršku za .NET, nećete moći da koristite tu osobinu.

Za preispitivanje programskog koda postoji poseban element u glavnom meniju i kontekstnom meniju editora. Element u glavnom meniju je Refactor, a u kontekstnom meniju Refactoring. Oba elementa menija imaju isti sadržaj koji zavisi od simbola ili programskog koda koji je označen u editoru. Kao što sam već rekao, u Delphiju se prilikom preispitivanja programskog koda ne koriste sve formalne definicije ove tehnike, već se koriste samo neki opšti pomagači, kao što je Declare Variable.

Ali ja suvuše idem unapred. Pre nego što opišem vrste preispitivanja programskog koda (stvarno i lažno) u Delphiju, moram da vam kažem da preispitivanje programskog koda ima specifičnu podršku

za Unicode skup znakova, što je zgodno jer se veliki broj operacija obavlja odjednom, recimo kada koristete Rename.

Deklarisanje promenljivih

U Delphiju postoje dve opcije za deklarisanje prilikom preispitivanja programskog koda. Pomoću jedne (koja se poziva upotrebom kombinacije tastera Ctrl+Shift+V) se deklarišu lokalne promenljive, pomoću druge opcije (koje se poziva upotrebom kombinacije tastera Ctrl+Shift+D) se deklarišu polja u tekućoj klasi. Ove dve operacije treba da vam omoguće pravljenje novih promenlivih, a da se pri tom ne menja pozicija postojećeg programskog koda, tako da treba samo da kucate. Sve što treba da uradite jeste da popunite (ukoliko vam predloženi sadržaj ne odgovra) i prihvatite okvir za dijalog.

Pored toga što se programski kod brže pravi, vaše razmišljanje se ne prekida pretraživanjem programskog koda. Ne morate prelaziti na početak metoda, dodavati odeljak var, ukoliko takav ne postoji (da biste dodali lokalni promenljivu) ili prelazili u metod klase (da biste deklarisali polje). Pažnju možete u potpunosti usmeriti na programski kod koji pišete.

Oba pomagača za novu promenljivu sugerišu tip podataka na osnovu konteksta, kao što je dodeljivanje sa desne strane, tip parametra metoda ili kontekst izraza.

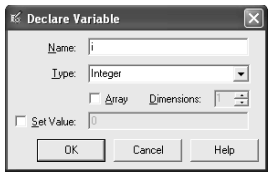
Na primer, deklarisanje promenljive dobro dođe prilikom pisanje for petlji, jer ne morate sami deklarisati brojač petlje. Sve što treba da uradite jeste da unesete:

```
for i := 0 to
```

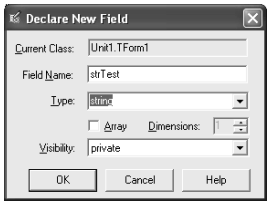
vratite se na simbol i aktivirate deklarisanje promenljive (pomoću tastaturne prečice, komande glavnog menija ili komande kontekstnog menija). Prikazaće se okvir za dijalog kakav vidite na slici 10.1. Upamtite da ukoliko ne napišete :=, onda IDE neće moći da predvidi vašu nameru i neće predložiti odgovarajući tip podataka.

Upamtite da ovu tehniku možete koristiti i za deklarisanje niza (ukoliko potvrdite odgovarajuće polje) i da promenljivoj možete dodeliti početnu vrednost. U tom slučaju će Delphi dodati programski red sa odgovarajućim iskazom dodeljivanja na početak metoda ili procedure.

Deklarisanje polja se obavlja na sličan način. Kada deklarišete polje u okviru za dijalog postoji još jedna opcija, opcija pomoću koje zadajete vidljivost promenljive tako što zadajete specifikator pristupa (uključujući nove specifikatore strict private i strict protected) (pogledajte sliku 10.2).



SLIKA 10.1
Okvir za dijalog Declare Variable



SLIKA 10.2
Okvir za dijalog Declare New Field

U tim okvirima za dijalog možete promeniti simbol tako što menjate tekst koji je označen u editoru. Ukoliko to učinite, onda će se, ako se simbol ne može koristiti, prikazati simbol za grešku. Ukoliko želite da vidite kako taj simbol izgleda, definišite identifikator koji nije dozvoljen tako što ćete u imenu napraviti razmak ili tako što ćete prvo uneti cifru. Međutim, ukoliko izmenite ime identifikatora, u deklaraciji će se koristiti novo ime, ali se označeni tekst u editoru neće promeniti; moraćete sami da ga promenite. Meni je to veoma čudno.

Declare Variable i Declare New Variable koji postoje u Delphiju nisu deo zvaničnih vrsta preispitivanja programskog koda. Međutim, mogu se koristiti kako bi se lakše implementirale standardne vrste preispitivanja programskog koda:

- Introduce Explaining Variable
- Consolidate Conditional Expression
- Push Down Field i Pull Up Field
- Move Field

Izdvajanje u resursni string

Druga vrsta preispitivanja programskog koda koja postoji u Delphiju 2005 je pojednostavljeni način za prebacivanje konstantnog stringa u resursni string. Resursni stringovi se ne kompajliraju u segment programskog koda aplikacije već se zapisuju u poseban resursni blok. Na taj način se štedi memorija, a resursni stringovi su korisni prilikom prevođenja aplikacije.

Prevođenje koje postoji u Delphiju izdvaja sve resursne stringove iz programa i omogućava vam da ih zapišete u DLL datoteku zajedno sa prevodiocem. Tokom izvršavanja aplikacije Delphi može odabrati jezik na osnovu podešavanja računara na kojem se aplikacija izvršava. Nešto slično se odigrava u okruženju .NET Framework. Postojanje posebnog skupa stringova, ili po jednog skupa stringova u svakoj jedinici, umesto postojanja konstantnih stringova u programskom kodu, omogućava slanje razumljivijih poruka korisniku jer se sve poruke nalaze na istom mestu na kojem se lako mogu proveriti.

