embarcadero
Developer Camp

【T5】Delphi/C++Builderテクニカルセッション

「バグを生まないための開発技法」

山本　聡

embarcadero
Developer Camp

本日

震災

- 被災地の方、ご家族ご親類が震災に遭われた方には、大変な心労重なる時期かと思います。お見舞い申し上げます。
- みんなで復興しましょう。
- kizuna311.com

- 関東近郊にお住みの皆さん。
  - 雨が降ろうが、雪が降ろうが、原発だろうが、槍が降ろうが、何があろうとも、東京が**被災地救済の前線基地です。**
  - 我々は日本のビジネスを支える基盤です。
  - ここ東京をしっかり守りましょう。

- よく働き、いい仕事をして、いい生産物を生み出し、経済を盛り上げて復興支援をしましょう。

embarcadero
Developer Camp

このセミナーを聞く心構え

世の中の嘘とホント

- 世の中に様々な情報が存在します。

- 正しい情報と、嘘の情報、
- 役に立つ情報と役に立たない情報、
- 常に使える普遍的な情報、その場でしか使えない使い捨ての情報。

- 情報には正誤の違いがあり、そして重要度も違います。

- 専門家や権威と言われる人が意図的あるいは無意識に、偽りの情報を発信します。正しい情報と偽りの情報を見分けられたらいいですね。
- 見分ける為にはよく観察して判断する事が大事。

- だから、今日の内容は鵜呑みにせずに、自分で検証し、しっかりと自分の頭で考えて判断して使ってみてください。

embarcadero
Developer Camp

本題

「バグを生まないための開発技法」

「バグを生まないための開発技法」

- 【A5】Delphi/C++Builderテクニカルセッション
- 「アプリケーション品質の向上は永久の課題。バグを生まないための開発技法の導入は、誰もが求めるところです。
- しかし、意外に盲点なのが、ツールやコンパイラに用意された機能の活用。
- 品質向上を支援するツールをうまく活用すれば、コードにバグが忍び込むのを防ぐことができます。
- このセッションでは、テストと例外を使って、バグを抑止する開発の方法をデモを交えて紹介します。」

バグを生まないためには何をすればよいか

- **バグを見つける**仕組みを導入しよう。

- コンパイルする
  →当たり前。**コンパイルが通らなければバグ**があることはわかる。
- Exeを動作確認する。
  →当たり前。**Exeが動かなければバグ**があることはわかる。
- 関数(もしくはクラスのメソッド)を動作確認する。
  →当たり前。**関数を呼び出して動かなければバグ**があることはわかる。
  アプリケーションの中で一度は関数を呼び出している箇所があるよね。

でも、実際にはバグが混入していてもわからない場合がある。
アプリケーション中に特定の呼び出し方法が使われていない場合がある。
だから、バグを生まないためには

- **関数を動作確認するためにはテストを書く**。
  →これを**当たり前のレベル**にする。

だって、関数のテストがなければ正しく動くかどうかわからない。

embarcadero
Developer Camp

「バグを生む」典型例

典型例:年齢計算コード

- ある日あの時あの場所での出来事

- DelphiML/AboutDelphiに載っている年齢計算ソースコードを使わせていただきました。
- リリース後、数ヶ月前には問題なく動いていたが、どうやら近頃になって突然動かなくなったという報告が入りました。
- なんで？なんで？なんで？どこが悪いのか分からない。
- 現地に行き、トレース実行するまでわかりませんでした。

- 閏日2月29日生まれの人が登録された時だけ例外がでてしまい、例外発生を予想してなかったので思わぬところでシステム停止。

- ほんと、すいません。
「俺は悪くないっすよ」なんて言ってごめんなさい。
- テストをしてない俺が悪い

典型例:PutYenMark

- ファイルパスの最後に[¥]が付属していようといまいと[¥]を付属させる関数、IncludeBackSlashが、Kylix対応でIncludePathDelimiterと名前が変わっていった、あの時代。
- ファイルパスの区切り記号は、日本語版Windowsでは、￥マークだが、英語版ではバックスラッシュだと初めて気がついたのも、この時でした。

- そしてその関数は、もう日本の有志が作っていました。

FDelphi PutYenMark

- 使ってみると、なんか正しくファイルパスが指定できない場合があるようだぞ。なぜだ？
  - 半角[¥]のSJIS文字コードは[5C]
  - 全角[ソ]のSJIS文字コードは[835C]

- だから、パス名の最後に[ソ]がくる場合誤動作します。
  - SJISの時だよ

典型例：TMemIniFile

- 私の関わったVBで作られたあるシステムが、起動時設定をIniファイルで読み込んでいました。
  - Iniファイルを手動で修正する仕様です。
  - Iniファイルには項目がたくさんありすぎて、設定がめんどくさい。
  - Item=1と2と3の意味の違いがわからなくて困る。

- そこで私はDelphiでIniファイル設定専用アプリをすぐに作りました。
  完成してしばらく動かした後に、気がついた時にはIniファイル内にコメントでかかれていた大量の説明のためのドキュメントがなくなっていたことに気がつきました。

- ありえない！

- TMemIniFileはIniファイルのコメント行仕様をなぜか無視します。
  IniFileの仕様を確認するテストを通していないからなのか？

典型例：TStringList。改行コードの取扱い

```
Str1 := 引数から指定する。
StrList1 := TStringList.Create;
  StrList1.Text := Str1;
  Str2 := StrList1.Text;
  if Str1 = Str2 then
    ShowMessage('EQUAL')            ……A
  else
    ShowMessage('NOT EQUAL');       ……B
StrList1.Free;
```

- あなたがビジネスで成功するには、Aに賭ける？Bに賭ける？
- 引数から指定したStr1が
  - Str1 := "abc"#13#10の時はEQUAL
  - Str1 := "abc"の時はNOT EQUAL
- これは仕様か？それともバグか？

embarcadero
Developer Camp

「バグの芽を摘む」

ソースコードを高品質にするためのソースコード

> バグを見つける仕組みを導入しよう。
> 関数を動作確認するためにはテストを書く

- どのようにテストを書くべきか。
- 実に簡単。わずか6行です。(実質1行)

```
procedure Check(A, B: String);
begin
  if (A<>B) then
    ShowMessage('間違っているよ。'+#13#10+
      'A='+A +#13#10+"B="+B);
end;
```

- これだけで、あなたのプログラムは高品質になる！！
  - かもしれません
  - ならないかもしれません。

現代風汎用的に改良

- 現代風にFormatを使い、テストコードにスピードは求められないので、Variant化してより汎用的にしてみましょう。

```
procedure Check(A, B: Variant);
begin
    Assert(A = B,
      Format("%s"#13#10"A=%s"#13#10"B=%s",
             ["間違っているよ。", A, B]) );
end;
```

