



Embarcadero Akademija: Kako do delujočih programov?

Primož Gabrijelčič
<http://primoz.gabrijelcic.org>

Kazalo

Kako do delajočih programov?.....	3
Varno programiranje	4
Razumljiva koda.....	4
Preverjanje vhodnih podatkov	7
Preverjanje rezultatov funkcij	9
Ustvari / uniči	10
Izjeme	10
Design by Contract	12
Assert ali raise?.....	13
Stavek »with«	13
Splošni nasveti.....	14
Statična analiza.....	14
Napredno razhroščevanje	15
Nastavitev.....	15
Lastnosti prekinitiv	16
Pogojne prekinitve	17
Število ponovitev.....	18
Iskanje napak med inicializacijo	18
Izpisovanje dogajanja	19
Skupine prekinitiv.....	20
Razhroščevanje večnitnih programov	21
Izjeme	21
Strojno podprte prekinitve	22
Premikanje točke izvajanja.....	23
Prekinitve na skladu izvajanja	24
Vrnitev iz metode	24
Oblikovanje prikaza podatkov	24
Organiziranje opazovanih vrednosti.....	25
Prelisičite optimizator	26
Viri	27
Orodja.....	27
Principi dobrega programiranja	27
Razhroščevanje.....	27

Vsi programi, omenjeni v tem dokumentu, so na voljo na naslovu

<http://17slon.com/EA/EA-Defensive.zip>.

Kako do delajočih programov?

Ko damo v svet novo različico programa, smo šele na pol poti. Prej ali slej bodo uporabniki naleteli na množico napak, ki jih bo treba odpraviti. Pri iskanju razlogov, zaradi katerih pride do napačnega delovanja, se še kako pozna »kilometrina« programerja in njegova sposobnost rabe programerskih orodij. Seveda pa je še lepše, če programiramo tako, da do napak ne more priti.

Pot do zadovoljne stranke je torej program, ki ne »crkuje«, ki ga lahko hitro spremenimo, ko se za to pojavitjo potrebe, in ki ga znamo hitro popraviti, ko (tu ni nobenega »če«) pride do napake. Enostavne rešitve, ki bi to dosegla v dveh klikih, žal ni. Do dobrega programa pripelje le naporno delo.

Danes si bomo ogledali dve ločeni, a povezani temi – varno programiranje in rabo naprednih funkcij razhroščevalnika.

Varno programiranje

Če v spletu iščete frazo »defensive programming«, ne boste našli kaj dosti. Članek na Wikipediji in nekaj krajsih navodil, pretežno namenjenih programerjem v jezikih iz družine C. Pa še ti članki se bodo ukvarjali pretežno s tem, kako preverjati velikost vhodnih podatkov (bufferjev), pa povedali bodo, da je treba preverjati rezultate sistemskih funkcij in še kakšno malenkost. To so sicer smiselnii nasveti, a po mojem mnenju se varno programiranje začne bistveno prej.

Zavedati se moramo, da je programska koda živa tvorba. S časom se spreminja, prav tako pa se spreminja okolje, v katerem se izvaja (operacijski sistem) in ekologija celotne aplikacije. Ne moremo vedeti, kdo bo čez nekaj let kodo uporabljal, kako jo bo klical, kako jo bo hotel predelati, kje jo bo izvajal in še in še. Zato mora biti koda napisana razumljivo in z mislio na prihodnost (»future-proof«). Varno programiranje se torej začne z razumljivostjo.

Razumljiva koda

Pa začnimo s protiprimerom – relativno slabo napisano kodo, ki sem jo našel v eni od komponent, ki jih še vedno uporabljam v naših programih (primer na sosednji strani).

Verjetno se boste strinjali, da je razumevanje te funkcije za programerja, ki je ne pozna, a bi jo želel spremeniti, kar težka naloga. Naštejmo nekaj največjih težav.

1. Dolžina. Funkcija je predolga, da bi jo lahko hitro razumeli.
2. Namen. Funkcija počne preveč različnih stvari. Najprej razdeli vhodni niz v eno polje, nato to polje predela v drugo polje, nato vrednost drugega polja pošlje v neko drugo funkcijo, da dobi rezultat.
3. Vhodi format ni dokumentiran. Vzeti si moramo kar nekaj časa, če hočemo razumeti, na kakšen način so zapisani vhodni podatki.
4. Nelokalnost. Funkcija spreminja zunanjí spremenljivki `VOpacity` in `Color`, namesto da bi ju vrnila kot parametra. Obe spremenljivki sta parametra zunanje funkcije, znotraj katere je definirana lokalna funkcija `FindHSLColor`. Celota je torej še daljša in težje razumljiva, kot bi si mislili na prvi pogled.
5. Slaba imena spremenljivk. Bistveno bolje bi bilo, če bi se `A` imenoval `sComponent`, `C` `intComponent` in `K` `colorComponent`. Ali pa karkoli drugega, kar ustreza vaši filozofiji poimenovanja.

```

function FindHSLColor(S: string): Boolean;
type
  Colors = (hue, saturation, luminance);
var
  I, J: Integer;
var
  A: array[hue..luminance] of string;
  C: array[hue..luminance] of Integer;
  K: Colors;
begin
  I := Pos('(', S);
  J := Pos(')', S);
  if (I > 0) and (J > 0) then
    begin
      S := copy(S, 1, J - 1);
      S := Trim(Copy(S, I + 1, 255));
      for K := hue to saturation do
        begin
          I := Pos(',', S);
          A[K] := Trim(copy(S, 1, I - 1));
          S := Trim(Copy(S, I + 1, 255));
        end;
      I := Pos(',', S);
      if I > 0 then begin
        A[luminance] := Trim(copy(S, 1, I - 1));
        S := Trim(Copy(S, I + 1, 255));
        VOpacity := OpacityFromStr(S);
      end else begin
        A[luminance] := S;
        VOpacity := 255;
      end;

      C[hue] := StrToIntDef(A[hue], 0);
      while C[hue] >= 360 do begin
        C[hue] := C[hue] - 360;
      end;
      while C[hue] < 0 do begin
        C[hue] := C[hue] + 360;
      end;
      for K := saturation to luminance do begin
        I := Pos('%', A[K]);
        if I > 0 then begin
          Delete(A[K], I, 1);
        end;
        C[K] := StrToIntDef(A[K], 0);
        if C[K] > 100 then begin
          C[K] := 100;
        end;
        if C[K] < 0 then begin
          C[K] := 0;
        end;
      end;
      Color := HSLUtils.HSLtoRGB(C[hue], C[saturation], C[luminance]);
      Result := True;
    end
    else
      Result := False;
  end;
end;

```

Lažje za razumevanje so krajše metode, ki uporabljajo spremenljivke in funkcije s smiselnimi imeni.

```
procedure TStaticCollector.AddToGeneration(generation: integer;
  const aData: TPointers; count: integer = 1);
var
  dataInfo: TDataInfo;
  idxData : integer;
begin
  CheckPromoteGeneration(generation);

  with FGenerationInfo[generation] do begin
    idxData := FindInGeneration(generation, aData);
    if idxData >= 1 then begin
      Data^[idxData].Count := Data^[idxData].Count + count;
      ResortGeneration(generation, idxData);
    end
    else begin
      dataInfo.Data := aData;
      dataInfo.Count := count;
      InsertIntoGeneration(generation, dataInfo);
    end;
  end;
end;
```

Kaj se lahko naučimo iz teh dveh primerov? Metode naj ne bodo predolge – največ toliko, da lahko celo metodo hkrati vidite na zaslonu. Uporablajte smiselna imena spremenljivk, funkcij, enot, kontrol itd. Dele, ki jih ni enostavno razbrati iz kode, dokumentirajte. Ne poskušajte vsega narediti v eni metodi, raje razbijte funkcionalnost na več delov.

Pri popravljanju stare kode so zelo priročna orodja za refaktorizacijo, denimo v RAD Studio vgrajeni *Refactor*, *Rename*, ali zunanje orodje *ModelMaker CodeExplorer (MMX)*.