Problem je to što je korišćenje resursnih stringova uvek bilo zamorno, jer morate kopirati string, preći sa mesta na kojem pravite izmene u prethodni blok ili drugu jedinicu, smestiti string na novo mesto, deklarisati resursni string i vratiti se na početnu poziciju i uneti ime stringa. Sav taj posao se sada može obaviti pomoću samo jedne komande koja se zadaje pomoću kobinacije tastera Ctrl+Shift+L.

To znači da ukoliko imate metod sličan sledećem:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  strTest: string;
begin
  strTest := 'Hello';
```

Onda možete preći na string (ne morate označavati ceo string, već je dovoljno da kursor pomerite unutar stringa) i aktivirati preisipitivanje pa će se prikazati okvir za dijalog koji vidite na slici 10.3 i (ukoliko prihvatite ponuđene izmene) dobiti:

```
resourcestring
  StrHello = 'Hello';
...
procedure TForm1.Button1Click(Sender: TObject);
var
  strTest: string;
begin
```

```
  strTest := StrHello;
end;
```

Sistem predlaže ime za resursni string tako što koristi tekst i prefiks "Str". Na žalost, ne postoji način da prefiks izmenite (na primer, da prefiks bude "str", što se meni više dopada).

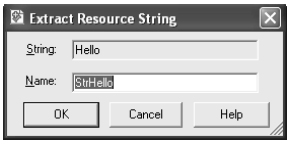
Ukoliko je string dugačak, Delphi će izdvojiti nekoliko prvih reči. Na primer, ukoliko u jednom redu unesete:

```
strTest := 'This is a very long string that won''t even fit in
  a line of code after I refactor it';
```

nakon pozivanja Extract Resource String i nakon što prihvatite predložene vrednosti, on postaje (drugi i treći red se nalaze u jednom redu u editoru):

```
resourcestring
  StrThisIsAVeryLong = 'This is a very long string that won''t
    even fit in a' +
  ' line of code after I refactor it';
```

Extract Resource String nije deo zvaničnog spiska, verovatno zato što u programskim jezicima Java i C++ postoje druga preispitivanja programskog koda.



SLIKA 10.3 *Okvir za dijalog Extract Resource String*

Promena imena

Promena imena (Rename refactoring, tastaturna prečica je Ctrl+Shift+E) je u Delphiju verovatno najbitnije preispitivanje programskog koda koje postoji u IDE-u i jedno je od zvaničnih preispitivanja. Promena imena se može koristiti za razne entitete: promenljive, procedure, tipove podataka (uključujući klase), polja i metode. Primer okvira za dijalog vidite na slici 10.4. Pošto u okviru za dijalog kliknete komandno dugme OK, ukoliko je polje View references potvrđeno, u prozoru Refactorings će se prikazati spisak izmena (to je panel koji je dokiran u dnu editora). Sve izmene možete proveriti, prihvatiti ih ili odbaciti (pogledajte sliku 10.5).

Promena imena je vrsta pronalaženja i zamene vrednosti i veoma je moćna tehnika jer može da odredi kada se isto simboličko ime odnosi na drugi simbol. Neka je u dve različite jedinice deklarisano ime tipa podataka: Delphi će promeniti sve reference entiteta za koji menjate ime, dok će ostale instance ostati nepromenjene.

Promena imena se ne odnosi samo na deklaracije i definicije klase ili metoda, već i na sve reference na njih u okviru tekućeg projekta. Možete promeniti ime simbola u celom projektu, ali Delphi možete koristiti za pojedinačne projekte u okviru grupe projekta. Drugim rečima, ukoliko se projekat poziva na asembli ili paket, promena imena će se odnositi i na paket u kojem je deklarisan simbol i projekat u kojem se koristi (sve to pomoću samo jedne operacije promene imena).

Još jedan primer razlike između operacija pronalaženja i zamene i operacije promene imena jeste da se preklopljenim metodima menja ime, tako da možete izbeći prekomerno preklapanje ili ga napraviti.

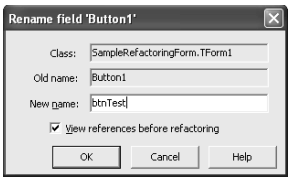
Slično, ukoliko u klasi roditelja promenite ime virtuelnog metoda, Delphi će promeniti ime metoda u izvedenim klasama.

Mada je promena imena ponekad nepredvidiva, ipak se može koristiti za simbole koji se nalaze u DFM datotekama (na primer, imenima klasa formulara, imena komponenata). Na primer, ukoliko promenite ime komponente, u prozoru Refactoring će se prikazati spisak izmena koji sadrži "VCL Designer Updates". Primer vidite na slici 10.5.

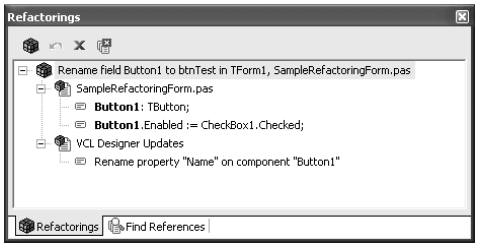
Ono što ne funkcioniše kako se očekuje jeste promena imena svojstva: metodi za zadavanje i čitanje vrednosti, ukoliko takvi postoje, neće biti automatski izmenjeni. Sami ćete morati da ih izmenite.

Postoji još jedna situacija u kojoj možete upotrebiti promenu imena: za konzistentnu upotrebu malih i velikih slova. Ukoliko se isti simbol često piše na različite načine, onda možete upotrebiti promenu imena kako biste to popravili. Međutim, pošto nećete moći da promenite ime simbola sa istim imenom osim ako se ne koriste drugačija slova, onda morate upotrebiti jedan od trikova. Prvi trik je da simbolu dva puta promenite ime, tako da nakon prvog menjanja imena sve reference imaju pogrešno ime, a nakon drugog ispravno ime. Drugi način je da u originalnoj definiciji koristite pogrešno napisano ime, i da zatim pozovete Rename (ili na bilo koje pojavljivanje istog simbola, jer se okviru za dijalog koristi simbol onako kako je zapisan).

SLIKA 10.4
Okvir za dijalog Rename field



SLIKA 10.5
Prozor Refactoring, posle promene imena vizuelne komponente



Izdvajanje metoda

Još jedno od zvaničnih preispitivanja programskog koda, verovatno ono koje proizvodi najviše programskog koda, jeste izdvajanje metoda (Extract Method, kombinacija tastera Ctrl+Shift+M). Ideja je da Extract Method isečak programskog koda pretvori u metod. To radite kada metod postaje previše složen da bi mogao da se shvati ili zato što morate da upotrebite deo programskog koda koji se nalazi u drugom metodu. Ono što je interesantno jeste da se promenljive koje se koriste u isečku programskog koda automatski prosleđuju kao parametri metodu koji se pravi. (Treba da znate da se ne postavlja pitanje vidljivosti metoda, jer je on svakako tipa private; kada koristite izdvajanje metoda onda sigurno ne želite da izmenite interfejs klase.)

U ovom slučaju u okviru za dijalog se vide buduće izmene (pogledajte sliku 10.6). U okviru za dijalog zadajete ime novog metoda. Na primer, telo petlje for, koju vidite u sledećem programskom kodu, se pretvara u metod koji je prikazan na slici:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  outStr: string;
begin
  for i := 1 to 10 do
  begin
    outStr := IntToStr (i);
    ListBox1.Items.Add (outStr);
  end;
end;
```