- これで、IntegerやDateTimeなど他の型に対しても値をチェックできます。

- RectやPoint型に対しては、Variantは効かないので、必要な場合は自作してください。

年齢計算

- 年齢計算用コードの内部の正しい実装例は紹介しません。
- ネットを検索すれば出てきますし、
  あなたが工夫すればたぶん書けます。

重要なのは！

テストの書き方

年齢計算

```
function GetAge(BirthDay, CheckDay:TDateTime):Integer;
begin  …省略… end;

procedure testGetAge;
begin
//1992年は閏年。2/28･2/29･3/1誕生日の人は、翌年の2/27･2/28･3/1に
//それぞれ何歳になるか総当たりでチェックする。

  //閏年が誕生日 と 閏年じゃない年 との年齢
  //     2月28日生まれの人は、2月27日まだ0歳、2月28日で1歳
  Check(0, GetAge('1992/02/28', '1993/02/27'));
  Check(1, GetAge('1992/02/28', '1993/02/28'));
  Check(1, GetAge('1992/02/28', '1993/03/01'));

  //     2月29日生まれの人は、2月28日まだ0歳、3月1日で1歳
  Check(0, GetAge('1992/02/29', '1993/02/27'));
  Check(0, GetAge('1992/02/29', '1993/02/28'));
  Check(1, GetAge('1992/02/29', '1993/03/01'));

  //     3月1日生まれの人は、2月28日まだ0歳、3月1日で1歳
  Check(0, GetAge('1992/03/01', '1993/02/27'));
  Check(0, GetAge('1992/03/01', '1993/02/28'));
  Check(1, GetAge('1992/03/01', '1993/03/01'));
```

年齢計算

```
//2008年も閏年。1992/2/28･2/29･3/1誕生日の人が、2008年の2/27･2/28･2/29･3/1に
//それぞれ何歳になるか総当たり

  //閏年が誕生日 と 閏年 との年齢
  Check(15, GetAge('1992/02/28', '2008/02/27'));
  Check(16, GetAge('1992/02/28', '2008/02/28'));
  Check(16, GetAge('1992/02/28', '2008/02/29'));
  Check(16, GetAge('1992/02/28', '2008/03/01'));

  Check(15, GetAge('1992/02/29', '2008/02/27'));
  Check(15, GetAge('1992/02/29', '2008/02/28'));
  Check(16, GetAge('1992/02/29', '2008/02/29'));
  Check(16, GetAge('1992/02/29', '2008/03/01'));

  Check(15, GetAge('1992/03/01', '2008/02/27'));
  Check(15, GetAge('1992/03/01', '2008/02/28'));
  Check(15, GetAge('1992/03/01', '2008/02/29'));
  Check(16, GetAge('1992/03/01', '2008/03/01'));

  //閏年じゃない年が誕生日 と 閏年じゃない年 との年齢 省略
  //閏年じゃない年が誕生日 と 閏年 との年齢 省略
end;
```

年齢計算

- Button1Clickなどから、このprocedure testGetAgeを呼び出してください。
- GetAgeが正しい場合は、何も起きません。
- GetAgeが誤っている場合は、例外が出たり、落ちたり、メッセージがでてわかります。
- 普通の日同士の年齢計算の結果チェックだけではなく、要注意ポイントをテストしてください
- 年齢計算の場合、要注意ポイントは、閏年。年齢計算コードの場合には誕生日が閏年の閏日をテストしないわけにはいきません。
- テストをしていなかったから、例外発生して不具合になる事がわからなかったのです。

PutYenMark

- PutYenMarkの場合は、誤動作する事を知らなければ、事前にはテストを書けません。
- ですから、SJIS文字コードの誤動作に気がつかない限りは、テストはこのようなものになるでしょう。

```
Check('C:¥',                PutYenMark('C:'));
Check('C:¥test¥',    PutYenMark('C:¥test'));
Check('C:¥test¥',    PutYenMark('C:¥test¥'));
Check('C:¥test¥あ¥',        PutYenMark('C:¥test¥あ'));
Check('C:¥test¥あ¥',        PutYenMark('C:¥test¥あ¥'));

Check('¥',           PutYenMark(''));
```

- 空文字に関しての仕様は曖昧になりがちなので、テストを記述しておくのがいいでしょう。
(※FDelphiのPutYenMarkは空文字には対応してなさそうです。)

PutYenMark

- 誤動作に気がついた時点でテストを増やしましょう
  - [¥]のSJIS文字コードは[5C]
  - [ソ]のSJIS文字コードは[835C]
  - なので、パス名の最後に[ソ]がくる場合誤動作します。

```
  Check('C:¥test¥ソ',          PutYenMark('C:¥test¥ソ'));     //★
//  Check('C:¥test¥ソ¥',       PutYenMark('C:¥test¥ソ'));
  Check('C:¥test¥ソ¥',         PutYenMark('C:¥test¥ソ¥'));
```

- 最後に¥マークが付属しない仕様は変だけど、仕様だとしたら、このように明示的に知らせるのがよいでしょう。
- もしくは、PutYenMarkの内部実装を変更して正しい実装をしましょう。

TMemIniFile

- テストは単純なのですが、量は膨大なので掲載しません。
  - TAccessIniFileの実装とテストコードは近日公開しておきます。
- WindowsAPIをラップしているTIniFileの挙動をテストコードを作りながら確認して、そのテストを通過するように、TAccessIniFileを作りました。
- テスト内容は、＝が複数個あったり、コメントの残し方、インデントがある時、ダブルクウォーテーションの扱い(複数個連続して値中にある場合など)、セクションやキーの削除、などなどです。
- TIniFileとTMemIniFileの動作が異なる事がテストを書く事で明らかになります。
- 同等のモノが存在するかもしれませんが、部品の品質を確認するためにもテストコードは重要です。
- TIniFileを旧バージョン、TAccessIniFileを新バージョンを見なすと、バグの出ない新バージョンクラスを作りたい場合には、テストを書くと挙動に違いをなくすことができる。という実装例になるでしょう。

TStringListの改行問題

- 考えてテストを書いてみてください。
- なぜそうなるのか理解できない仕様でも
  明示的にテストとして書かれてあると読むときにわかりやすいですよ。
- Pos関数に空文字を指定してみたり。
  TStrings.IndexOfが空文字や大小文字区別するのかどうか調べておいたり。
  例をあげればきりがないです。
- 全ての関数、メソッドに対して、Check関数の挙動をドキュメントに記述して欲しいくらいです。

embarcadero
Developer Camp

テストコードを書くのが日常に
なってからというもの・・・

このテクニックを知ってからの展開。

- 自分のソースの品質を、自分自身が信頼できるようになりました。
- 様々な文字列変換関数を作ってもバグ無しで仕上げられます。
- TStringListのそっくりクラスでCR/LFを完全に処理できるようになものを作ってもバグ無しで仕上げられます。
- 任意桁指定での四捨五入関数を自作しても問題なし。