Preglednost lahko znatno izboljša vpeljava pomožnih spremenljivk (tudi tu lahko pomaga *MMX*). Naslednji primer ne počne nič posebnega, a je njegovo funkcijo težko razbrati zaradi komplikiranih struktur podatkov.

```
for iTeletext := 0 to cfgSettings.Teletext.Count - 1 do begin
  if cfgSettings.Teletext[iTeletext].Input.DeviceType <> tdtDataBridge or
    continue;
  Activate(cfgSettings.Teletext[iTeletext].Input.CardNumber);
  for iOutput := 0 to cfgSettings.Teletext[iTeletext].Outputs.Count - 1 do begin
    if cfgSettings.Teletext[iTeletext].Outputs[iOutput].OutputType <> otSDI then
      continue;
    Activate(cfgSettings.Teletext[iTeletext].Outputs[iOutput].SDI.CardNumber);
  end;
end;
```

Koda postane bistveno lepša, če vpeljemo začasni spremenljivki `teletext` in `output`.

```

for iTeletext := 0 to cfgSettings.Teletext.Count - 1 do begin
    teletext := cfgSettings.Teletext[iTeletext];
    if teletext.Input.DeviceType <> tdtDataBridge then
        continue;
    Activate(teletext.Input.CardNumber);
    for iOutput := 0 to teletext.Outputs.Count - 1 do begin
        output := teletext.Outputs[iOutput];
        if output.OutputType <> otSDI then
            continue;
        Activate(output.SDI.CardNumber);
    end;
end;

```

Še lepo kodo bi dobili, če bi v strukturo vpeljali podporo za enumeratorje. S tem se izognemo števcema `iTeletext` in `iOutput`.

```

for teletext in cfgSettings.Teletext do begin
    if teletext.Input.DeviceType <> tdtDataBridge then
        continue;
    Activate(teletext.Input.CardNumber);
    for output in teletext.Outputs do begin
        if output.OutputType <> otSDI then
            continue;
        Activate(output.SDI.CardNumber);
    end;
end;

```

Preverjanje vhodnih podatkov

Pa smo prišli do nasveta, ki je tako ljub C-jevskim programerjem: »Preverjajte velikosti bufferjev!« No, v Delphiju to malokrat počnemo. Na voljo imamo veliko dinamičnih tipov podatkov, pri katerih ni tako lahko pisati izven obsega shrambe. Vseeno pa moramo včasih paziti tudi na to. Običajno takrat, ko uporabljamо sistemskо funkcijo `Move` ali statična polja (array).

V enoti DSiWin32 sem recimo našel test, ki pred premikanjem podatkov med dvema strukturama preveri, ali sta enako veliki.

```

Assert(SizeOf(TStartupInfoA) = SizeOf(TStartupInfoW));
Move(useStartInfo^, startupInfoW, SizeOf(TStartupInfoW));

```

Podoben primer iz moje kode, ki se pogovarja z gonilnikom za neko kartico.

```

Assert(SizeOf(h1Status.dsBoardAttributes) = SizeOf(l1Status.BrdAttrib));
Move(attrib, h1Status.dsBoardAttributes, SizeOf(h1Status.dsBoardAttributes));

```

Ali pa tale košček kode, ki preverja, da ne bo pisal izven meja polja.

```

var
  FEvents: array [0..63] of THandle;

procedure THTTransmitterWorker.AddEvent(event: THandle; cardNumber: integer;
  eventType: THTEventType);
begin
  if event = INVALID_HANDLE_VALUE then
    Exit;
  if FNumEvents > High(FEvents) then
    raise Exception.Create(['%s Too many events!']);
  FEvents[FNumEvents] := event;
  FEventToHT[FNumEvents] := cardNumber;
  FEventToType[FNumEvents] := eventType;
  Inc(FNumEvents);
end;

```

Trenutno se sicer nikakor ne more zgoditi, da bi koda zapolnila polje `FEvents`, ker je celoten razred napisan tako, da bo dodal v polje največ $2 + 3 * 16 = 51$ eventov, a kdo ve, kaj se bo zgodilo v daljni prihodnosti. Mogoče bo kdo predelal kodo, ki kliče `AddEvent`, velikost polja `FEvents` pa pozabil popraviti ... No, v tem primeru bo dobil dokaj jasno sporočilo o napaki. (Pa veliko sreče mu želim, prihodnjemu programerju, kajti `FEvents` se pošlje v Windows API klic `WaitForMultipleObjects`, ki sprejme največ 64 eventov ...).

Včasih se torej splača preverjati vhodne podatke tudi takrat, ko smo prepričani, da so pravilni. (»Saj to se ne more nikoli zgoditi!«) Naj ponovim: Kdo ve, kako se bo v prihodnosti spreminja program.

Takšni primeri, ko bo koda na neočiten način nehala delovati zaradi omejitve zunanjih funkcij, so odličen vir napak v prihodnosti in jih je koristno dokumentirati.

```

const
  //used in WaitForMultipleObjectsEx, don't increment
  CMaxEvents = 63;
  //must be <= (CMaxEvents-2) div (num elements in THTTransmitFormat)
  CMaxHTCards = 16;
var
  FEvents      : array [0..CMaxEvents] of THandle;
  FEventToHT   : array [0..CMaxEvents] of integer;
  FEventToType: array [0..CMaxEvents] of THTEventType;
  FHT         : array [1..CMaxHTCards] of THTInfo;

```

V moji kodi se tak »future-proofing« velikokrat pojavi na mestih, ki obdelujejo enumerirane tipe. Takrat običajno poskrbim, da koda javi napako, če naleti na nepričakovano vrednost. Na primer:

```
procedure THTTransmitterWorker.ProcessMessages;
var
  msg: T0mniMessage;
begin
  while FTask.Comm.Receive(msg) do begin
    if msg.MsgID = MSG_REGISTER_HT then
      MsgRegisterHT(msg)
    else if msg.MsgID = MSG_SEND then
      MsgSend(msg)
    else
      Log(['%s] Invalid message received: %d', [FTask.Name, msg.MsgID], svError);
  end;
end;
```

Iz istega razloga imajo moji stavki `case` zelo pogosto tudi del `else`.

```
case device of
  tdtDataBridge,
  tdtSubtitleGenerator: FWorkers[device].ThreadLimit := 10;
  tdtPCTV:             FWorkers[device].ThreadLimit := 1;
  else raise Exception.Create('TGpDVBTTeletextReceiver.Create: Unknown device');
end;
```

Prav tako bom le redko zapisal:

```
for deviceType := tdtDataBridge to tdtPCTV do
```

Raje:

```
for deviceType := Low(TTXDeviceType) to High(TTXDeviceType) do
```

Preverjanje rezultatov funkcij

Vedno preverite, kakšen rezultat je vrnila funkcija! To bi moralo biti očitno vsem, a v praksi na to hitro pozabimo, še posebej pri klicih sistemskih funkcij.

Dve minuti z Googlom sta mi dala naslednja grozna primera.

```
MayContinue = ::CreateEvent (NULL, FALSE, FALSE, NULL);
iconn=InternetOpen(NULL,INTERNET_OPEN_TYPE_PROXY,proxy_url,NULL,
  INTERNET_FLAG_ASYNC);
call=InternetSetStatusCallback(iconn,(INTERNET_STATUS_CALLBACK)CallbackFunction);
while(f[FLAG_FTP_ITERATE])
```

Koda mirno pošlje rezultat klica `InternetOpen` v `InternetSetStatusCallback`, tudi če prvi klic ne uspe. Potem pa še ignorira rezultat drugega klica ☺

```
BufLen:= sizeof(IP_ADAPTER_INFO);
GetAdaptersInfo(nil, BufLen);
```

`BufLen` predstavlja dolžino bufferja, ki pa je `nil`, zato `GetAdaptersInfo` vrne napako. Nadaljevanje kode to napako ignorira. Juhej!

Ustvari / uniči

Vsaki dodelitvi kateregakoli sredstva naj sledi tudi uničenje tega sredstva. Vsak `GetMem` tako potrebuje `FreeMem`, vsak `Create` svoj `Destroy`, vsak `Acquire` svoj `Release` ...

Še najlepše je, če vsak blok ustvari/uniči zavijete v `try ... finally`. Potem se bo sredstvo zagotovo uničilo, tudi če znotraj bloka pride do izjeme (exception). Pa še struktura programa postane zaradi zamikanja bolj pregledna.

```
destructor TGpDVBTTeletextReceiver.Destroy;
var
  map: TDeviceMap;
begin
  Stop;
  for map in FInputToDeviceMap do begin
    map.Lock.EnterWriteLock;
    try
      map.List.Free;
    finally map.Lock.ExitWriteLock; end;
  end;
  inherited;
end;
```

Sproščanje sredstev nas pripelje do velike polemike: »`Free` ali `FreeAndNil`,« ki se na internetu razplamti vsaj dvakrat letno. Moje mnenje je zelo enostavno – vedno uporabim `FreeAndNil`, ker pri klicu `Free` v spremenljivki (polju) ostane neveljavna vrednost, ki lahko pripelje do zelo čudnih in težko ulovljivih napak, če tako spremenljivko kasneje po nesreči uporabimo.

Izjeme

Izjeme (exceptions) naj bodo namenjene za sporočanje izjemnih primerov, ko koda res ne ve, kako dalje, in ne za sporočanje normalnega stanja. Razloga za to sta vsaj dva – razhroščevanje programov, ki sprožajo izjeme, je naporno (o tem več v nadaljevanju), poleg tega pa iz definicije funkcije, ki jo kličemo, ne moremo izvedeti, katere izjeme utegne ta funkcija sprožiti in zato ne vemo, katere izjeme loviti.