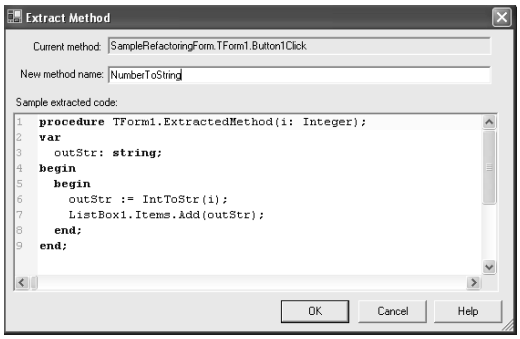
U ovom slučaju brojačka promenljiva i petlje for se kao parametar prosleđuju metodu, dok su ostale lokalne promenljive premeštene u novi metod, jer se više ne koriste u prvobitnom programskom kodu. Nakon izdvajanja metoda se dobija sledeći programski kod:

```
procedure TForm1.NumberToString(i: Integer);
var
  outStr: string;
begin
  begin
    outStr := IntToStr(i);
    ListBox1.Items.Add(outStr);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to 10 do
    NumberToString(i);
end;
```

Pored nepotrebnog bloka begin end u novom metodu, dobijeni programski kod je sasvim korektan, naročito kada u obzir uzmete da za sličnu operaciju koju sami obavljate treba prilično vremena.

Svaki isečak programskog koda ne možete pretvoriti u metod. U programskom kodu ne smeju postojati greške i ne može sadržati iskaze width, lokalne procedure i izvedene pozive. Takođe, izdvajanjem metoda se generiše procedura, nikada funkcija, iako se po potrebi mogu koristiti var parametri. Izdvajanjem metoda se u nekim slučajevima sugerišu pravilni parametri metoda.



SLIKA 10.6
Okvir za dijalog Extract Method u kojem se vidi izdvajanje metoda

Na primer, ukoliko imate sledeći programski kod:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
  nTotal: Integer;
begin
  nTotal := 0;
  for i := 1 to 10 do
  begin
    nTotal := nTotal + i;
    ListBox1.Items.Add (IntToStr (i));
  end;
  ListBox1.Items.Add (IntToStr (nTotal));
end;
```

Izdvajanjem metoda će se napraviti parametar za metod, jer telo petlje for menja spoljašnju promenljivu:

