13

Dijagnostika i kodni ugovori

Kada stvari krenu naopako, važno je imati informacije koje će vam pomoći da postavite dijagnozu problema. Integrisano razvojno okruženje ili alatka za otkrivanje grešaka mogu biti od velike pomoći ka tom cilju – ali su uglavnom na raspolaganju samo tokom razvoja aplikacije. Pošto aplikaciju otpremite korisnicima, sama aplikacija mora da prikuplja i beleži dijagnostičke podatke. Da bi zadovoljio taj zahtev, .NET Framework nudi skup alatki za beleženje dijagnostičkih podataka, nadziranje ponašanja aplikacije, otkrivanje grešaka pri izvršavanju i integrisanje s alatkama za otkrivanje grešaka, ako su one na raspolaganju.

.NET Framework omogućava i da koristite *kodne ugovore* (engl. *code contracts*). Uvedeni u verziji Frameworka 4.0, kodni ugovori omogućavaju metodama interakciju kroz skup međusobnih obaveza koje čine da metode *rano* prekidaju rad ako se te obaveze ne ispunjavaju.

Tipovi koje pominjemo u ovom poglavlju definisani su prevashodno u imenskim prostorima `System.Diagnostics` i `System.Diagnostics.Contracts`.

Uslovno prevođenje koda

Pretprocesorske direktive omogućavaju da uslovno prevedete proizvoljan odeljak C# koda. To su specijalne naredbe namenjene kompajleru koje počinju simbolom # (i, za razliku od ostalih konstrukata jezika C#, moraju se zadavati svaka u zasebnom redu). Logički gledano, one se izvršavaju pre glavng postupka prevođenja koda (mada, u praksi, kompajler ih obrađuje tokom faze leksičke analize koda). Pretprocesorske direktive za uslovno prevođenje koda jesu: `#if`, `#else`, `#endif` i `#elif`.

Direktive `#if` nalažu kompajleru da zanemari određeni odeljak koda ako nije definisan zadati *simbol*. Simbol možete definisati pomoću pretprocesorske direktive `#define` ili odgovarajuće opcije koju zadate kompajleru. Direktiva `#define` važi samo za sadržaj *datoteke* u kojoj je zadata, dok opcija kompajlera važi za ceo *sklop*:

```
#define TESTMODE            // direktive #define morate zadati na početku datoteke
                            // Imena simbola se po konvenciji pišu velikim slovima.
using System;

class Program
{
  static void Main()
  {
#if TESTMODE
```

```
        Console.WriteLine ("in test mode!");     // REZULTAT: in test mode!
#endif
  }
}
```

Kada bismo izbrisali prvi red, program bi se preveo u izvršnu verziju iz koje je bi bila uklonjena naredba `Console.WriteLine`, kao kada bismo je u izvornoj verziji pretvorili u komentar.

Direktiva `#else` je analogna naredbi `else` jezika C#, a `#elif` je ekvivalentna direktivi `#else` kojoj sledi `#if`. Pomoću operatora `||`, `&&` i `!` možete obavljati logičke operacije *disjunkcije*, *konjukcije* i *negacije*:

```
#if TESTMODE && !PLAYMODE      // if TESTMODE and not PLAYMODE
  ...
```

Međutim, imajte u vidu da ne sastavljate običan C# izraz, a simboli s kojima radite nemaju apsolutno nikakve veze s *promenljivama* – ni statičkim, ni drugačijim.

Da biste definisali simbol koji važi za ceo sklop, upotrebite opciju kompajlera `/define` kada kôd prevodite s komandne linije:

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Visual Studio omogućava da zadate opcije za uslovno prevođenje, u odeljku Project Properties.

Ako ste određeni simbol definisali na nivou celog sklopa, ali želite da se on u određenoj datoteci zanemari, to možete zadati pomoću direktive `#undef`.