Drugi primer se pojavi tudi v Delphijevem VCL. Klasičen primer je `TFileStream.Create`, ki lahko sproži izjemi `EFOpenError` ali `EFCREATEERROR`, a to lahko izvemo le iz dokumentacije ali iz kode v `System.Classes`. Zato jaz raje uporabljam svojo funkcijo za izdelavo tega objekta (*GpStreams.pas*).

```

function SafeCreateFileStream(const fileName: string; mode: word; var fileStream: TFileStream; var errorMessage: string): boolean; overload;
begin
  Result := false;
  errorMessage := '';
  try
    fileStream := TFileStream.Create(fileName, mode);
    Result := true;
  except
    on E: EFCreateError do
      errorMessage := E.Message;
    on E: EFOpenError do
      errorMessage := E.Message;
  end;
end;

```

Izredno slaba navada, ki jo lahko primerjamo s slavnim `On Error Resume Next` iz Visual Basica, je koda, ki ignorira izjeme s konstruktom `try ... except end`. Pa še vseeno sem primer take slabe rabe našel celo v svojem `OmniXMLUtils`:

```

if reg.ValueExists(value) then begin
  try
    Result := XMLLoadFromString(xmlDocument, reg.ReadString(value));
  except end;
end;

```

Ta koda bi morala loviti samo `ERegistryException`, pa še tega samo pri klicu `reg.ReadString`, ne pa med klicem `XMLLoadFromString`.

```

if reg.ValueExists(value) then begin
  try
    s := reg.ReadString(value);
  except
    on E: ERegistryException do
      Exit(False);
  end;
  Result := XMLLoadFromString(xmlDocument, s);
end;

```

Izjeme vedno lovimo čim bolj eksplisitno in ne kar »počez«. Če se pojavi neznana izjema na nepričakovanim mestu, je še vedno bolje, da priopruje do kode VCL in prikaže okno s sporočilom, kot da zaradi nje program deluje narobe in nič ne javi uporabniku (in morda spotoma še pokvari kakšne podatke).

Kadar program crkne zaradi neulovljene izjeme, je koristno shraniti tako imenovani »stack trace« (stanje izvajalnega sklada) vseh niti v programu. Na podlagi te informacije boste bistveno lažje našli napako. Najpogosteje uporabljeni dodatki, ki pomagajo pri pripravi in izdelavi te informacije, so *EurekaLog*, *madExcept* in prosti *JclDebug*.

Design by Contract

Varno programiranje pogosto povezujemo s konceptom *Design by Contract*, pri katerem načrtujemo program tako, da se *klicatelj* zaveže, kakšne podatke bo *poslal* funkciji, *klicani* pa, kakšne podatke bo *vrnil*. V programu prvo obljubo testiramo na začetku funkcije (*precondition*), drugo obljubo pa na koncu (*postcondition*).

```
function ByContract(value: integer): integer;
begin
    // test preconditions (value)

    // do something with the value

    // test postconditions (Result)
end;
```

Ker niti Delphi niti C++Builder ne podpirata tega koncepta v samem jeziku, obljube običajno testiramo s stavkom `Assert` ali s konstruktom `if <pogoj> then raise Exception.`

Nekaj primerov iz prakse, obljube so označene z rumeno.

```
procedure TGpBookmark.JumpTo;
begin
    Assert(not IsEmpty);
    gbDataSet_ref.GotoBookmark(gbBookmark);
end;

procedure TDecayTimer.SetSteps(const steps: array of int64);
begin
    Assert(Length(steps) > 0);
    SetLength(FSteps, Length(steps));
    Move(steps[0], FSteps[0], Length(steps) * SizeOf(steps[0]));
    Reset;
end;

procedure InitializeGlobals;
begin
    InitializeCriticalSection(GDSiWndHandlerCritSect);
    GTerminateBackgroundTasks := CreateEvent(nil, false, false, nil);
    GDSiWndHandlerCount := 0;
    GTimeGetTimeBase := 0;
    GLastTimeGetTime := 0;
    if not QueryPerformanceFrequency(GPerformanceFrequency) then
        GPerformanceFrequency := 0;
    GCF_HTML := RegisterClipboardFormat('HTML Format');
    DynaLoadAPIs;
    timeBeginPeriod(1);
    Assert(Length(DSiCPUIDs) = 64);
end;
```

Assert ali raise?

Odločitev, ali v takih testih uporabiti `Assert` ali `raise`, je odvisna od več faktorjev. Zavedati se morate, da bo prevajalnik stavke `Assert` ignoriral (ne bo jih prevedel), če ugasnete opcijo *Debugging, Assertions v Project, Options*. Če imate proces izdelave končnega produkta pod kontrolo in lahko zagotovite, da bo ta nastavitev ostala prizgana tudi ob izdelavi končne različice, potem je `Assert` dobra izbira. Sicer pa raje uporabite `raise`.

V obeh primerih pa priporočam dober opis napake. V ta namen ima `Assert` drugi parameter tipa `string`, ki se uporabi za vsebino izjeme, ki jo sproži `Assert` ob napaki.

```
Assert(not((cooCanRepair in options) and (createHeader = '')),
       'TCSVDB.Open: cooCanRepair is set, createHeader is not');
Assert([cooCanRepair,cooOpen_READONLY]*options <> [cooCanRepair,cooOpen_READONLY],
       'TCSVDB.Open: cooCanRepair and cooOpen_READONLY cannot be set at the same time');
if cooAllowDifferentStructure in options then
  Assert(cooOpen_READONLY in options,
        'TCSVDB.Open: when cooAllowDifferentStructure is set, cooOpen_READONLY must be
too');
```

Stavek »with«

Vsi vemo, da stavka `goto`, ki ga Delphi sicer podpira, ni priporočljivo uporabljati. Podobno velja v večini primerov tudi za stavek `with`.

Lep primer je že pred leti predstavil Maro Cantu, našel pa ga je kar v Delphijevem VCL.

```
with LMargins, GlassFrame do
begin
  if Enabled then
    begin
      if not SheetOfGlass then
        begin
          cxLeftWidth := Left;
          cxRightWidth := Right;
```

V takem sestavljenem stavku `with` nikoli ne vemo, na kateri objekt (`LMargins` ali `GlassFrame`) se nanašajo lastnosti in polja v nadaljevanju. Tudi razhroščevati je tak program težko, saj se ne moremo postaviti na (recimo) `Enabled` in pritisniti Ctrl+F7 (*Evaluate*). Dobili bomo namreč napako *Undeclared identifier: 'Enabled'*.

V težave lahko pridemo tudi pri bolj enostavnih kodih.

```
with component do
  Caption := 'test';
```

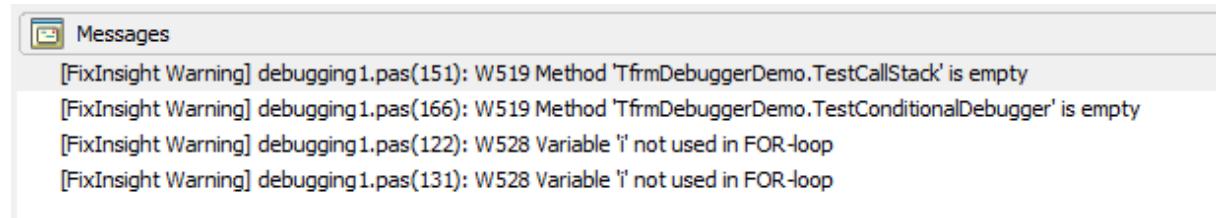
Zgornji vrstici nastavita lastnost `Caption` neki komponenti – vse do trenutka, ko se bo koda te komponente spremenila in komponenta ne bo imela več lastnosti `Caption`. Program se bo še vedno prevedel brez napake, le da bo nastavil to lastnosti na trenutno aktivni formi.

Splošni nasveti

Pri programiranju se splača držati tudi drugih pravil dobrega programiranja, kot so *Don't Repeat Yourself (DRY)*, *Single Responsibility Principle*, *Separation of Concerns* in *SOLID*.

Statična analiza

Veliko problematičnih delov v kodi zna najti orodje za statično analizo *FixInsight*, ki analizira izvorno kodo in pokaže na morebitna problematična mesta.