```
procedure TForm1.AddToList(i: Integer; var nTotal: Integer);
begin
  nTotal := nTotal + I;
  ListBox1.Items.Add(IntToStr(i));
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
  nTotal: Integer;
begin
  nTotal := 0;
  for i := 1 to 10 do
    AddToList(i, nTotal);
  ListBox1.Items.Add (IntToStr (nTotal));
end;
```

Ukoliko u prethodnom programskom kodu pozovete funkciju `Inc` (`nTotal`, `I`) umesto dodeljivanja vrednosti, Delphi neće primetiti sporedan efekat za promenljivu `nTotal`, jer je `Inc` funkcija kompajlera koja izvodi trikove. Kada pozovete funkciju `Inc`, parametar `nTotal` se neće proslediti po referenci, pa će se dobiti rezultat koji ne odgovara rezultatu koji se dobija pomoću originalnog programskog koda. Međutim, takav problem se lako rešava.

Ponekad ćete morati sami da ispravljate konstantne parametre kako biste poboljšali performanse, kao što je slučaj sa stringovnim parametrima.

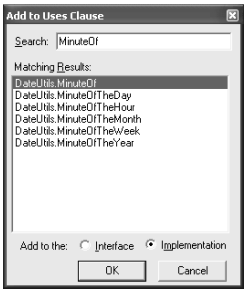
Pronalaženje jedinice

Poslednje preispitivanje programskog koda u Delphiju 2005 omogućava automatsko dodavanje jedinice u iskaze uses. Kada u programskom kodu programa postoji simbol koji nije deklarisan, razlog može biti to što je simbol deklarisan u jedinici koja nije navedena u iskazima uses.

Na primer, ukoliko pozivate funkciju `MinuteOf`, a da jedinicu DateUtils niste naveli u iskazima uses, onda možete iz menija Refactor (ili upotrebom kombinacije tastera Ctrl+Shift+A) pozvati pronalaženje jedinice (Find Unit). Prikazaće se spisak mogućih referenci (pogledajte sliku 10.7). U tom spisku birate funkciju jedinice (ili tip podataka), odlučujete da li da iskaz uses treba smestiti u implementacioni odeljak jedinice i automatski ga unosite. Pozitivno je to što nećete prekidati tok misli i nećete morati da se krećete kroz programski kod kako biste obavili ovu operaciju.

Iako je ova osobina veoma dobra, problem je što postoje ograničenja. Delphi koristi CodeDOM. To znači da jedinica koja vam je potrebna već mora biti referencirana u projektu ili grupi projekta. Pošto to uključuje indirektne reference (recimo, jedinice koje se koriste u VCL jedinicama), velika je šansa da Delphi ne može da pronađe jedinicu koja vam je potrebna, ali treba reći da postoje slučajevi kada se sve odvija kako treba. Pronalaženje jedinice u Delphiju za .NET, kada se na ceo imenovani prostor pozivate iz iskaza uses, radi bolje nego u Delphiju za Win32.

SLIKA 10.7
Okvir za dijalog Uses Unit



Pronalaženje referenci

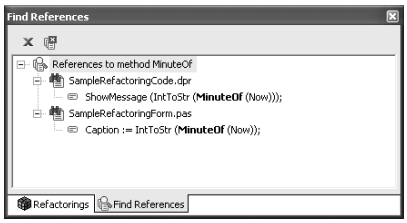
Još jedna nova osobina u Delphiju 2005, koja je tesno povezana sa preispitivanjem programskog koda, jer se zasniva na istoj arhitekturi, jeste mogućnost pronalaženja referenci na simbol programskog koda. To je osnova na kojoj je napravljena promena imena, ali je ponekad zgodno pronalaziti reference kako bi se nekim referencama promenilo ime (a ne svim).

U drugom odeljku menija Search i u podmeniju Find kontekstnog menija editora ćete pronaći tri nove komande:

Find References Komanda Find References vam omogućava da pronađete sve reference na zadati simbol (klasu, metod, funkciju, tip podataka, i tako dalje) u tekućem projektu. Na primer, za metod to obuhvata deklaraciju metoda u klasi, definiciju metoda i sva pozivanja metoda u programu. Rezultat pronalaženja referenci za funkciju biblioteke vidite na slici 10.8. U tom slučaju dobijate spisak lokacija izvornog programskog koda na kojima se funkcija poziva, a ne gde je definisana.

Find Local References Komanda Find Local References vam omogućava da pretražujete samo tekuću jedinicu.

Find Declaration Symbol Komanda Find Declaration Symbol vam omogućava da pronađete gde je zadati simbol deklarisan. Isti efekat postižete kada pritisnete taster Ctrl i kliknete simbol (pogledajte Poglavlje 1).



SLIKA 10.8
Prozor Find References nakon pretraživanja

Testiranje jedinica u Delphiju 2005

Ja sam vam već teoretski objasnio testiranje jedinica i kao deo ekstremnog programiranja ranije u ovom poglavlju kada sam objašnjavao postupak Test First, kada sam objasnio šta se podrazumeva pod testiranjem jedinica. Objasnio sam zašto su testiranje jedinica i preispitivanje programskog koda blisko povezane tehnike. Sada je vreme da pažnju posvetim testiranju jedinica u Delphiju 2005.