Uslovno prevođenje u poređenju sa statičkim opcijama

Prethodni primer se može implementirati i pomoću jednostavnog statičkog polja:

```
static internal bool TestMode = true;

static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

Prednost ovog rešenja je to što omogućava konfigurisanje u vreme izvršavanja koda. Zašto bismo se onda opredelili za uslovno prevođenje? Razlog je to što uslovno prevođenje omogućava da radite i stvari koje nisu moguće pomoću statičkih opcija, kao što su:

- Uslovno uključivanje atributa
- Menjanje deklarisanog tipa promenljive
- Menjanje imenskog prostora ili alijasa tipa unutar direktive `using` – na primer:

```
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
     MyCompany.Widgets.Gadget;
  #endif
```

Unutar direktive za uslovno prevođenje možete zadati i vrlo složene promene, što vam omogućava da birate između starih i novih verzija, da pišete biblioteke koje se prevode za razne verzije Frameworka ili da koristite mogućnosti najnovije verzije Frameworka kada su na raspolaganju.

Još jedna prednost uslovnog prevođenja jeste to što se kôd za otkrivanje grešaka može odnositi na tipove koji se nalaze u sklopovima koji neće biti sastavni deo konačne verzije aplikacije koju ćete distribuirati.

Atribut Conditional

Atribut `Conditional` nalaže kompajleru da zanemari pozive određenoj klasi ili metodi, ako zadati simbol nije definisan.

Da biste videli koliko je to korisno, pretpostavimo da ste napisali sledeću metodu za evidentiranje satusnih podataka u datoteku:

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

A sada zamislite da želite da se ova metoda izvršava samo ako je definisan simbol `LOGGINGMODE`. Prvo rešenje je da pozive metodi `LogStatus` umetnete unutar direktive `#if`:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

To daje savršen rezultat, ali je nezgrapno. Drugo rešenje je da direktivu `#if` umetnete unutar koda metode `LogStatus`. Međutim, to pravi problem ako se metoda `LogStatus` poziva na sledeći način:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

U tom obliku se metoda `GetComplexMessageHeaders` uvek poziva – što može biti uzrok slabijih performansi.

Funkcionalnost prvog rešenja možemo kombinovati s pogodnošću drugog ako metodi `LogStatus` pridružimo atribut `Conditional` (definisan u imenskom prostoru `System.Diagnostics`):

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

Time nalažemo kompajleru da pozive metodi `LogStatus` obrađuje kao da se nalaze unutar direktive `#if LOGGINGMODE`. Ako taj simbol nije definisan, svi pozivi metodi `LogStatus` uklanjaju se iz prevedene verzije – uključujući i iz izraza koji se metodi zadaju kao argument. (Iz tog razloga, izbegavaju se i svi sporedni efekti.) Ovo je izvodljivo čak i kad se metoda `LogStatus` i njen pozivalac nalaze u različitim sklopovima.

Još jedna korist od atributa `[Conditional]` jeste to što se ispunjavanje zadatog uslova proverava kada se prevodi *pozivalac* metode, a ne kada se prevodi *pozvana* metoda. To je dobro zato što omogućava da napišete biblioteku metoda kao što je `LogStatus` – i da prevedete samo jednu verziju te biblioteke.

U vreme izvršavanja koda, atribut `Conditional` se zanemaruje – to je naredba koja važi isključivo za kompajler.

Alternative za atribut Conditional

Atribut `Conditional` je beskoristan ako vam treba da određenu funkcionalnost uključujete ili isključujete u vreme izvršavanja koda: umesto atributa, morate primeniti neko rešenje sa statičkom promenljivom. To nameće pitanje kako elegantno zaobići izračunavanje vrednosti argumenata kad pozivate metode za uslovno beleženje podataka u dnevnik. Rešenje pomoću funkcije izgleda ovako:

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

  static void LogStatus (Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
  }
}
```

Lambda izraz omogućava pozivanje metode bez složene sintakse:

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

Ako polje `EnableLogging` ima vrednost `false`, metoda `GetComplexMessageHeaders` se ne poziva.

Klase Debug i Trace

Klase `Debug` i `Trace` su statičke klase koje pružaju osnovne mogućnosti za beleženje podataka i ispitivanje da li su dati uslovi ispunjeni. Klase su međusobno vrlo slične; glavna razlika je njihova namena. Klasa `Debug` je namenjena upotrebi u verzijama koda koje se prevode u režimu `Debug`; klasa `Trace` je namenjena upotrebi i u režimu `Debug` i u režimu `Release`. U tom cilju:

Sve metode u klasi `Debug` definisane su s atributom `[Conditional("DEBUG")]`.
Sve metode u klasi `Trace` definisane su s atributom `[Conditional("TRACE")]`.