The screenshot shows a software interface with a window titled "Messages". Inside the window, there are four lines of text, each starting with "[FixInsight Warning]". The messages are:

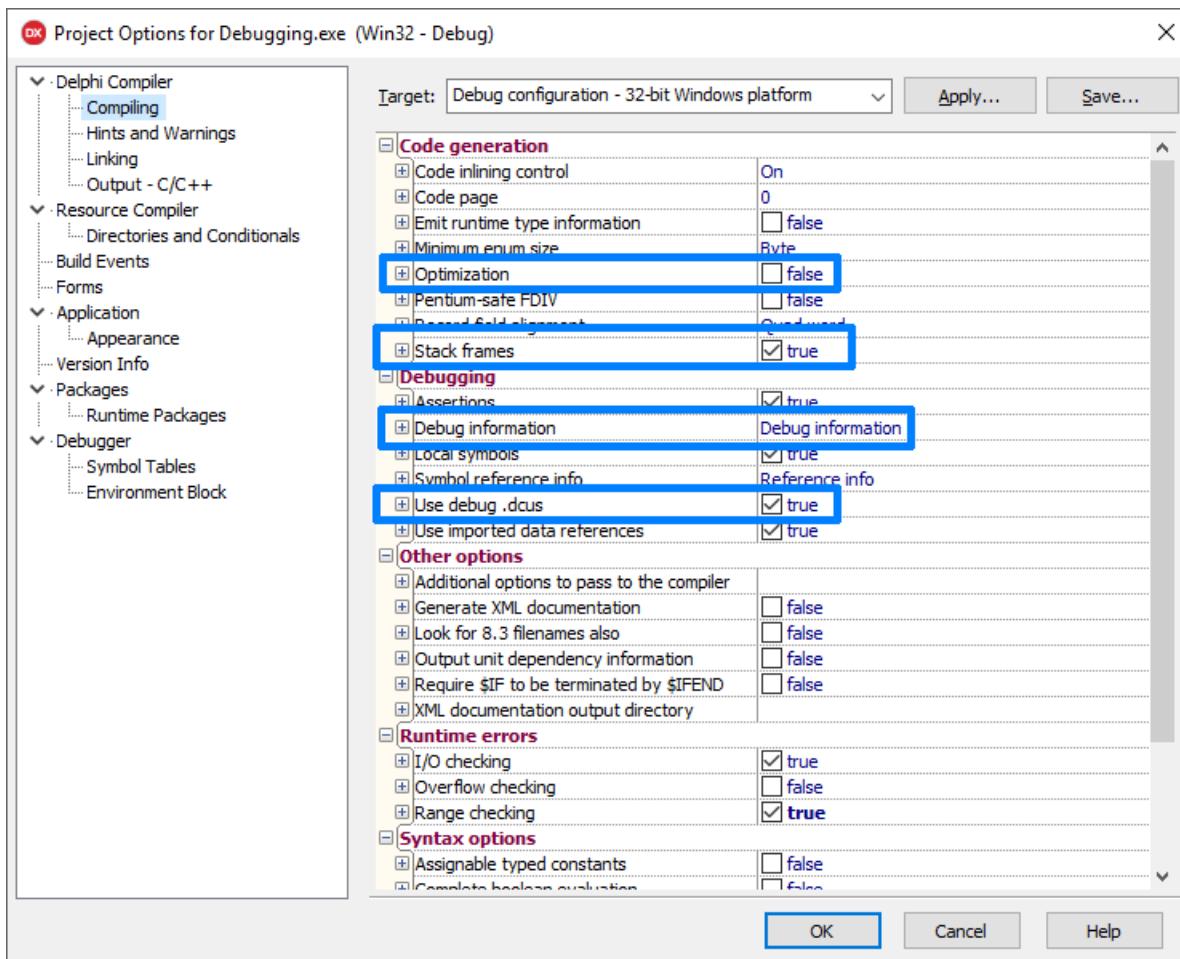
- [FixInsight Warning] debugging1.pas(151): W519 Method 'TfrmDebuggerDemo.TestCallStack' is empty
- [FixInsight Warning] debugging1.pas(166): W519 Method 'TfrmDebuggerDemo.TestConditionalDebugger' is empty
- [FixInsight Warning] debugging1.pas(122): W528 Variable 'l' not used in FOR-loop
- [FixInsight Warning] debugging1.pas(131): W528 Variable 'l' not used in FOR-loop

Napredno razhroščevanje

Ko se program sesuje ali pa ne deluje pravilno, je treba nekako najti razlog za napako. Večinoma to ni zelo težko, v nekaterih primerih pa se pošteno namučimo, preden se dokopljemo do pravega razloga. Takrat pomaga dobro poznavanje razhroščevalnika, vgrajenega v RAD Studio, ter nekaterih koristnih trikov.

Nastavitev

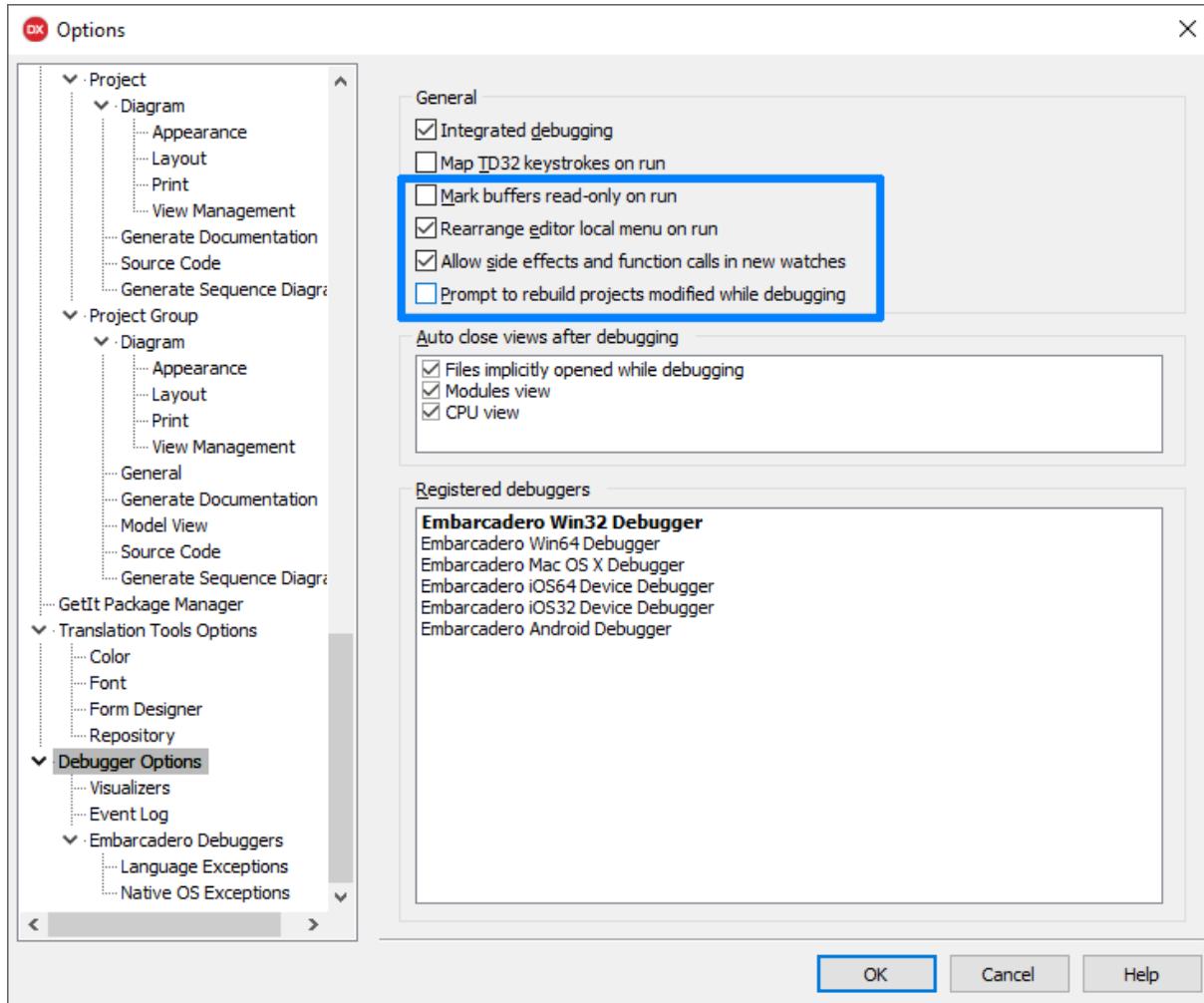
Pred razhroščevanjem se prepričajte, da ste program prevedli s podporo za razhroščevanje (*Debugging, Debug Information*), izključeno optimizacijo (*Code generation, Optimization*) in vključenimi okvirji na skladu (*Code generation, Stack frames*). Vse to najdete v projektnih opcijah (*Ctrl+Shift+F11*) v veji *Delphi Compiler, Compiling*. Včasih je koristno pokukati tudi v izvorno kodo, priloženo RAD Studiu, kar omogočite z opcijo *Debugging, Use debug .dcus*.



Veliko koristnih nastavitev, ki niso vezane na projekt, najdete v *Tools, Options* v veji *Debugger Options*. Priporočam, da si vključite *Rearrange editor local menu on run*, kar med razhroščevanjem programa predela pojavní menu editorja (desni klik), tako da so najbolj koristne funkcije na vrhu in *Allow side effects and function calls in new watches*, kar izboljša prikaz vrednosti v oknu *Watch List*.