- 複雑な「TreeViewのItemセット処理」を修正する場合は、既存処理の項目値をテキストで保持しておき、新しい処理に書き直して、古い処理と出力結果をテキストで比較。出力結果に差が無いことを確認してコードを書けば、バグは入りません。
-

- どんなコードを書いてもバグを作る事は全くなくなりました。(嘘)
  - バグは時々生み出しますが、コードの品質は高まりました。

- 品質のよいプログラムを組む為には必須の技術です。

プログラムを作る上で大切なのは

- 重要なのはスピードより品質。正しく動く事。
- なぜならば、遅くても、しばらく待っていればよいし、ハードウェアの改良でソフトウェアが改善する余地があります。ですが、品質が低くてエラーになって動作不良になれば、そこで終了。
- そして品質のために重要なのは、実装よりもテストコード。

  なのかもしれません。

- 相対アドレスパスを絶対アドレスパスと相互変換する関数はDelphiテクニック(もしくはWindowsAPIのテクニック)としてネットで紹介されていますが、品質が確保されていないために作り直しました。
  - 実装は大したロジックじゃないです。
  - 品質を確保するのに重要なのはテストコードでした。

embarcadero
Developer Camp

DUnit

テストするには、なんとかUnit

- テストするには、ユニットテストを使おうという風潮があるようです。JavaのJUnitや、.NETのNUnit、他に何があるかよくわかりませんが、
- Delphiならば、DUnit。
- 単なるCheck関数の亜種です。
  お化けのように肥大化したCheck関数。
  - 使わなくても、テストは出来るけれど
  - 使った方が「俺は高度なテストが出来る」と見せかけられるかも。
- IDEから使えます。

TextCalcクラス、テストコード

```
//Unit1.pasの内部

uses Unit2;

procedure TForm1.Button1Click(Sender: TObject);
var
  TextCalc: TTextCalc;
begin
  TextCalc := TTextCalc.Create;

  Check(2, TextCalc.Calc('1+1'));
  Check(5, TextCalc.Calc('2.5+2.5'));

  TextCalc.Free;
end;
```

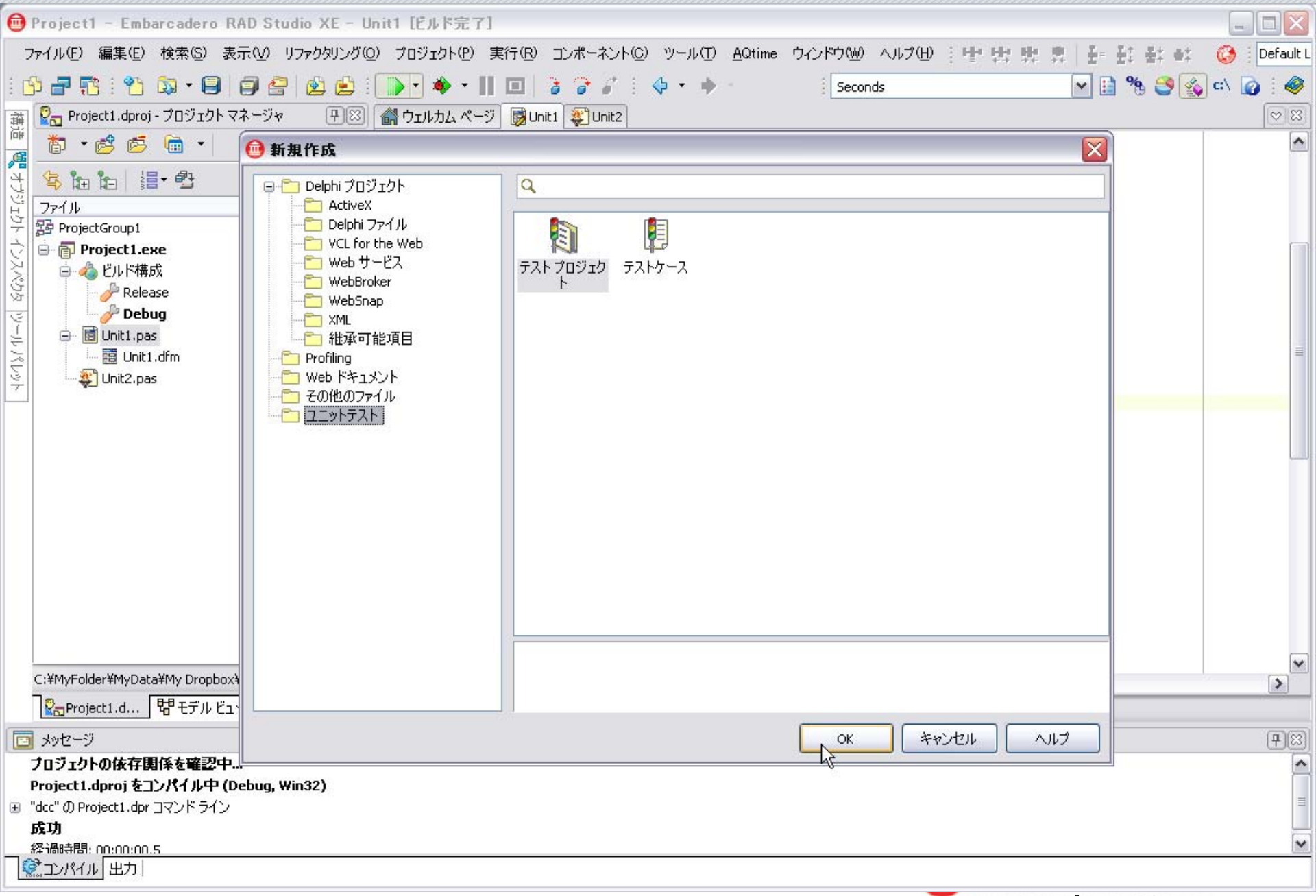
- 「1+1」や「2.5+2.5」という**数式を文字列**を渡すと**結果が数値**で戻るクラス。
- これは！もしや！