To znači da kompajler uklanja sve pozive metoda klasa `Debug` ili `Trace` osim kada definišete simbole `DEBUG` odnosno `TRACE`. Visual Studio standardno definiše oba simbola (`DEBUG` i `TRACE`) kada za projekat odaberete konfiguraciju *debug* – ali samo simbol `TRACE` u konfiguraciji *release*.

Obe klase, `Debug` i `Trace`, imaju metode `Write`, `WriteLine` i `WriteIf`. One standardno ispisuju poruke u prozor za rezultate dibagera:

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

Klasa `Trace` ima i metode `TraceInformation`, `TraceWarning` i `TraceError`. Razlika u ponašanju između njih i metoda `Write` zavisi od aktivnih osluškivača (objekti tipa `TraceListener`), koje opisujemo u odeljku „Klasa TraceListener“.

Metode Fail i Assert

Klase `Debug` i `Trace` obe imaju metodu `Fail` i `Assert`. Metoda `Fail` šalje poruku svakom objektu tipa `TraceListener` u kolekciji `Listeners` (videti naredni odeljak) klase `Debug` ili `Trace`. Svaki takav objekat standardno ispisuje poruku u prozor za rezultate alatke za otkrivanje grešaka (dibagera) i u okviru za dijalog:

```
Debug.Fail ("File data.txt does not exist!");
```

Dijalog koji se otvara nudi opcije ignore (zanemariti), abort (prekinuti) i retry (pokušati ponovo). Ova poslednja opcija omogućava da program povežete s nekim dibagerom, što je korisno za brzo dijagnostikovanje problema.

Metoda `Assert` samo poziva metodu `Fail` ako njen argument `bool` ima vrednost `false` – to se zove *zadavanje pretpostavke* (engl. *assertion)* i pokazuje da u kodu postoji greška ako pretpostavka nije potvrđena. Za taj slučaj možete opciono zadati odgovarajuću poruku:

```
Debug.Assert (File.Exists ("data.txt"), "Datoteka data.txt ne postoji!");
var result = ...
Debug.Assert (result != null);
```

Metode `Write`, `Fail` i `Assert` imaju i preklopljene verzije koje, osim teksta poruke, prihvataju i argument za opis kategorije greške tipa `string`, što može biti korisno pri obradi rezultata.

Umesto da zadate pretpostavku, možete generisati izuzetak ako je ispunjen suprotan uslov. To je uobičajena praksa kada se ispituje ispravnost argumenata metode:

```
public void ShowMessage (string message)
{
  if (message == null) throw new ArgumentNullException ("message");
  ...
}
```

Takve „pretpostavke" se bezuslovno prevode i manje su fleksibilne jer rezultate nepotvrđene pretpostavke ne možete obraditi pomoću objekata `TraceListener`. Osim toga, tehnički govoreći, to nisu pretpostavke. Pretpostavka je nešto što, ako nije potvrđeno, pokazuje grešku u kodu tekuće metode. Pojava izuzetka pri ispitivanju ispravnosti argumenta metode znak je greške u kodu *pozivaoca* metode.

Uskoro ćemo videti kako *kodni ugovori* proširuju principe metoda `Fail` i `Assert` da bi pružili moćnija i fleksibilnija rešenja.

Klasa TraceListener

Obe klase – `Debug` i `Trace` – imaju svojstvo `Listeners`, čija je vrednost statička kolekcija osluškivača, tj. instanci klase `TraceListener`. Ti objekti su odgovorni za dalju obradu sadržaja koji proizvode metode `Write`, `Fail` i `Trace`.

Kolekcija `Listeners` svake klase standardno sadrži samo jedan osluškivač (`DefaultTraceListener`). Taj podrazumevani osluškivač ima dve ključne odlike:

- Kada se poveže s nekom alatkom za otkrivanje grešaka, kao što je Visual Studio, poruke osluškivača se ispisuju u prozoru za rezultate te alatke; u suprotnom, sadržaj poruka se zanemaruje.
- Kada se pozove metoda `Fail` (ili pretpostavka ne bude potvrđena), pojavljuje se okvir za dijalog koji korisniku nudi opcije da nastavi program (continue), da ga prekine (abort) ili da ponovo pokuša (retry) ili program poveže sa alatkom za otkrivanje grešaka (attach/debug) – bez obzira na to da li je programu već pridružen dibager.