Morda boste želeli še ugasniti *Prompt to rebuild projects modified while debugging* (ukine vprašanje *Source has been modified. Rebuild?*) ali prižgati *Mark buffers read-only on run* (prepreči spreminjanje

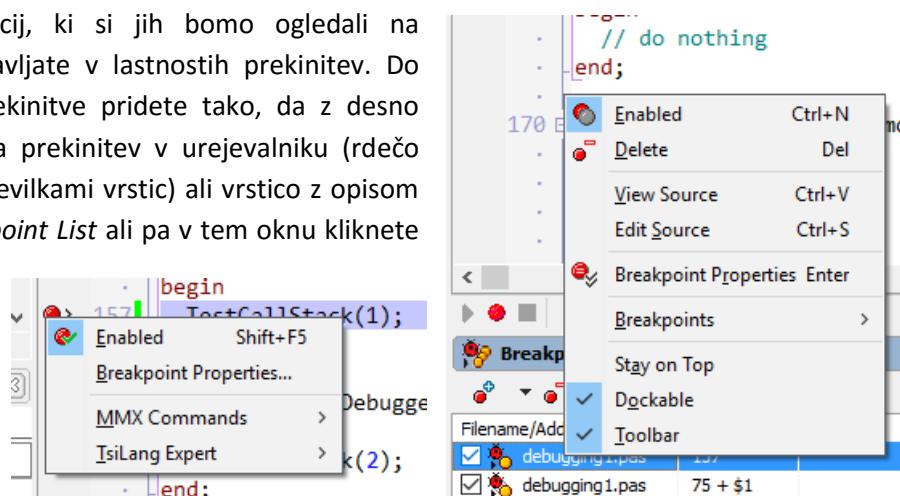
programske kode med razhroščevanjem). Uporabne so tudi izbire v skupini *Auto close views after debugging*.



Če želite zmanjšati množico sporočil, ki se med razhroščevanjem zapisujejo v okno *Event Log*, si oglejte možnosti v veji *Debugger Options, Event Log*.

Lastnosti prekinitvev

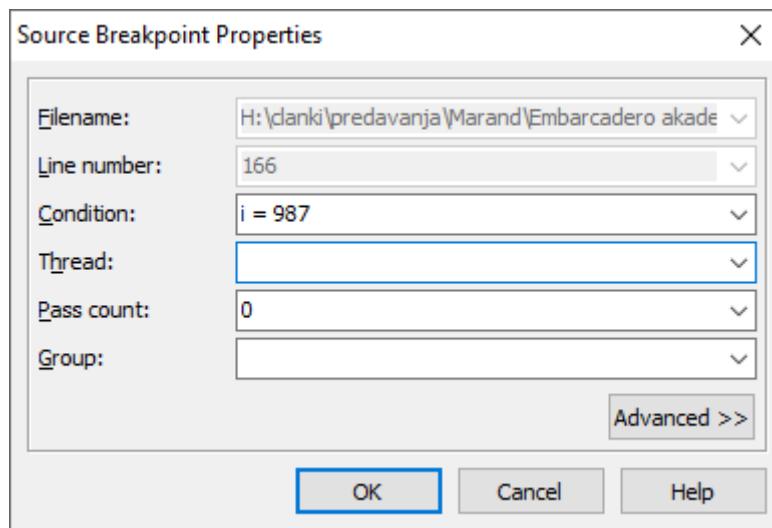
Večino naprednih funkcij, ki si jih bomo ogledali na naslednjih straneh, upravljate v lastnostih prekinitvev. Do lastnosti pozamezne prekinitve pridete tako, da z desno tipko kliknete oznako za prekinitvev v urejevalniku (rdečo piko levo od stolpca s številkami vrstic) ali vrstico z opisom prekinitvev v oknu *Breakpoint List* ali pa v tem oknu kliknete prekinitvev z levo tipko in pritisnete *Enter*.



Prekinitve lahko postavite s tipko *F5*, ali pa kliknete na mesto, kjer se pojavi rdeča pika. Na enak način prekinitve izbrišete. Manj znano pa je, da lahko prekinitve tudi prestavite na drugo mesto, tako da rdečo piko z miško premaknete drugam, ali pa tako, da v lastnostih prekinitve spremenite polje *Line number*.

Pogojne prekinitve

Vsaki prekinitvi lahko nastavite pogoj, pri katerem se bo prekinitve sprožila. V polje *Condition* okna z lastnostmi prekinitve vpišete logični pogoj in razhroščevalnik se bo na prekinitvi ustavi le, če bo pogoj izpolnjen (če bo rezultat izraza *True*).



Pogojne prekinitve so zelo uporabne zato, ker lahko pogoje proženja spreminjam med izvajanjem programa. So pa tudi zelo počasne. Če bo program več tisočkrat šel skozi prekinitve, preden bo pogoj izpolnjen, se bo to na hitrosti izvajanja zelo pozna.

Pogojno prekinitve lahko nadomestimo z enostavnim trikom – tako da jo vpišemo v kodo. (Seveda pa moramo za to program ustaviti, prevesti in ponovno pognati.)

V program vpišemo vrstici

```
if (pogoj) then
  Sleep(0);
```

nato pa na drugo vrstico postavimo običajno prekinitve.

```
procedure TfrmDebuggerDemo.
begin
  if i = 987 then
    Sleep(0);
```

Lahko pa si pomagamo z odprtokodno enoto *GpStuff* (<https://github.com/gabr42/GpDelphiUnits>), ki vsebuje funkcijo *DebugBreak*. Tej podamo pogoj, pri katerem naj se program ustavi v razhroščevalniku.

```
DebugBreak(i = 987);
```

Število ponovitev

Nastavimo lahko, po koliko ponovitvah naj se sproži prekinitve (lastnost *Pass count*). Če jo na primer nastavimo na 3, se bo razhroščevalnik ustavil vsakič tretjič, ko bo šel skozi prekinitve.

Ta možnost je zelo koristna tudi, kadar ne vemo, po koliko ponovitvah zanke program neha delovati. Poglejmo si to možnost na praktičnem primeru.

Iskanje napak med inicializacijo

Kadar naredimo napako v inicializaciji enote (v sekciji *initialization*) ali v zaključevanju enote (v sekciji *finalization*) jo je včasih zelo težko najti, ker razhroščevalnik noče pokazati mesta napake. V tem primeru si lahko pomagamo z lastnostjo *Pass count*.

Vključeno moramo imeti izvorno kodo sistemskih knjižnic (*Use debug .dcus*). Nato odpremo enoto *System* in poiščemo metodo *InitUnits* (če nastopi problem pri zagonu programa) ali *FinalizeUnits* (če nastopi problem med zapiranjem programa). (Mimogrede – enoto *System* najhitreje odprete tako, da nekam v urejevalnik natipkate *System* in pritisnete *Ctrl+Enter*. Potem pa odvečni *System* seveda pobrišete.)

V metodi *InitUnits* postavite prekinitve za vrstico *P := Table^ [I].Init* in ji nastavite *Pass count* na neko veliko številko (na spodnji sliki smo uporabili 999999). Program poženete in ko crkne, pogledate številko v polju *Pass count*. V našem primeru tam piše *175 of 999999*, kar pomeni, da je program nehal delovati v 175-ti ponovitvi zanke.

The screenshot shows the Delphi IDE interface. In the code editor, the cursor is at line 22737, which contains the assignment `P := Table^ [I].Init;`. This line is highlighted with a blue selection bar. The code is part of the `InitUnits` procedure. The code editor has a light gray background with syntax highlighting for keywords and identifiers. Below the code editor is a toolbar with standard IDE icons. To the right of the code editor is a status bar showing "22737: 1 Insert". Further down is a "Breakpoint List" window. This window has a header with buttons for "Code" and "History". The main area of the window shows a table:

Filename/Address	Line/Length	Condition	Thread	Action	Pass Count
System.pas	22737			Break	175 of 999999

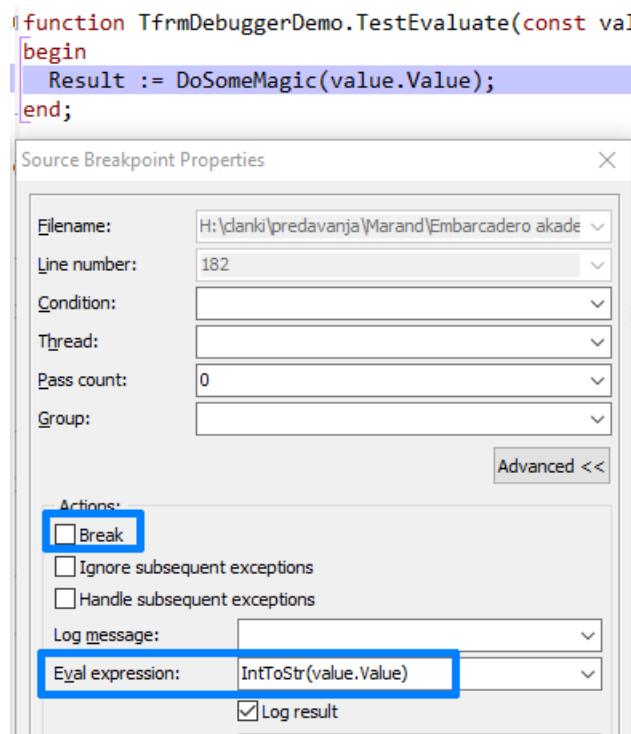
Zdaj popravite število ponovitev na 175 in ponovno poženite program. Ustavl se bo tik pred mestom napake in zdaj se lahko s tipkama *F8* in *F7* sprehodite do problematičnega mesta.