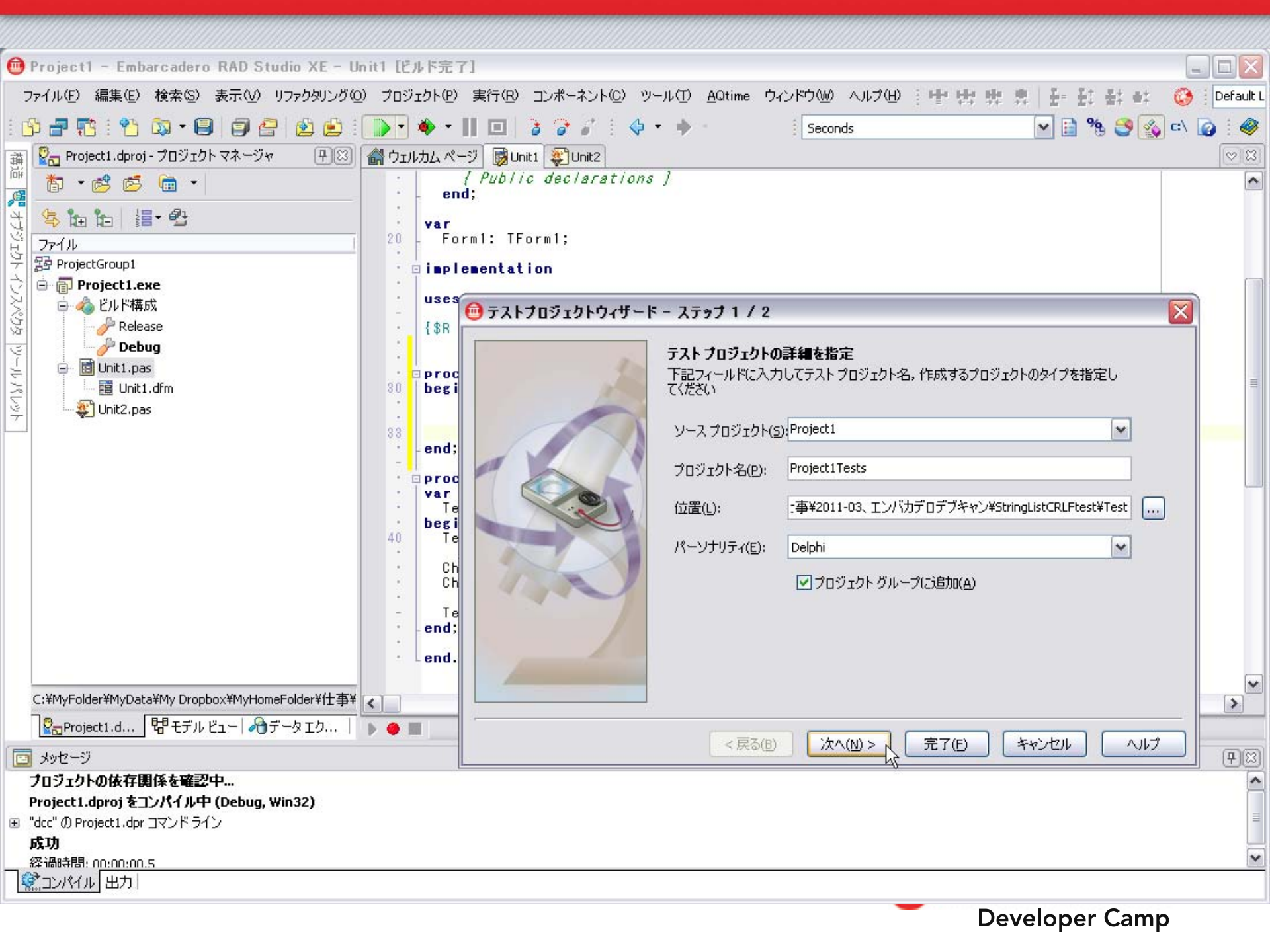
TextCalcクラス、実装

```
//Unit2.pasの内部

type
  TTextCalc = class
  public
    function Calc(Text: String): double;
  end;

function TTextCalc.Calc(Text: String): double;
begin
  if Text = '1+1' then result := 2.0;
  if Text = '2.5+2.5' then result := 5.0;
end;
```