To ponašanje možete promeniti ako iz kolekcije (opciono) uklonite podrazumevani osluškivač, a zatim kolekciji dodate jedan ili više vlastitih osluškivača. Osluškivač možete napisati od početka (tako što nalsedite klasu `TraceListener`) ili upotrebite neki od postojećih tipova:

- `TextWriterTraceListener` upisuje u tok, u objekat `TextWriter` ili u datoteku.
- `EventLogTraceListener` upisuje u Windowsov sistemskii dnevnik događaja.
- `EventProviderTraceListener` upisuje u podsistem ETW (Event Tracing for Windows) pod Windowsom Vista i novijim.
- `WebPageTraceListener` upisuje na ASP.NET veb stranicu.

Klasu `TextWriterTraceListener` dalje nasleđuju klase `ConsoleTraceListener`, `Delimited-ListTraceListener`, `XmlWriterTraceListener` i `EventSchemaTraceListener`.

Nijedan od ovih osluškivača ne otvara okvir za dijalog kada se pozove metoda `Fail` – tako se ponaša samo `DefaultTraceListener`.

Primer koji sledi uklanja podrazumevani osluškivač klase `Trace`, zatim dodaje tri osluškivača – jedan koji podatke upisuje u datoteku, drugi koji ih ispisuje na konzolu i treći koji ih upisuje u Windowsov sistemski dnevnik događaja:

```
// Uklanja podrazumevani osluškivač:
Trace.Listeners.Clear();

// Dodaje pisač koji podatke dopisuje na kraj datoteke trace.txt:
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));

// Pribavlja izlazni tok konzole, koji zatim dodaje kao osluškivač:
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add (new TextWriterTraceListener (tw));

// Definiše izvor za Windowsov dnevnik događaja a zatim dodaje osluškivač.
// Pošto CreateEventSource zahteva administratorska ovlašćenja, ovaj deo
// bi se obično obavio tokom instaliranja aplikacije.
if (!EventLog.SourceExists ("DemoApp"))
  EventLog.CreateEventSource ("DemoApp", "Application");

Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

(Osluškivače možete dodati i pomoću konfiguracione datoteke aplikacije; to je zgodno jer ispitivačima aplikacije omogućava da konfigurišu osluškivač nakon prevođenja i sklapanja aplikacije – MSDN članak o tome nalazi se na *http://albahari.com/traceconfig*.)

U slučaju Windowsovog dnevnika događaja, poruke koje šaljete iz metoda `Write`, `Fail` ili `Assert` uvek se prikazuju u kategoriji „Information“, u prozoru prikaza Windowsovih događaja. Međutim, poruke koje šaljete pomoću metoda `TraceWarning` i `TraceError`, prikazuju se u kategoriji upozorenja (warning) ili u kategoriji grešaka (errors).

Klasa `TraceListener` ima i svojstvo `Filter` tipa `TraceFilter` koje se može podesiti radi filtriranja poruka koje osluškivač prima. Da biste to postigli, treba da napravite instancu jedne od unapred definisanih potklasa (`EventTypeFilter` ili `SourceFilter`), ili da nasledite klasu `TraceFilter` i redefinišete njenu metodu `ShouldTrace`. To možete iskoristiti da biste, na primer, filtrirali poruke po kategoriji.

Klasa `TraceListener` definiše i svojstva `IndentLevel` i `IndentSize` za zadavanje nivoa i veličine uvlaka, kao i svojstvo `TraceOutputOptions` za upisivanje dodatnih podataka:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Opcije zadate pomoću objekata `TraceOutputOptions` primenjuju se kada pozivate metode `Trace`:

```
Trace.TraceWarning ("Orange alert");