Tudi štetje ponovitev upočasni izvajanje programa.

Izpisovanje dogajanja

Včasih je težko ugotoviti, kdaj sploh pride do nepravilnega delovanja programa (še posebej takrat, kadar program deluje dalje, a narobe). Takrat pomaga, če si v bližini problematičnega mesta izpišemo nekatere vrednosti, po katerih lahko bolje ugotovimo, v katerih programih koda »nagaja«.

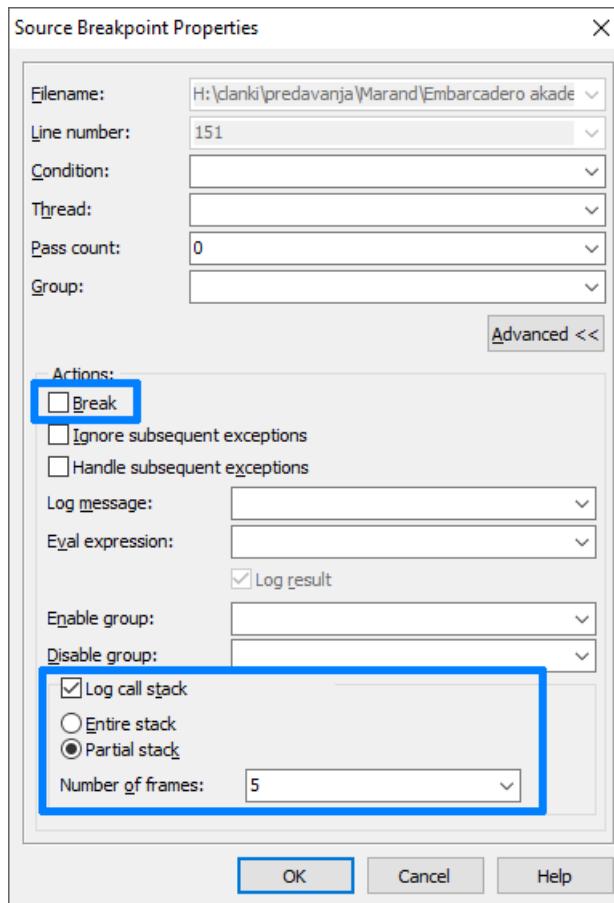
Vrednosti lahko izpisujemo v *Event log* kar z lastnostmi prekinitve. Ugasnemo *Break* (kar pomeni, da prekinitve ne bo ustavila izvajanja programa, temveč bo imela samo stranske učinke) ter v polje *Eval expression* vpišemo izraz, ki vrne niz (*string*). Vrednost se bo izpisala vsakič, ko bo program prišel do vrstice s prekinitvijo; preden se bo ta vrstica tudi zares izvedla.



Podobno kot pri pogojnih prekinitvah izpisovanje dogajanja upočasni program. Včasih je zato bolje, če spremenite kodo programa, tako da bo pisala neposredno v *Event log*.

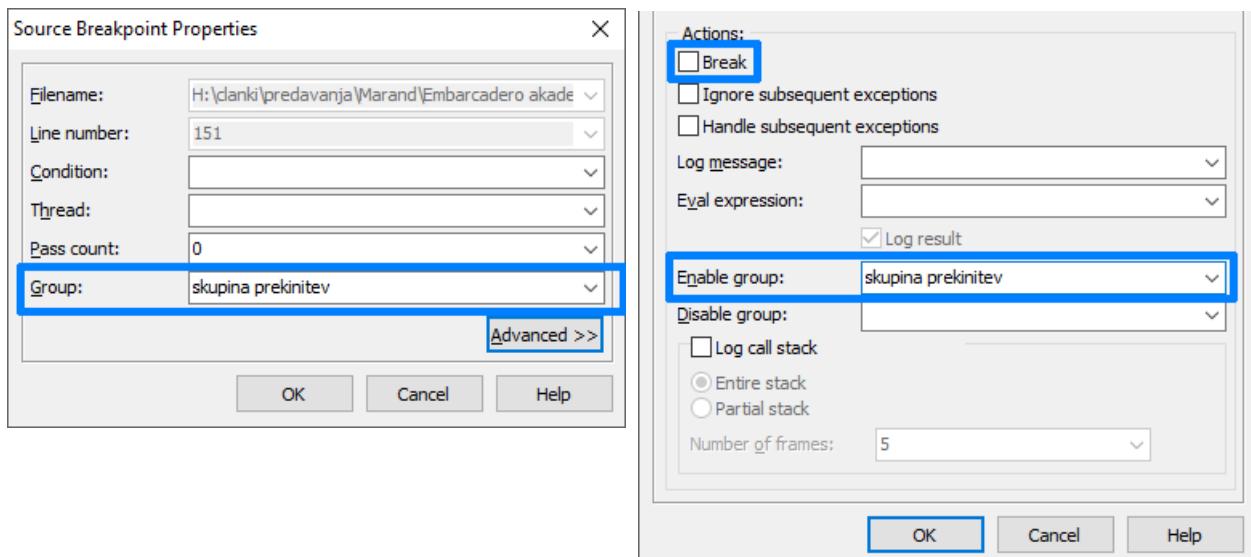
```
function TfrmDebuggerDemo.TestEvaluate(const value: IIntValue): real;
begin
  OutputDebugString(PChar(Format(' %d', [value.Value])));
  Result := DoSomeMagic(value.Value);
end;
```

V zelo redkih primerih pride prav tudi možnost izpisa metod, ki so pripeljali do trenutne točke (*Call stack*). To seveda tudi upočasni delovanje programa, a utegne posredovati informacije, do katerih drugače ne bi mogli priti.



Skupine prekinitiv

Prekinitve lahko ročno prižigamo in ugašamo (desni klik, *Enabled*; ali pa *Shift-F5*), še bolj koristno pa je, da lahko prekinitve združimo v skupine (*Group*), nato pa z drugimi prekinitvami te skupine prižigamo (*Enable group*) in ugašamo (*Disable group*).



Na ta način lahko nastavimo prekinitiv, tako da se bo sprožila samo takrat, kadar je koda klicana iz iz metode *A*, ne pa iz metode *B* in podobno.

Razhroščevanje večnitnih programov

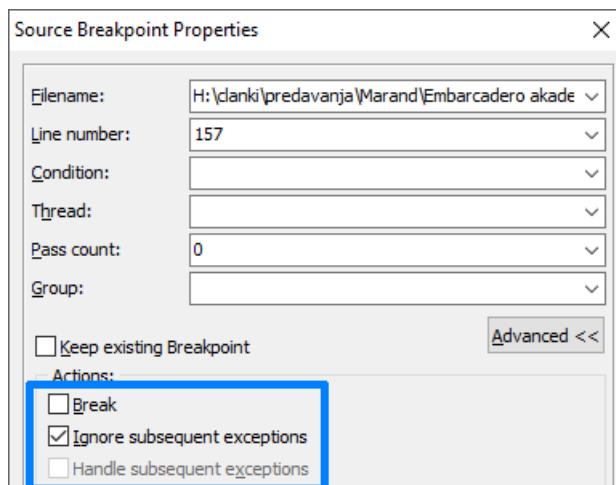
Pri razhroščevanju večnitnih programov pride prav vnosno polje *Thread*, v katerem nastavimo, da se prekinitve sproži le, če je koda izvedena v izbrani niti.

Kadar se sprehajate skozi večnitni program z F8 in F7 se velikokrat zgodi, da razhroščevalnik skoči v drugo nit. Temu se lahko izognete tako, da v oknu *Thread Status* z desno tipko kliknete niti, ki vas zanima, in izbere *Freeze All Other Threads* (zamrzni vse ostale niti). Seveda kasneje ne smete pozabiti teh niti spet »odtajati« (*Thaw All Threads*).