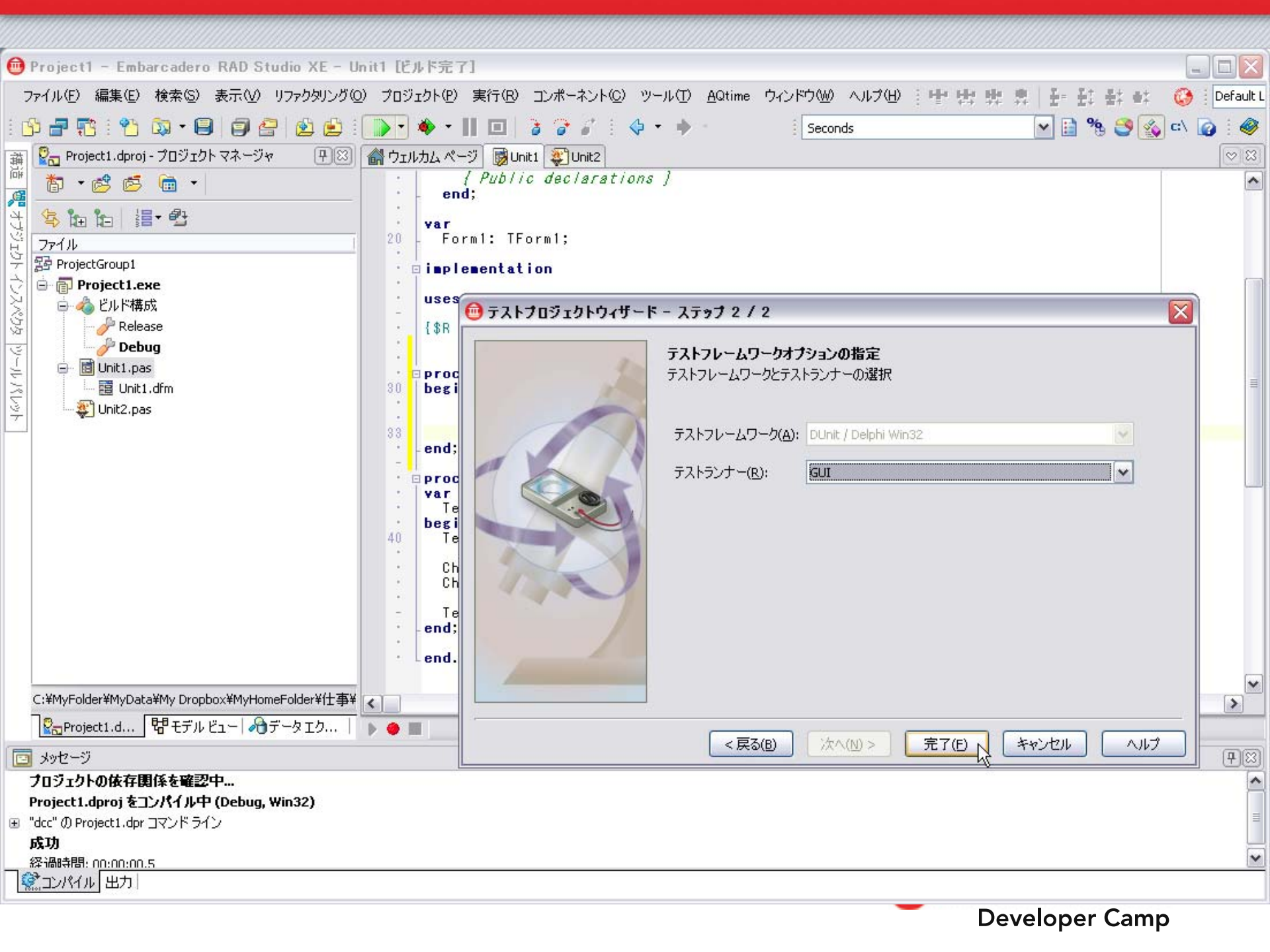
- 残念！
  偽クラスでした。

- 単にテストの実験に使います。

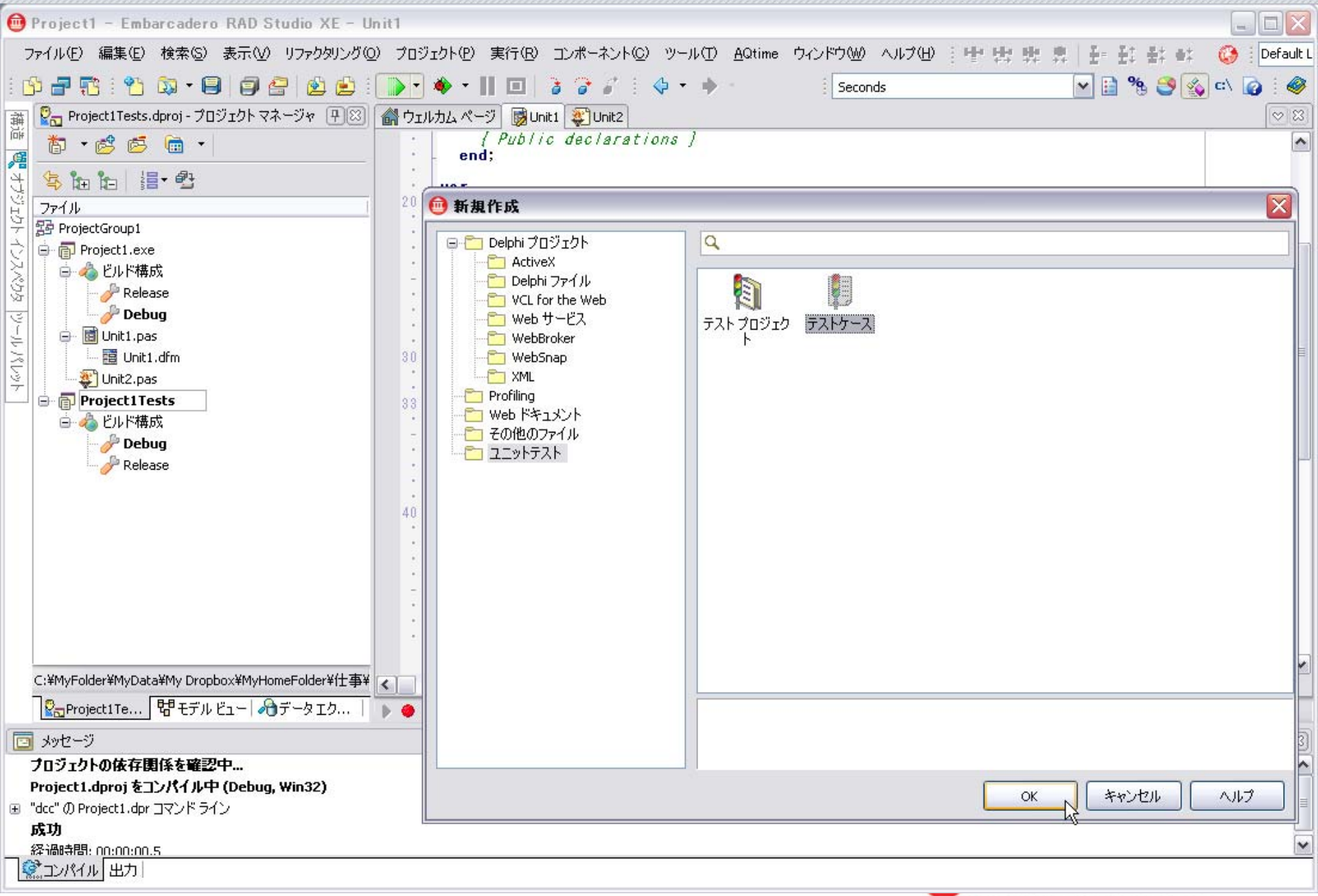

Project1 - Embarcadero RAD Studio XE - Unit1 [ビルド完了]
ファイル(F) 編集(E) 検索(S) 表示(V) リファクタリング(O) プロジェクト(P) 実行(R) コンポーネント(C) ツール(T) AQtime ウィンドウ(W) ヘルプ(H)
Seconds
Project1.dproj - プロジェクト マネージャ
ウェルカム ページ
Unit1
Unit2
ファイル
ProjectGroup1
Project1.exe
ビルド構成
Release
Debug
Unit1.pas
Unit1.dfm
Unit2.pas
新規作成
Delphi プロジェクト
ActiveX
Delphi ファイル
VCL for the Web
Web サービス
WebBroker
WebSnap
XML
継承可能項目
Profiling
Web ドキュメント
その他のファイル
ユニットテスト
テスト プロジェクト
テストケース
OK
キャンセル
ヘルプ
メッセージ
プロジェクトの依存関係を確認中...
Project1.dproj をコンパイル中 (Debug, Win32)
"dcc" の Project1.dpr コマンド ライン
成功
コンパイル
出力