DiagTest.vshost.exe Warning: 0 : Orange alert
     DateTime=2007-03-08T05:57:13.6250000Z
     Callstack=   at System.Environment.GetStackTrace(Exception e, Boolean
needFileInfo)
     at System.Environment.get_StackTrace()     at ...
```

Pražnjenje i zatvaranje osluškivača

Neki osluškivači, kao što je `TextWriterTraceListener`, upisuju svoje podatke u tok, koji je podložan keširanju. To ima dve posledice:

- Poruka se možda neće odmah pojaviti u izlaznom toku ili datoteci.
- Osluškivač morate zatvoriti – ili barem isprazniti – pre nego što aplikacija završi rad jer ćete inače izgubiti ono što se nalazi u kešu (standardne veličine najviše 4 KB, ako upisujete u datoteku).

Klase `Trace` i `Debug` imaju statičke metode `Close` i `Flush` koje pozivaju metodu `Close` odnosno `Flush` svih osluškivača (koji pak pozivaju metodu `Close` odnosno `Flush` internog pisača ili toka s kojim rade). Metoda `Close` implicitno poziva metodu `Flush`, zatvara resurse operativnog sistema i sprečava dalje upisivanje podataka.

Opšte pravilo preporučuje da metodu `Close` pozivate na završetku aplikacije, a metodu `Flush` kad god vam zatreba da upišete tekuće podatke. To važi u slučajevima kada radite s pisačima čije odredište je tok ili datoteka.

Klase `Trace` i `Debug` imaju i svojstvo `AutoFlush`, koje, ako mu zadate vrednost `true`, pokreće izvršavanje metode `Flush` posle svake poruke.

Preporučuje se da svojstvo `AutoFlush` objekata `Debug` i `Trace` podesite na `true` ako radite s osluškivačima koji upisuju u tok ili datoteku. Ako ne uradite tako, kada se pojavi neobrađeni izuzetak ili katastrofalna greška, možete izgubiti poslednjih 4 KB dijagnostičkih podataka.

Kodni ugovori

Već smo ranije pomenuli koncept *pretpostavke*, pomoću koje utvrđujete da li su u programu ispunjeni određeni uslovi. Ako data pretpostavka nije potvrđena, to je znak da postoji greška, koja se obrađuje najčešće tako što se pokrene alatka za otkrivanje grešaka (kada je projekat u režimu Debug) odnosno generiše izuzetak (kada je projekat sklopljen u varijanti Release).

Pretpostavke slede princip da ako nešto krene naopako, najbolje je da to saznamo rano i blizu izvora greške. To je obično bolje nego da pokušamo da nastavimo s lošim podacima – što kasnije u programu može postati uzrok pogrešnih rezultata, neželjnih sporednih efekata ili izuzetka (a sve je to teže dijagnostikovati).

Tokom vremena, pretpostavke su se zadavale u dva oblika:

- Pozivanjem metode `Assert` klase `Debug` ili `Trace`
- Izazivanjem izuzetaka (kao što je `ArgumentNullException`)

Framework 4.0 je uveo novu mogućnost nazvanu *kodni ugovor* (engl. *code contract*), koji oba oblika zamenjuje jednoobraznim sistemom. Taj sistem omogućava da zadajete ne samo jednostavne pretpostavke, nego i moćnije pretpostavke u obliku *ugovora*.

Kodni ugovori potiču od principa „Design by Contract" (dizajn zasnovan na ugovorima) programskog jezika Eiffel, gde se interakcija između funkcija odvija kroz sistem međusobnih obaveza i koristi. Ukratko, funkcija definiše *preduslove* (engl. *preconditions*) koje treba da ispuni klijent (pozivalac) funkcije, a zauzvrat ona klijentu garantuje pouzdano *konačno stanje* (engl. *postcondition*) kada funkcija završi svoj rad.

U imenskom prostoru `System.Diagnostics.Contracts` postoje dve vrste kodnih ugovora.

Iako su tipovi koji podržavaju kodne ugovore ugrađeni u .NET Framework, binarni pisač i alatke za statičku proveru ugovora treba zasebno preuzeti sa veb lokacije Microsoft DevLabs (*http://msdn.microsoft.com/devlabs*). Te alatke morate instalirati pre nego što počnete da koristite kodne ugovore u Visual Studiju.

Zbog čega biste koristili kodne ugovore?

Ilustracije radi, napisaćemo metodu koja dodaje novu stavku na listu samo ako ona ne postoji na listi – s dva *preduslova* (engl. *precondition*) i jednim *konačnim stanjem* (engl. *postcondition*):

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Preduslov
  Contract.Requires (!list.IsReadOnly);      // Preduslov
  Contract.Ensures (list.Contains (item));   // Konačno stanje
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

Preduslovi se definišu pomoću metode `Contract.Requires`, a da li su ispunjeni ispituje se kada se ciljna metoda pokrene. Konačno stanje se definiše pomoću metode `Contract.Ensures` a ispituje se, ne na mestu gde je definisano u kodu, nego *kada se metoda završi*.

Preduslovi i konačna stanja nalik su pretpostavkama i, u ovom primeru, otkrivaju sledeće greške:

- Metoda je pozvana s argumentom koji je `null` ili imamo listu koja se može samo čitati
- U kodu metode smo zaboravili da dodamo stavku na listu

Preduslove i (očekivana) konačna stanja morate navesti na početku metode. To olakšava dobar dizajn: ako u kodu metode koji sledi ne ispunite uslove ugovora, greška će biti otkrivena.

Osim toga, ti uslovi i završna stanja čine *ugovor* za tu metodu koji se može čitati. Metoda `AddIfNotPresent` poručuje svojim korisnicima:

- „Morate me pozvati sa listom koja nije `null` i koja dozvoljava dodavanje nove stavke".
- „Kada završim svoj posao, lista će sadržati stavku koju ste mi zadali".