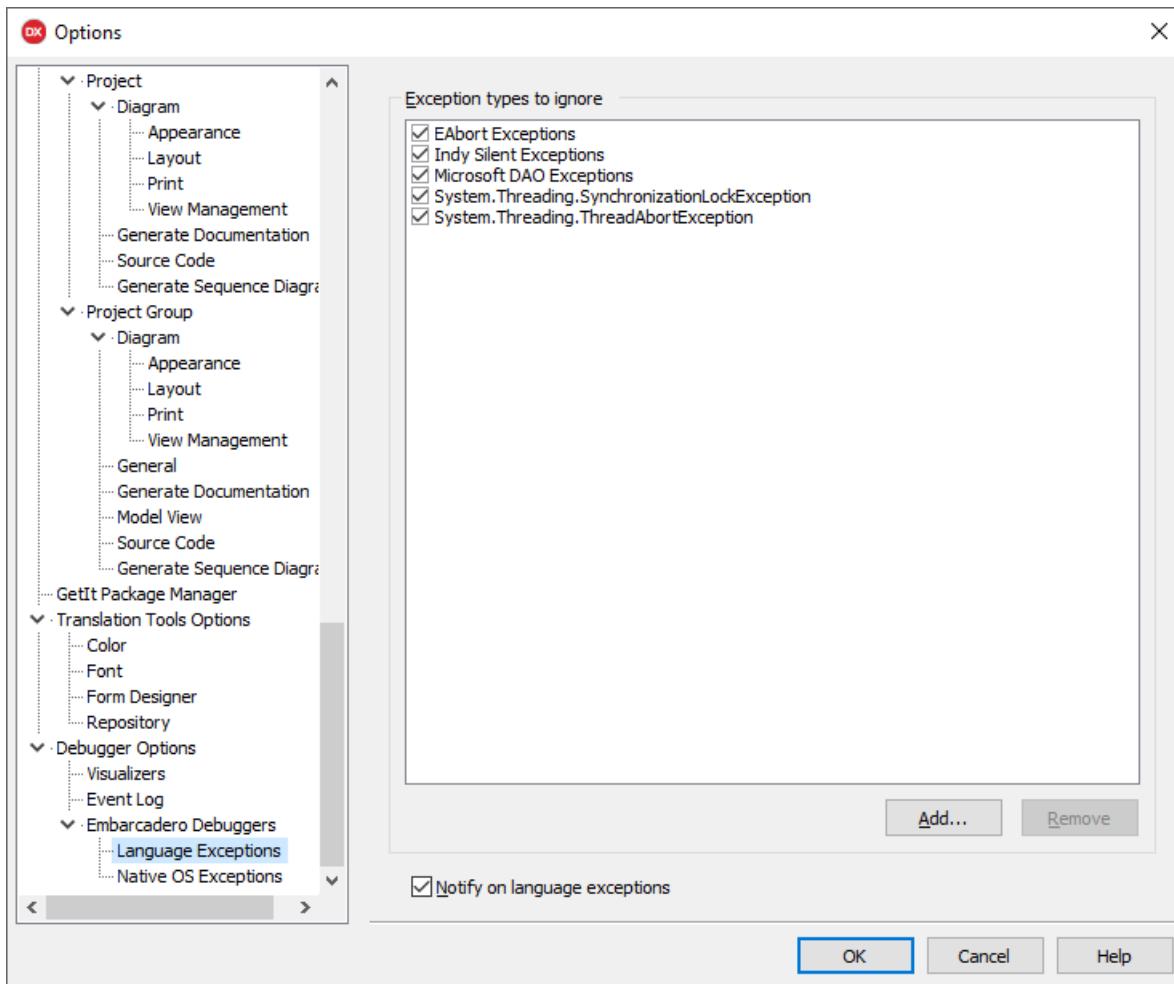
Izjeme

Razhroščevanje postane zapleteno, če koda proži izjeme. Te zaustavijo razhroščevalnik, tudi če jih kasneje program ujame in obdelva. RAD Studio pozna dva načina, kako to nerodnost zaobiti.

Prvi način je, da pred kodo, ki proži takšno izjemo, postavimo prekinitve s stranskim učinkom *Ignore subsequent exceptions*. Od tega trenutka dalje bo razhroščevalnik ignoriral vse izjeme. Za mestom, ko se izjema sproži, pa moramo postaviti še eno prekinitve s stranskim učinkom *Handle subsequent exceptions* (razen, seveda, če ne želimo, da nam izjeme ustavlja razhroščevalnik).



Drugi način je globalen. V *Tools, Options, Debugger Options, Embarcadero Debuggers, Language Exceptions* lahko nastavite izjeme, ki jih razhroščevalnik ignorira (se ne zaustavi, ko so sprožene). Lahko pa tudi globalno ugasnete zaustavljanje razhroščevalnika ob vseh izjemah, tako da izklopite nastavitev *Notify on language exceptions*.



Strojno podprtne prekinitve

Vse doslej omenjene prekinitve delujejo programsko. Ko v kodo postavite prekinitve, razhroščevalnik na mesto prekinitve postavi poseben ukaz v zbirniku, ki ustavi program. Kadar se prekinitve ne sprožiti (ni še doseženo nastavljeno število ponovitev, nastavljeni pogoj ne vrne *True*, prekinitve nima nastavljene možnosti *Break* ...) razhroščevalnik popravi program in nadaljuje z njegovim izvajanjem. Zaradi tega je izvajanje v razhroščevalniku pogosto počasno.

Intelovi procesorji pa poznajo tudi prekinitve, pri katerih sodeluje processor. Te prekinitve se skonfigurirajo preko posebnih registrov, zato jih lahko nastavimo le nekaj. Poznamo dve vrsti strojno podprtih prekinitrov – naslovne (*address*) in podatkovne (*data*).

Naslovna prekinitve se sproži, ko program izvede kodo na nastavljenem naslovu v pomnilniku. V bistvu gre za enak princip kot pri običajnih prekinitvah, le da lahko naslovne prekinitve nastavimo tudi na kodi, ki je ne moremo spremenjati.

Bolj zanimive so podatkovne prekinitve. Taka prekinitve se sproži, ko program piše podatke na nastavljeni naslov v pomnilniku. Uporabne so, kadar bi radi ugotovili, kje med izvajanjem programa se spremeni nek podatek.

Poglejmo si primer. Spodnja koda naredi testni objekt, izvede metodo *Run* in potem preveri, ali je neka vrednost objekta enaka 1. Ker ni, sproži izjemo.

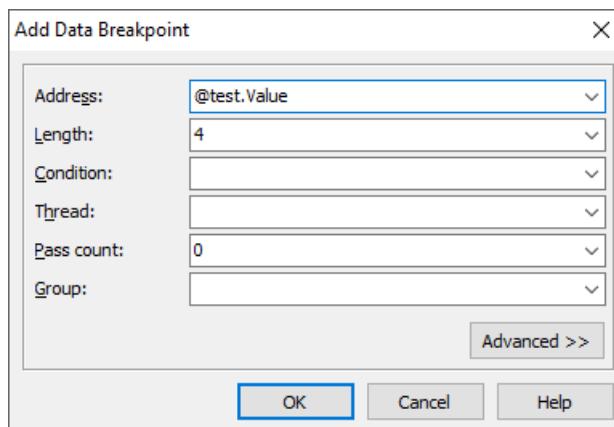
```

procedure TfrmDebuggerDemo.btnDataBreakpointClick(Sender: TObject);
var
  test: TTestObject;
begin
  test := TTestObject.Create(1);
  try
    test.Run;
    Assert(test.Value = 1, 'Test value <> 1');
  finally
    FreeAndNil(test);
  end;
end;

```

Zanima nas, kje v `test.Run` se ta vrednost spremeni. Podatkovne prekinitve lahko nastavljamo le med izvajanjem programa (kar je smiselno, ker šele takrat vemo, kateri naslov nas zanima), zato postavimo običajno prekinitve na `try` in poženemo program. Ko se ustavi, nastavimo podatkovno prekinitve, tako da z desno tipko kliknemo v prazen del okna *Breakpoint List* in izberemo *Add, Data Breakpoint*.

Prekinitvi moramo nastaviti naslov, ki nas zanima, ter velikost podatka, ki se zapisuje. V našem primeru je to naslov vrednosti `test.Value`, oziroma `@test.Value`. Dolžina podatka je štiri bajte (*integer*).

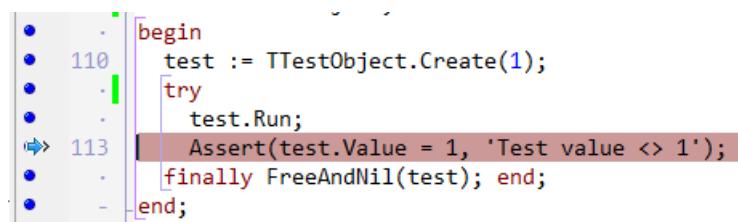


Program poženemo dalje in ustavl se bo na kodi, ki spremeni vrednost `test.Value`.