U Delphiju postoje dva različita okruženja za testiranje jedinica, jedno koje je namenjeno isključivo za Delphi i drugo koje je namenjeno isključivo za platformu .NET. Prvo okruženje je okruženje DUnit (adresa je `http://dunit.sourceforge.net`), i koristi se u Delphiju za platforme Win32 i .NET, a drugo je NUnit (adresa je `www.nunit.org`), i koristi se u platformi .NET za programske jezike Delphi i C#. Drugim rečima, u Delphijevim projektima za .NET možete odabrati jedno od okruženja; inače birate okruženje koje postoji.

NAPOMENA

Dok budete čitali ovu knjigu, trebalo bi da se pojavi verzija Delphija 2005 uz koju se dobija okruženje DUnit. Pošto se oba okruženja stalno unapređuju, posećujte oba web sajta kako biste unapređenja preuzeli. ■

Važna osobina u Delphiju, pored integracije okruženja za testiranje, jeste postojanje čarobnjaka pomoću kojeg se podešava test projekat i prave osnove metoda klasa. Pokušaću da vam u narednim odeljcima pokažem nekoliko primera, mada je veoma teško demonstrirati testiranje jedinica kada ne postoji (veliki) projekat. Treba da znate da sav programski kod nije deo projekta koji testirate (a to je ključna osobina testiranja jedinica u poređenju sa ostalim tehnikama). To je velika prednost, jer testiranje ne utiče na izvorni programski kod aplikacije, niti na izvršnu datoteku.

Korišćenje čarobanjaka je gotovo nemoguće za Test First pristup. Međutim, u praksi se pokazalo da možete napraviti lažnu klasu, napraviti testove za metode, i zatim pisati programski kod klase. Najzad, ta okruženja za testiranje jedinica se mogu koristiti za funkcionalna testiranja više klasa ili slojeva, a

mogu se koristiti za merenje efikasnosti, jer se meri vreme potrebno za obavljanje svakog testa. S druge strane, veoma je teško upotrebiti testiranje jedinica za testiranje korisničkog interfejsa. To je oblast za koju testiranje jedinica nije namenjeno.

Okruženje DUnit

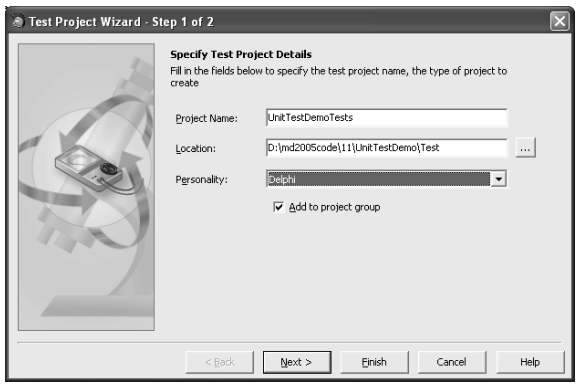
Razlog zbog kojeg više volim okruženje DUnit od okruženja NUnit je jednostavan: ja još uvek imam dosta Win32 programskog koda, a kompatibilnost programskog koda je važna za veliki broj .NET projekata. Kod vas svakako može biti drugačija situacija. Okruženje DUnit je napravio Huanko Anjez (Juanco Añez), kao deo dodatka JUnit za programski jezik Delphi, mada je tokom poslednjih godina u razvoj okruženja DUnit uključeno više programera. Za okruženje DUnit važi licenca MPL.

Da bih vam pokazao kako se koristi okruženje Dunit, ja sam napisao program UnitTestDemo. U glavnom projektu se nalazi sledeća klasa:

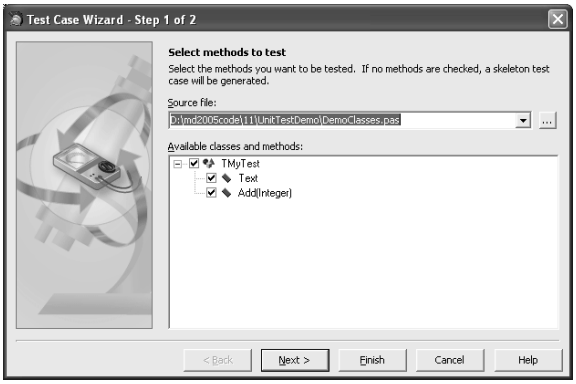
```
type
  TMyTest = class
  private
    FNumber: Integer;
    procedure SetNumber(const Value: Integer);
  public
    property Number: Integer read FNumber write SetNumber;
    function Text: string;
    procedure Add (n: Integer);
  end;
```

Pošto sam napisao ovu jedinicu i jednostavan formular u kojem se klasa koristi, ja sam zadao komandu File➥New➥Other (odnosno, možete upotrebiti odgovarajuće komandno dugme sa palete Tool Palette), kako bih otvorio okvir za dijalog New Items i na stranici Unit odabrao Test Project Wizard. Čarobnjak Test Project Wizard (pogledajte sliku 10.9) vam omogućava da zadate ime i lokaciju test projekta. U drugom okviru za dijalog čarobanjaka birate okruženje za testiranje (samo kada je u pitanju Delphijev projekat za .NET) i da odaberete tip testiranja (u grafičkom okruženju ili konzoli).

Pošto podesite test projekat, možete mu dodavati projekte. Možete odabrati Test Case Wizard (nalazi se na istoj stranici okvira za dijalog New Items ili na istoj karitici palete Tool Palette), i odgovarajući okvir za dijalog će vas voditi kroz nekoliko koraka i napraviće strukturu programskog koda. Čarobanjak Test Case Wizard mi za moj projekat omogućava da odaberem metode klase `TMytest` (pogledajte sliku 10.10).



SLIKA 10.9 *Čarobnjak Test Project Wizard koristite za pravljenje jedinice test projekta.*



SLIKA 10.10 *Čarobnjak Test Case Wizard vam omogućava da odaberete metode za koje želite da napišete testove.*

U sledećem okviru za dijalog čarobnjaka zadajete ime za test klasu i jedinicu gde želite da je zapišete. Rezultat je jedinica koja ima programski kod koji vidite u listingu 10.1 (izbačeni su neki komentari).

LISTING 10.1 Jedinica TestDemoClasses koju generiše Test Case Wizard