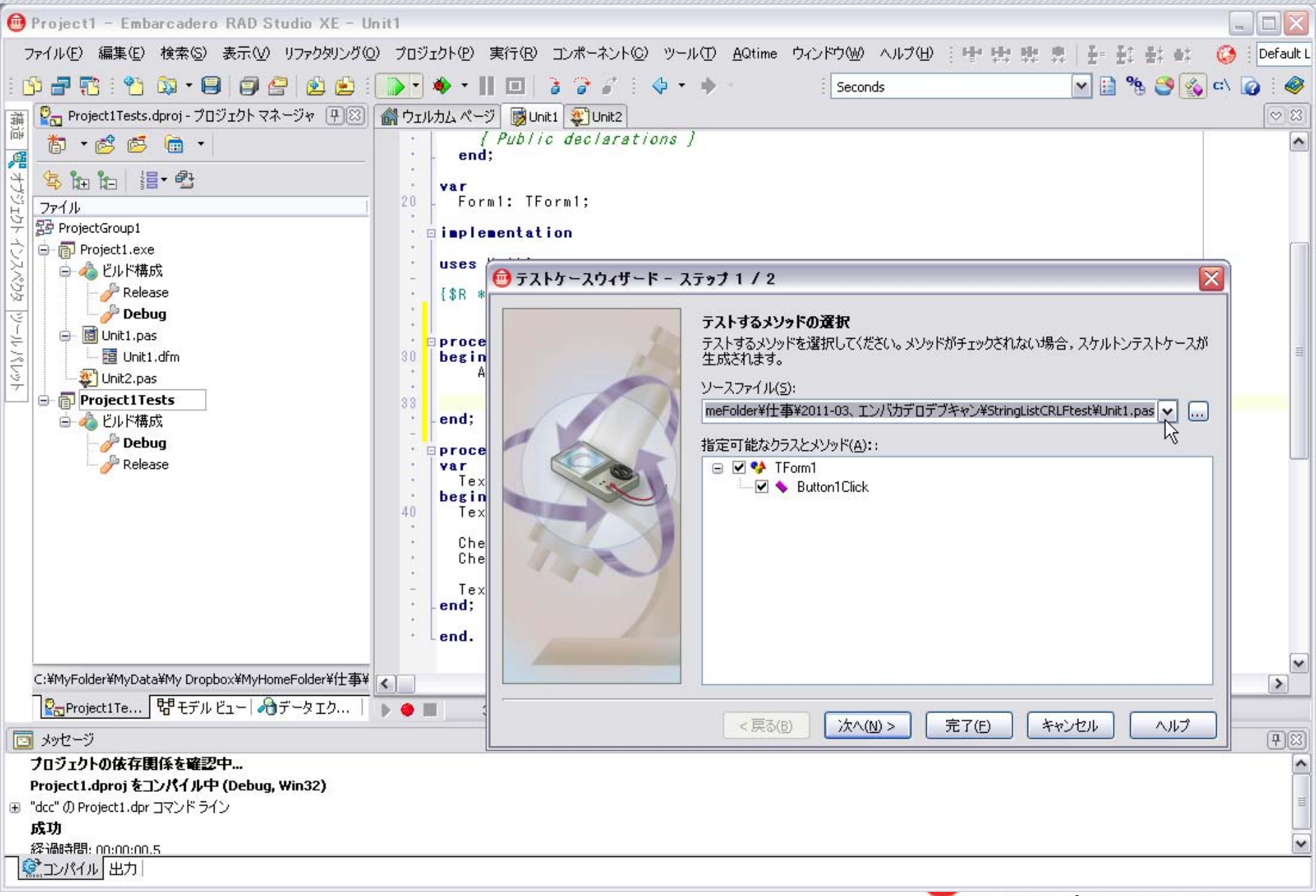


Developer Camp

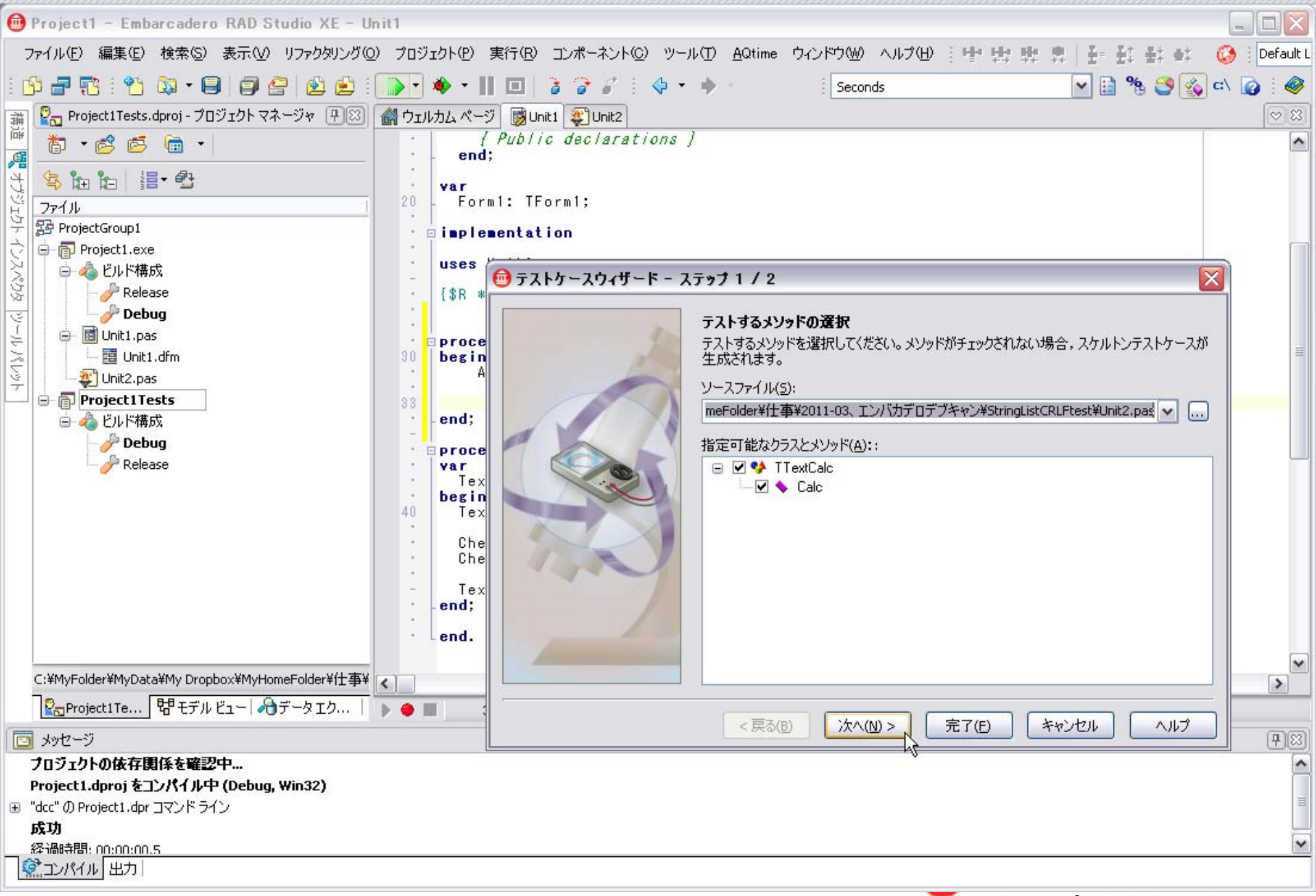


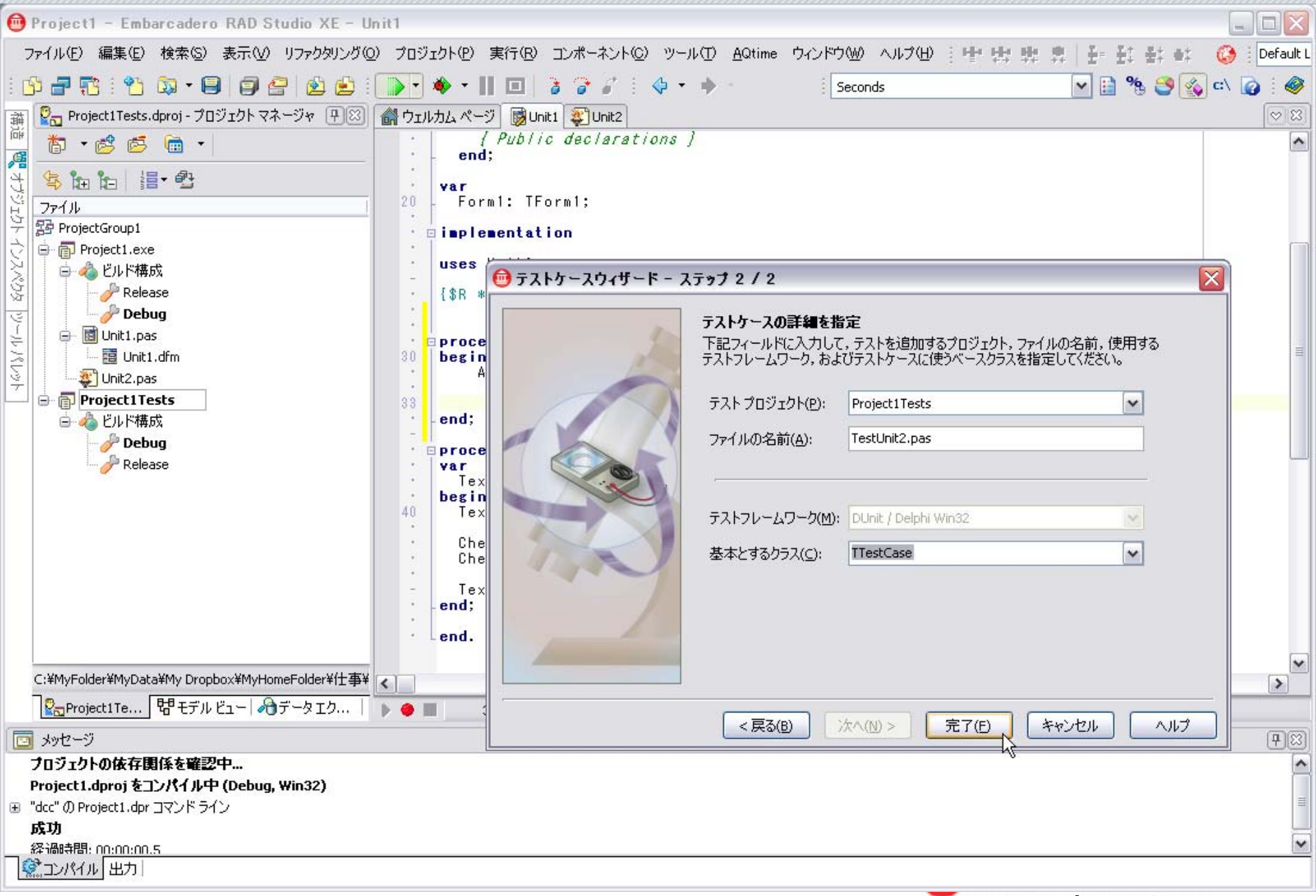
Developer Camp


Project1 - Embarcadero RAD Studio XE - Unit1
ファイル(F) 編集(E) 検索(S) 表示(V) リファクタリング(O) プロジェクト(P) 実行(R) コンポーネント(C) ツール(T) AQtime ウィンドウ(W) ヘルプ(H)
Default L
Seconds
Project1Tests.dproj - プロジェクト マネージャ
ウェルカム ページ
Unit1
Unit2
{ Public declarations }
end;
ファイル
ProjectGroup1
Project1.exe
ビルド構成
Release
Debug
Unit1.pas
Unit1.dfm
Unit2.pas
Project1Tests
ビルド構成
Debug
Release
C:¥MyFolder¥MyData¥My Dropbox¥MyHomeFolder¥仕事¥
Project1Te...
モデル ビュー
データ エク...
新規作成
Delphi プロジェクト
ActiveX
Delphi ファイル
VCL for the Web
Web サービス
WebBroker
WebSnap
XML
Profiling
Web ドキュメント
その他のファイル
ユニットテスト
テスト プロジェクト
テストケース
OK
キャンセル
ヘルプ
メッセージ
プロジェクトの依存関係を確認中...
Project1.dproj をコンパイル中 (Debug, Win32)
"dcc" の Project1.dpr コマンド ライン
成功
経過時間: 00:00:00.5
コンパイル
出力