Podatkovne prekinitve se onemogočijo ob vsakem zagonu programa in jih moramo ročno omogočiti, kadar jih potrebujemo.

Premikanje točke izvajanja

Med razhroščevanjem je vrstica, ki se bo izvedla naslednja, označena z modro puščico.

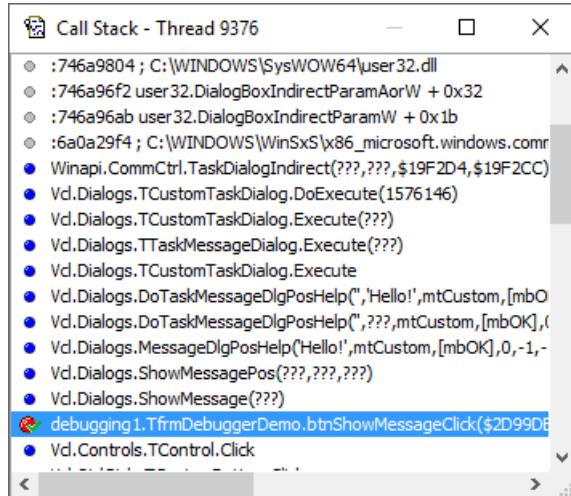


To modro puščico lahko z miško potegnemo na drugo mesto in s tem spremenimo točko, od katere se bo nadaljevalo izvajanje. Isto lahko storimo z desnim klikom v urejevalnik in izbiro možnosti *Set*

Next Statement. Pri tem morate biti previdni – točke izvajanje ne smete premakniti iz trenutne metode, pa tudi ne iz trenutnega bloka *try*.

Prekinitve na skladu izvajanja

Prekinitve lahko nastavite tudi v oknu s skladom izvajanja (*Call Stack*). Taka prekinitve se bo sprožila, ko se bo izvajanje programa vrnilo v metodo, ki ste ji nastavili prekinitve. Prekinitve nastavite z desnim klikom ali s pritiskom tipke *F5*.

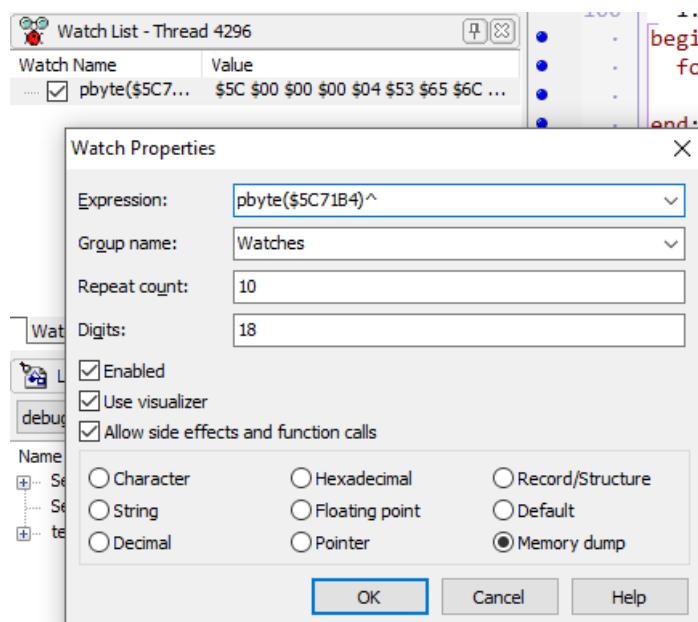


Vrnitev iz metode

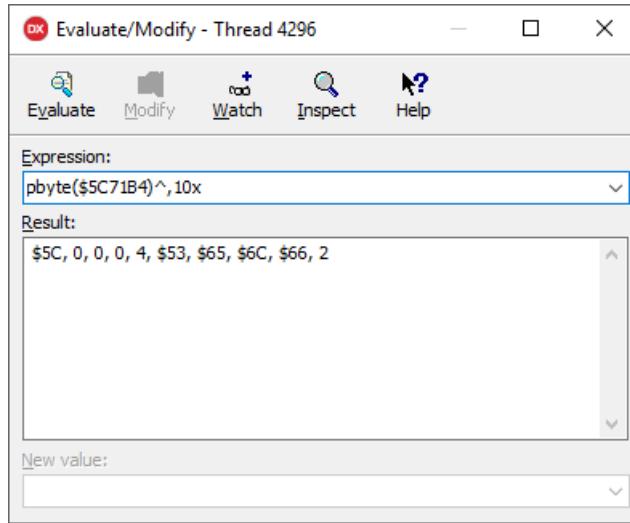
Ob pritisku na *Shift-F8* bo razhroščevalnik izvedel trenutno metodo do konca, se vrnil v klicatelja in se tam zaustavil. Bližnjica je zelo koristna, kadar pri vključenem *Use debug .dcus* zaidete v metodo sistemsko knjižnice in bi se radi vrnili iz nje.

Oblikovanje prikaza podatkov

Medtem ko v oknu za spremiščanje vsebine spremenljivk (*Watch List*) lahko spreminjamo način, kako so prikazani podatki (spodnja skupina radijskih gumbov), pa okno za izračun (*Evaluate/Modify*, *Ctrl+F7*) te možnosti nima.



V oknu *Evaluate/Modify* lahko isto funkcionalnost dosežemo tako, da dodamo na konec izraza vejico in določilo, ki spremeni način prikaza. Zanimivi sta predvsem možnosti »x« ki prikaže vrednosti v šestnajstškem zapisu in »r«, ki prikaže imena polj v razredih in zapisih. Pred »x« lahko dodate tudi številsko vrednost, ki določi, koliko elementov naj se izpiše. Enak rezultat, kot je na zgornji levi sliki (*Repeat count = 10, Memory dump*) lahko v oknu *Evaluate/Modify* dobimo z določilom formata »10x«.

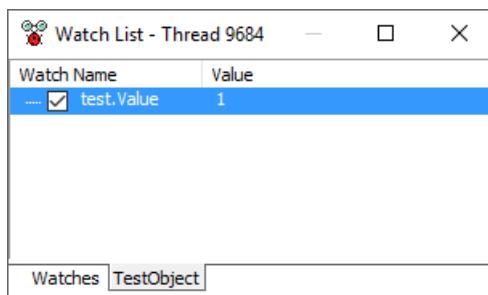


Mimogrede, enaka določila formata lahko uporabimo tudi v oknu *Watch List*.

Organiziranje opazovanih vrednosti

Okno *Watch List* omogoča organizacijo opazovanih izrazov v več skupin, ki so v uporabniškem vmesniku videti kot jezički. To omogoča boljši pregled nad opazovanimi izrazi.

Skupino moramo najprej dodati, tako da z desno tipko kliknemo v okno *Watch List* in izberemo *Add Group*. V oknu se pojavi nov jeziček. Kadar začnemo opazovati nov izraz, se bo dodal v trenutno aktivno skupino. Izraze lahko premikamo med skupinami, tako da z desno tipko kliknemo izraz in izberemo *Move Watch to Group*.



Prelisičite optimizator

Kadar razhroščevalnik noče prikazati vrednosti kakšne spremenljivke ali polja, ker je optimizator preveč agresivno opravil svoje delo, lahko v kodo dodate klic metode `DontOptimize`.

```
procedure DontOptimize(var data);
begin
    // do nothing
end;
```

Če, na primer, okno za prikaz vrednosti ne bi hotelo prikazati polja `test.Value`, bi v kodo na neko mesto dodali klic `DontOptimize(test.Value)`.

Viri

Vse o enumeratorjih

<http://www.thedelphigeek.com/2007/03/fun-with-enumerators.html>

Orodja

ModelMaker Code Explorer

<http://www.modelmakertools.com/code-explorer/index.html>

GpDelphiUnits

<https://github.com/gabr42/GpDelphiUnits>

MadExcept

<http://madshi.net/madExceptDescription.htm>

EurekaLog

<https://www.eurekalog.com/>

JCL

<http://jcl.delphi-jedi.org/>

JclDebug

http://wiki.delphi-jedi.org/wiki/JCL_Packages

FixInsight

<http://sourceoddity.com/fixinsight/>

Principi dobrega programiranja

DRY – Don't Repeat Yourself

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Single responsibility principle

https://en.wikipedia.org/wiki/Single_responsibility_principle

Separation of concerns

https://en.wikipedia.org/wiki/Separation_of_concerns

SOLID

[https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

Razhroščevanje

Možnosti razhroščevalnika

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Debugger_Options

Postavljanje in nastavljanje prekinitiv

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Setting_and_Modifying_Breakpoints

Okno sklada izvajanja

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Call_Stack_Window

Določanje oblike izpisa

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Evaluate_Modify#Display_Format_Specifiers

Oddaljeno razhroščevanje

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Overview_of_Remote_Debugging