```
unit TestDemoClasses;

interface

uses
  TestFramework, DemoClasses;

type
  // Test methods for class TMyTest
  TestTMyTest = class(TTestCase)
  strict private
    FMyTest: TMyTest;
  public
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestText;
    procedure TestAdd;
  end;

implementation

procedure TestTMyTest.SetUp;
begin
  FMyTest := TMyTest.Create;
end;

procedure TestTMyTest.TearDown;
begin
```

```
  FMyTest.Free;
  FMyTest := nil;
end;

procedure TestTMyTest.TestText;
var
  ReturnValue: string;
begin
  ReturnValue := FMyTest.Text;
  // TODO: Validate method results
end;

procedure TestTMyTest.TestAdd;
var
  n: Integer;
begin
  // TODO: Setup method call parameters
  FMyTest.Add(n);
  // TODO: Validate method results
end;

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

Kao što vidite, programski kod za inicijalizaciju omogućava da okruženje zna za postojanje klase pozivanjem metoda `RegisterTest` za klasu `TTestSuite` koja se dobija pomoću osnovnog metoda `Suite`. Metod `Suite` je definisan u osnovnoj klasi `TTestCase`.

U test klasi je definisan objekat koji se inicijalizuje u metodu SetUp, a uklanja u metodu TearDown. Ova dva metoda se automatski pozivaju iz okruženja kada obavljate testove. Testiranje klase (odnosno izvršavanje paketa test klase, da budem precizniji) obuhvata izvršavanje svakog metoda tipa published, što predstavlja jedan test. Čarobnjak definiše po jedan test za svaki metod, ali vi možete napraviti koliko god želite testova (to jest, metoda tipa published).

U test metodu možete operisati sa ciljnim objektom, a zatim pozvati jedan od metoda za testiranje, koji obuhvata proveravanje vrednosti (`Check`, `CheckEquals`, `CheckNotEquals`, `CheckNull`, `CheckNotNull`, `CheckIs`...), proveravanje izuzetaka (`CheckException`) ili grešaka (`Fail`, `FailEquals`, `FailNotSame`...).

Na primer, evo jednostavnog testa za metod Add:

```
procedure TestTMyTest.TestAdd;
begin
  FMyTest.Number := 10;
  FMyTest.Add(5);
  CheckEquals(15, FMyTest.Number);
end;
```

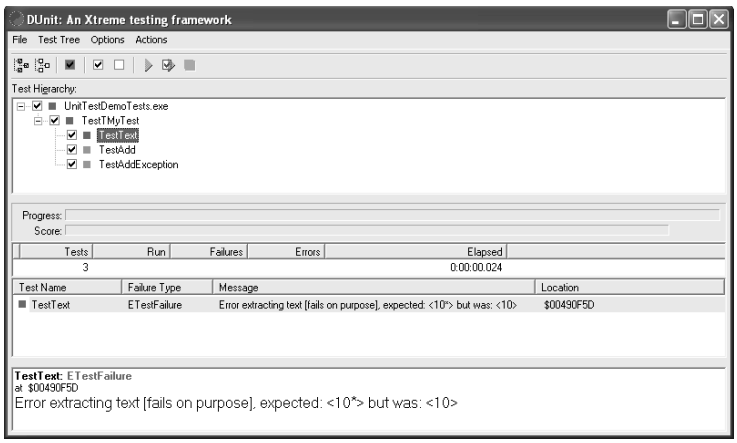
Pošto programski kod metoda prouzrokuje izuetak kada se kao parametar prosledi negativan broj, drugi test za metod Add bi bio proveravanje da li se u tom slučaju zaista poziva izuzetak. Da biste to postigli morate napraviti metod koji nije tipa published koji prouzrokuje izuzetak i proslediti ga metodu CheckException:

```
procedure TestTMyTest.TestAddException; // published test
begin
  FMyTest.Number := 10;
  CheckException (AddNegativeValue, Exception);
end;

procedure TestTMyTest.AddNegativeValue; // public helper method
begin
  FMyTest.Add(-5);
end;
```

Ukoliko test pokrenete u debageru, test će se zaustaviti kada se pozove izuzetak. Naravno, ukoliko koristite specijalan izuzetak, onda Delphijevom IDE-u možete naložiti da prekine izvršavanje, ili da testiranje nastavi izvan debagera.

Najzad, ja sam napisao test za `TestText` metod koji sasvim sigurno neće biti zadovoljen. Kada se napiše taj programski kod može se početi sa testiranjem programa. Prikazaće se grafički interfejs okruženja za testiranje (pogledajte sliku 10.11) pomoću kojeg pokrećete testove i u kojem dobijate rezultate.



SLIKA 10.11 *Korisnički interfejs glavnog prozora okruženja DUnit, kao i testovi koji su definisani u projektu UnitTestDemo*

Ukoliko u ciljnu klasu testiranja dodate nove metode, možete ponovo pozvati čarobnjaka Test Class Wizard kako biste napravili nove testove. Ukoliko odaberete metod za koji već postoji test, programski kod ovog testa se neće promeniti i neće se napraviti novi test.

Moram ipak da kažem da iako je čarobnjak od velike pomoći, pošto se naviknete na okruženje DUnit (ili drugo okruženje za testiranje), lakše je direktno pisati testove, pa ćete sve manje koristiti usluge čarobnajka.

NAPOMENA

Dodatak Update 2 za Delphi 2005 omogućava da čarobnjaka Test Case Wizard koristite za generisanje testova za interfejse i metode tipa protected. ■

Okruženje NUnit

U okruženju DUnit testirate projekte za platformu Win32 koji se mogu preneti na platformu .NET bez menjanja programskog koda. To je dobro ukoliko želite da napravite VCL aplikaciju, i njene testove, koja se može korisiti u platformi .NET i koja je kompatibilna sa platformom Win32. Kada ekskluzivno koristite platformu .NET onda treba da koristite okruženje NUnit. Okruženje NUnit je prvobitno bilo potpuna kopija JUnita, kao i većina okruženja za testiranje. Kasnije je ovo okruženje prepravljeno kako bi se iskoristile osobine .NET arhitekture, kao što su refleksija i atributi. Okruženje NUnit možete koristiti za svaki .NET programski jezik, uključujući programski jezik Delphi, naravno.

Za primer korišćenja okruženja NUnit i poređenja dva mehanizma za testiranje ja sam prethodni primer prebacio u platformu .NET i ponovo sam napravio test projekat koristeći okruženje NUnit. Rezultat tog rada je direktorijum NUnitDemo.