Te činjenice mogu se preneti u XML datoteku dokumentacije sklopa (u Visual Studiju, to možete uraditi ako u prozoru Project Properties izaberete jezičak Code Contracts, dozvolite izradu referentnog sklopa za ugovore i potvrdite opciju „Emit Contracts into XML doc file"). Alatke, kao što je SandCastle, mogu onda da umeću uslove ugovora u dokumentacione datoteke.

Kodni ugovori omogućavaju i da ispravnost programa analiziraju alatke za statičko ispitivanje ispunjavanja ugovora. Na primer, ako metodu `AddIfNotPresent` pozovete s argumentom `list` čija vrednost može biti `null`, takva alatka za statičko ispitivanje ispravnosti mogla bi da vas upozori čak i pre nego što pokrenete program.

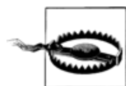
Još jedna prednost kodnih ugovora jeste lakoća upotrebe. U našem primeru, lakše je zadati konačno stanje na samom početku metode nego na obe njene izlazne tačke. Kodni ugovori podržavaju i *objektne invarijante* – koje dalje smanjuju potrebu za ponavljanjem koda i obezbeđuju pouzdanije poštovanje pravila.

Preduslovi i konačna stanja mogu se zadati i za članove interfejsa i apstraktne metode, što je nemoguće kod standardnih rešenja za utvrđivanje ispravnosti koda. Osim toga, ne može se dogoditi da nasleđene klase greškom zaobiđu preduslove i konačna stanja koji su definisani u virtuelnim metodama.

Još jedna prednost kodnih ugovora jeste to što se željeno ponašanje u slučaju kršenja ugovora može zadati lakše i na više načina nego kad se za tu namenu koristi klasa metoda `Debug.Assert` ili generišu izuzeci. Osim toga, može se obezbediti da se kršenje ugovora uvek registruje – čak i kad izuzetke zbog kršenja ugovora „progutaju" blokovi za obradu izuzetaka koji su na višem nivou stabla pozivanja.

Nedostatak upotrebe kodnih ugovora jeste to što .NET-ova implementacija koristi *binarni pisač* – alatku koja menja sadržaj sklopa posle prevođenja izvornog koda. To usporava postupak sklapanja aplikacije i komplikuje servise koji pri svom radu pozivaju kompajler C# koda (eksplicitno ili pomoću klase `CSharpCodeProvider`).

Upotreba kodnih ugovora može biti i uzrok nešto slabijih performansi u vreme izvršavanja koda, ali se to lako ispravlja tako što se u varijanti Release aplikacije isključi proveravanje da li se ugovori krše.



Još jedno ograničenje kodnih ugovora jeste to da se ne mogu koristiti za ispitivanje bezbednonih mera, zato što se u vreme izvršavanja aplikacije ugovori mogu zaobići (obradom događaja `ContractFailed`).

Principi kodnih ugovora

Kodni ugovori obuhvataju *preduslove*, *konačna stanja*, *pretpostavke* i *objektne invarijante*. To su sve elementi ugovora koji se nalaze u sklopu i mogu se čitati od spolja. Razlikuju se po trenutku u kojem se ispituje da li su ispunjeni:

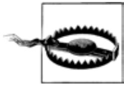
- *Preduslovi* se ispituju kada se funkcija pokrene.
- *Konačna stanja* se proveravaju kada funkcija završi rad.
- *Pretpostavke* se ispituju na mestima gde se pojavljuju u kodu.
- *Objektne invarijante* se ispituju nakon izvršavanja svake javne funkcije u klasi.

Kodni ugovori se definišu isključivo pozivanjem (statičkih) metoda klase `Contract`. To kodne ugovore čini *nezavisnim od jezika*.

Kodni ugovori se mogu definisati ne samo u metodama, nego i u drugim vrstama funkcija, kao što su konstruktori, svojstva, indekseri i operatori.

Prevođenje

Gotovo sve metode klase `Contract` definisane su s atributom `[Conditional("CONTRACTS_FULL")]`. To znači da sve dok ne definišete simbol `CONTRACTS_FULL`, (najveći) deo koda koji se tiče kodnih ugovora nema nikakvog efekta. Visual Studio automatski definiše simbol `CONTRACTS_FULL` ako u odeljku Code Contracts prozora Project Properties uključite proveravanje kodnih ugovora. (Da bi se pojavio jezičak Code Contracts, morate preuzeti i instalirati alatku Contracts sa Microsoftove veb lokacije DevLabs.)



Uklanjanje simbola CONTRACTS_FULL može se činiti kao jednostavan način za isključivanje provere ugovora. Međutim, on ne važi za uslove Requires<TException> (koje ćemo uskoro detaljno opisati).

Jedini način da u kodu koji koristi Requires<TException> isključite proveravanje kodnih ugovora jeste da definišete simbol CONTRACTS_FULL i da zatim naložite binarnom pisaču da ukloni kôd koji se tiče kodnih ugovora, tako što izabrete nivo provere „none" (ništa).

Binarni pisač

Pošto prevedete kôd koji sadrži ugovore, morate pozvati binarni pisač, *ccrewrite.exe* (Visual Studio to čini automatski ako je uključeno proveravanje ugovora). Binarni pisač premešta definicije konačnih stanja (i objektne invarijante) na odgovarajuće mesto, poziva preduslove i objektne invarijante u redefinisanim metodama i zamenjuje pozive članova klase Contract pozivima članova *klase za ugovore izvršnog okruženja*. Evo kako bi izgledala (pojednostavljena) verzija našeg ranijeg primera posle prepravljanja pomoću binarnog pisača:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  __ContractsRuntime.Requires (list != null);
  __ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
  }
  __ContractsRuntime.Ensures (list.Contains (item));   // Konačno stanje
  return result;
}
```

Ako ne pozovete binarni pisač, klasa Contract neće biti zamenjena klasom __ContractsRuntime i zato će se generisati izuzeci.

Tip __ContractsRuntime je podrazumevana klasa za ugovore u izvršnom okruženju. U složenijim slučajevima, možete zadati vlastitu klasu za ugovore u izvršnom okruženju, pomoću opcije /rw ili u Visual Studju, u odeljku Code Contracts prozora Project Properties.

Pošto se klasa __ContractsRuntime isporučuje uz binarni pisač (koji nije standardni deo .NET Frameworka), binarni pisač zapravo umeće klasu __ContractsRuntime u vaš preveden sklop. Kôd te klase možete videti ako pomoću neke alatke za pregled sklopa (disasembler) ispitate sklop u kojem se koriste kodni ugovori.

Binarni pisač ima opcije za isključivanje nekih ili svih nivoa provera ugovora: to opisujemo u odeljku „Selektivno proveravanje ugovora", na strani 481. Uobičajeno je da se kompletno proveravanje zadaje kada se aplikacija prevodi u varijantu Debug, a u varijanti Release zadaje se samo određeni podskup provera.

Pretpostavka ili izuzetak u slučaju kršenja ugovora

Binarni pisač omogućava i da birate da li da se u slučaju kršenja ugovora prikaže okvir za dijalog, kao kada pretpostavka nije potvrđena, ili da se generiše izuzetak ContractException. Prva mogućnost je uobičajena u varijanti Debug; druga u varijanti Release. Da biste izabrali ovu drugu mogućnost, zadajte opciju /throwonfailure kada pozovete binarni pisač, ili uklonite znak potvrde pored polja „Assert on contract failure" u odeljku Code Contracts prozora Project Properties u Visual Studiju.