Da bih napravio nov projekat ja sam koristio ista dva čarobnjaka. Međutim, rezultat je drugačiji. Test projekat okruženja DUnit je izvršna datoteka, dok je test projekat okruženja NUnit biblioteka asemblija. Asembli se može izvršiti zadavanjem programa u interfejsu okruženja NUnit, koje se zove nunitgui.exe i instaliran je u direktorijumu bin okruženja NUnit. Čarobnjak Test Project omogućava automatsko podešavanje.

Test programski kod je prilično drugačiji jer se oslanja na refleksiju i atribute, a ne na registraciju i metode tipa published. Test klasa je označena atributom `TestFixture`, metodi `SetUp` i `TearDown` imaju specifične atribute, a svaki test metod je označen atributom `Test`. Dobijenu klasu (pre nego što sam u nju dodao programski kod) vidite u listingu 10.2. Taj programski kod možete uporediti sa programskim kodom iz listinga 10.1, u kojem se nalazi slična jedinica dobijena u okruženju DUnit.

LISTING 10.2 Test klasa koju generiše čarobnjak Test Case Wizard kada se koristi u okruženju NUnit

```
unit TestNDemoClasses;

interface

uses
  NUnit.Framework, NDemoClasses;

type
  // Test methods for class TMyTest
```

```
  [TestFixture]
  TestTMyTest = class
  strict private
    FMyTest: TMyTest;
  public
    [SetUp]
    procedure SetUp;
    [TearDown]
    procedure TearDown;
  published
    [Test]
    procedure TestText;
    [Test]
    procedure TestAdd;
  end;

implementation

procedure TestTMyTest.SetUp;
begin
  FMyTest := TMyTest.Create;
end;

procedure TestTMyTest.TearDown;
begin
  FMyTest := nil;
end;

procedure TestTMyTest.TestText;
var
  ReturnValue: string;
begin
  ReturnValue := FMyTest.Text;
  // TODO: Validate method results
end;

procedure TestTMyTest.TestAdd;
var
  n: Integer;
begin
  // TODO: Setup method call parameters
  FMyTest.Add(n);
  // TODO: Validate method results
end;

end.
```

Ponovo možete da napišete programski kod za testove (koristeći sličan pristup). Da biste u okruženju NUnit imeplemntirali testove, umesto da koristite metode osnovne klase, treba da koristite metode nekoliko specifičnih klasa koje su definisane u imenovanom prostoru NUnit.Framework. Tačnije, treba da koristite klasu `Assert`, u kojoj postoje metodi `IsTrue`, `IsFalse`, `AreEqual`, `IsNotNull`, `IsNull` i `AreSame`. Drugim rečima, metod `Assert.AreEqual` koristite umesto metoda `CheckEquals`. Programski kod testa izgleda ovako:

```
procedure TestTMyTest.TestAdd;
begin
  FMyTest.Number := 10;
  FMyTest.Add(5);
  Assert.AreEqual(15, FMyTest.Number);
end;
```

Korisnički interfejs grafičke verzije okruženja NUnit je sličan grafičkom korisničkom interfejsu okruženja DUnit (i ostalim okruženjima za testiranje) (pogledajte sliku 10.12). Testiranje grešaka u aplikaciji sam ostavio kako bih vam pokazao kako izgleda neuspeh.

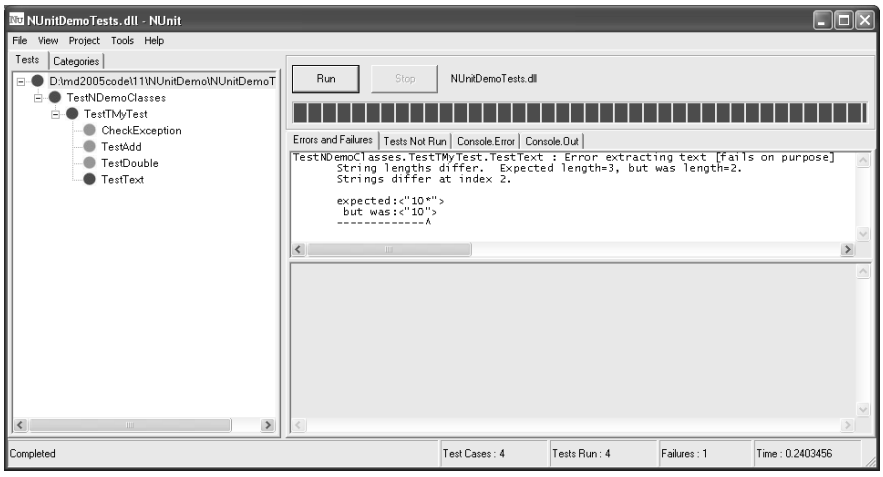
Način na koji okruženje NUnit testira izuzetke je zaista odličan. Treba samo da napišete programski kod koji prouzrokuje izuzetak, recimo ovakav programski kod:

```
procedure TestTMyTest.CheckException;
begin
  FMyTest.Add(-5); // raises exception
end;
```

Pomoću atributa označavate da očekujete pozivanje izuzetka, tako da deklaracija metoda izgleda ovako (u klasi `TestTMyTest`):

```
  [Test, ExpectedException(typeof(Exception))]
  procedure CheckException;
```

SLIKA 10.12
Korisnički interfejs okruženja NUnit



Ovo svakako nije kompletan uvod u okruženje NUnit, ali je sasvim dovoljno da počenete da ga koristite za testiranje. Za razliku od okruženja DUnit (koje se koristi samo za Delphi), koje je objašnjeno samo u nekoliko članaka, na Internetu postoji ogromna dokumentacija za okruženje NUnit, pa čak i knjige jer ga mnogi programeri koriste u radu sa raznim programskim jezicima i programskim okruženjima za .NET.

Saveti za testiranje jedinica

Pošto ste videli praktičnu primenu osnovnih osobina dva okruženja koja u Delphiju 2005 postoje za testiranje jedinica, pružiću vam još opštih informacija o testiranju jedinica. Smatrajte ih kolekcijom saveta koje vam dajem ja i iskusni programeri sa kojima sam razgovarao.

Testiranje jedinica, a ne proveravanje memorije Nije česta praksa da se testiraju konstruktori i destruktori, niti da se testira trajanje objekata. Pomoću posebnih alata se proverava gubljenje memorije.

Testiranje jedinica nije namenjeno testiranju korisničkog interfejsa Mada možete testirati status vizuelnih komponenti, testiranje jedinica ipak nije tome namenjeno. Uprkos nekim osobinama, čini se da je testiranje jedinica pogrešna alatka za obavljanje tog posla. Ono što možete uraditi jeste proveravanje logičkog toka vizuelne komponente, ali je u opštem slučaju bolje da taj programski kod podelite tako da postoje klase za logički tok elementa korisničkog intefejsa. U tom slučaju možete i morate testirati klasu kontrole korisničkog interfejsa.

Testiranje rada sa bazom podataka Testiranje jedinca možete upotrebiti za testiranje konzistentnosti rada sa bazom podataka ukoliko ste za objekte polja definisali poslovna pravila, na primer, za polja komponente DataSet. Moram da priznam da ovakvo testiranje spada u kategoriju funkcionalnog testiranja, a ne testiranja jedinica, ali okruženja za testiranje jedinica možete koristiti za obavljanje sličnih poslova.

Ne testirajte previše Kao što sam već pomenuo, ne testirajte svaku očiglednu mogućnost. Ukoliko operatoru dodele ne zadajete korektnu vrednost, onda bolje pronađite drugi kompajler! Testiranje trivijalnih operacija je uglavnom beskorisno. Testiranje svakog mogućeg slučaja je rasipanje. Napravite opšte testove. Napravite testove za greške koje ste uočili. Napravite testove za uslove koji su važni za postavljene zahteve. Pravite samo neophodne testove, a ne čitav skup testova.

Testiranje kao dokumentacija Dobar skup testova može biti dovoljno čitljiv kako bi mogao da se iskoristi za pravljenje dodatne dokumentacije za klase. Neki programeri kažu da testiranje može zameniti dokumentaciju, jer se može saznati šta metodi klase treba da obavljaju.

Testiranje pomaže da saznate koliko posla ste obavili Ukoliko prvo napravite testove, procenat uspešnih testova vam govori koliko toga ste uspeli da uradite, mada, ako testovima niste pokrili svaku oblast programa nećete dobiti preciznu indikaciju. Ukoliko napravite funkcionalne testove, ti testovi će vam pružiti bolju sliku o napretku projekta.

Testovi su bolji za OOP nego za RAD Ukoliko koristite objektno-orijentisano okruženje za rad sa bazama podataka, zapravo objektno-orijentisani sloj, možete obaviti mnogo bolje testiranje nego kada kontrole korisničkog intefejsa pridružite direktno komponentama za rad sa bazama podataka. Što više objektno-orijentisanog programskog koda pišete, to ćete imati više jedinica za testiranje.

Testiranje jedinica koristite kada preispitujete postojeći programski kod Kao što sam pomenuo, ukoliko morate da radite sa postojećim programskim kodom i planirate da ga preispitate, prvo treba da napišete jedinice za testiranje postojećeg programskog koda. Ukoliko su testovi uspešni, i budu uspešni nakon izmena, onda ćete znati da izmene koje ste načinili ne menjaju ponašanje programa.

Testiranje vremena potrebnog za izvršavanje Sa dobrim paketom testova, možete proveriti da li su izmene programskog koda ubrzale ili usporile izvršavanje programskog koda. Okruženje za testiranje jedinica nije zamena za dobar profajler, ali se može upotrebiti za grubo procenjivanje najsporijih pozivanja metoda i omogućava vam da uočite razlike u različitim implementacijama. Fino podešavanje ćete obaviti pomoću nekog drugog alata.

Šta je sledeće?

U ovom poglavlju ste saznali da u Delphiju 2005 postoji mnogo više od objektno-orijetnisanog programiranja. Delphi je potpuno objektno-orijentisan programski jezik koji podržava sve novije OOP pristupe i metodologije, uključujući korišćenje šablona i ekstremno programiranje. U Delphiju 2005 odmah možete koristiti novije tehnike, kao što su preispitivanje postojećeg programskog koda i testiranje jedinica. Iako se Delphijeve osobine ne mogu meriti sa alatkama koje su namenjene preispitivanju postojećeg programskog koda, kao što je Borlandov JBuilder, ipak imate neke osnovne mogućnosti.

Bez obzira na teoretski pristup, Delphijeve mogućnosti za preispitivanje programskog koda pomažu da se značajno ubrza pravljenje aplikacije. To je veoma važno.

Pošto ste se upoznali sa objektno-orijentisanim arhitekturama, možemo se posvetiti ostalim važnim oblastima Delphija. Ja ću vam objasniti strukturu fleksibilnih i dinamičkih arhitektura aplikacije, i korišćenje COM i Win32 programskog koda u platformi .NET. Kasnije, u delu knjige koji je posvećen bazama podataka, ponovo ću se vratiti na dizajniranje aplikacija (uključujući arhitekturu Model Driven Architecture - MDA).