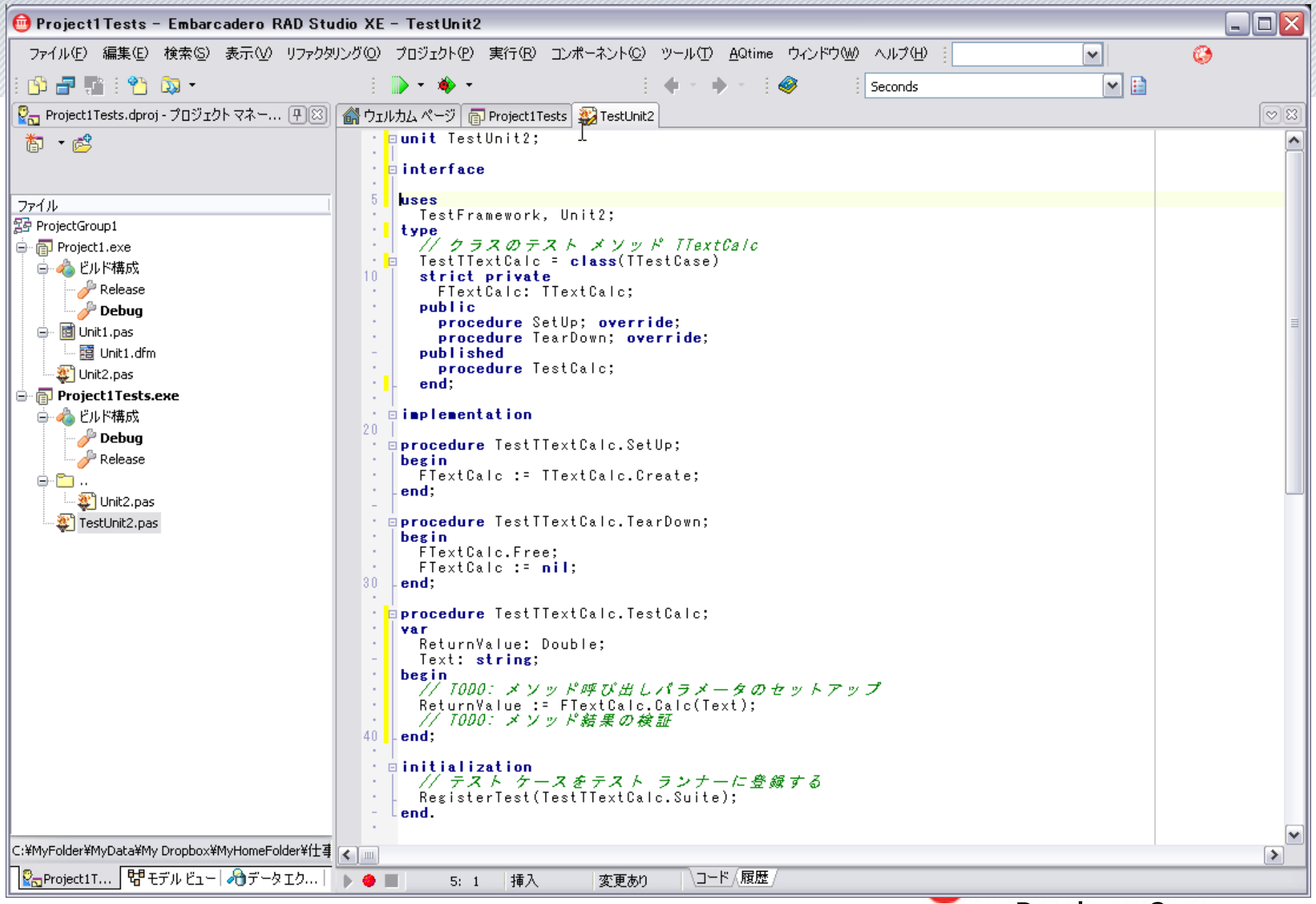


Developer Camp


Project1 - Embarcadero RAD Studio XE - Unit1
ファイル(F) 編集(E) 検索(S) 表示(V) リファクタリング(O) プロジェクト(P) 実行(R) コンポーネント(C) ツール(T) AQtime ウィンドウ(W) ヘルプ(H)
Seconds
Project1Tests.dproj - プロジェクト マネージャ
ウェルカム ページ
Unit1
Unit2
{ Public declarations }
end;
var
Form1: TForm1;
implementation
uses
{$R *
proce
begin
end;
proce
var
Tex
begin
Tex
Che
Che
Tex
end;
end.
ファイル
ProjectGroup1
Project1.exe
ビルド構成
Release
Debug
Unit1.pas
Unit1.dfm
Unit2.pas
Project1Tests
ビルド構成
Debug
Release
C:¥MyFolder¥MyData¥My Dropbox¥MyHomeFolder¥仕事¥
Project1Te...
モデル ビュー
データ エク...
テストケースウィザード - ステップ 1 / 2
テストするメソッドの選択
テストするメソッドを選択してください。メソッドがチェックされない場合，スケルトンテストケースが生成されます。
ソースファイル(S):
meFolder¥仕事¥2011-03、エンバカデロデブキャン¥StringListCRLFtest¥Unit2.pas
指定可能なクラスとメソッド(A):
TTextCalc
Calc
< 戻る(B)
次へ(N) >
完了(F)
キャンセル
ヘルプ
メッセージ
プロジェクトの依存関係を確認中...
Project1.dproj をコンパイル中 (Debug, Win32)
"dcc" の Project1.dpr コマンド ライン
成功
経過時間: 00:00:00.5
コンパイル
出力

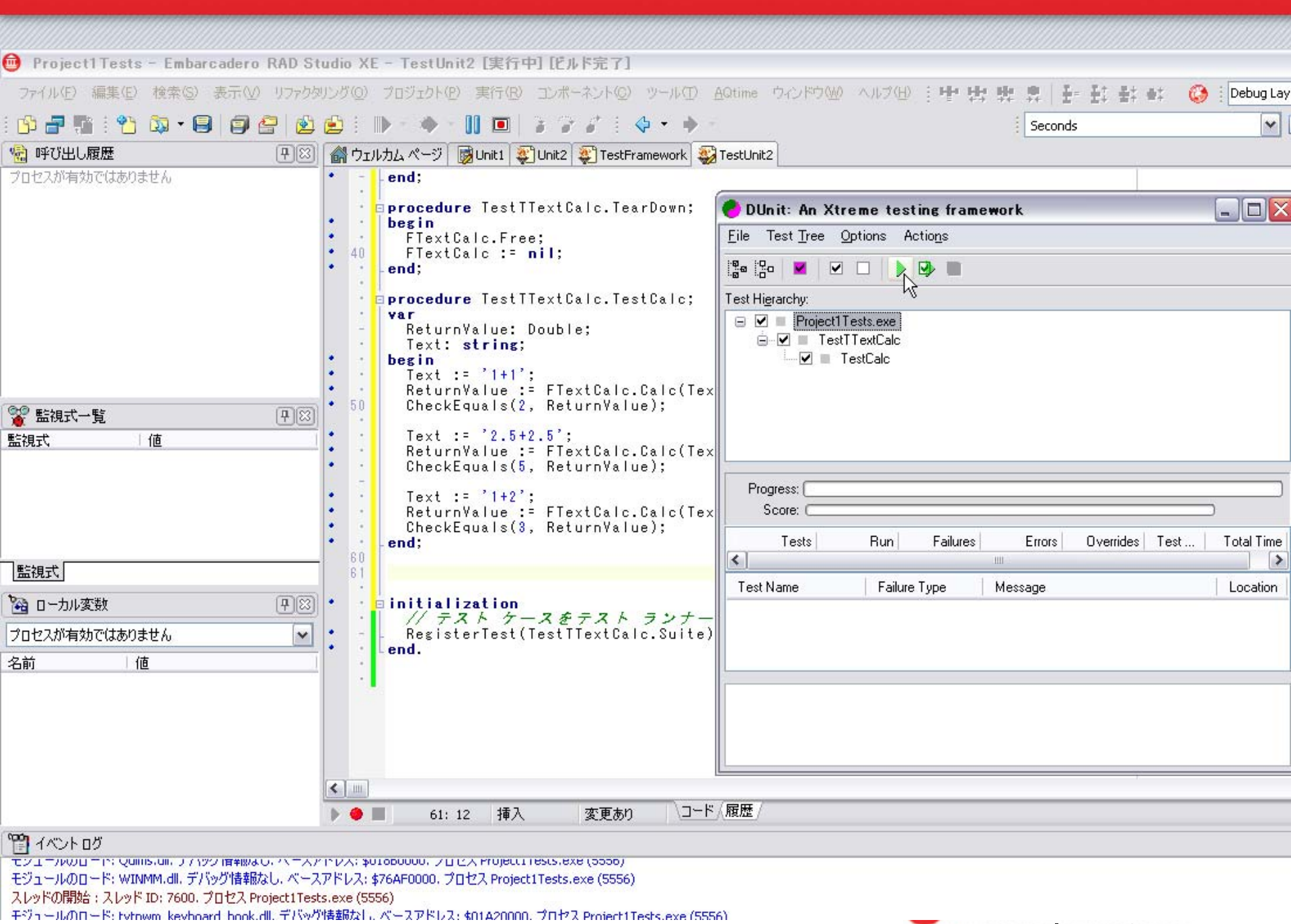


Developer Camp

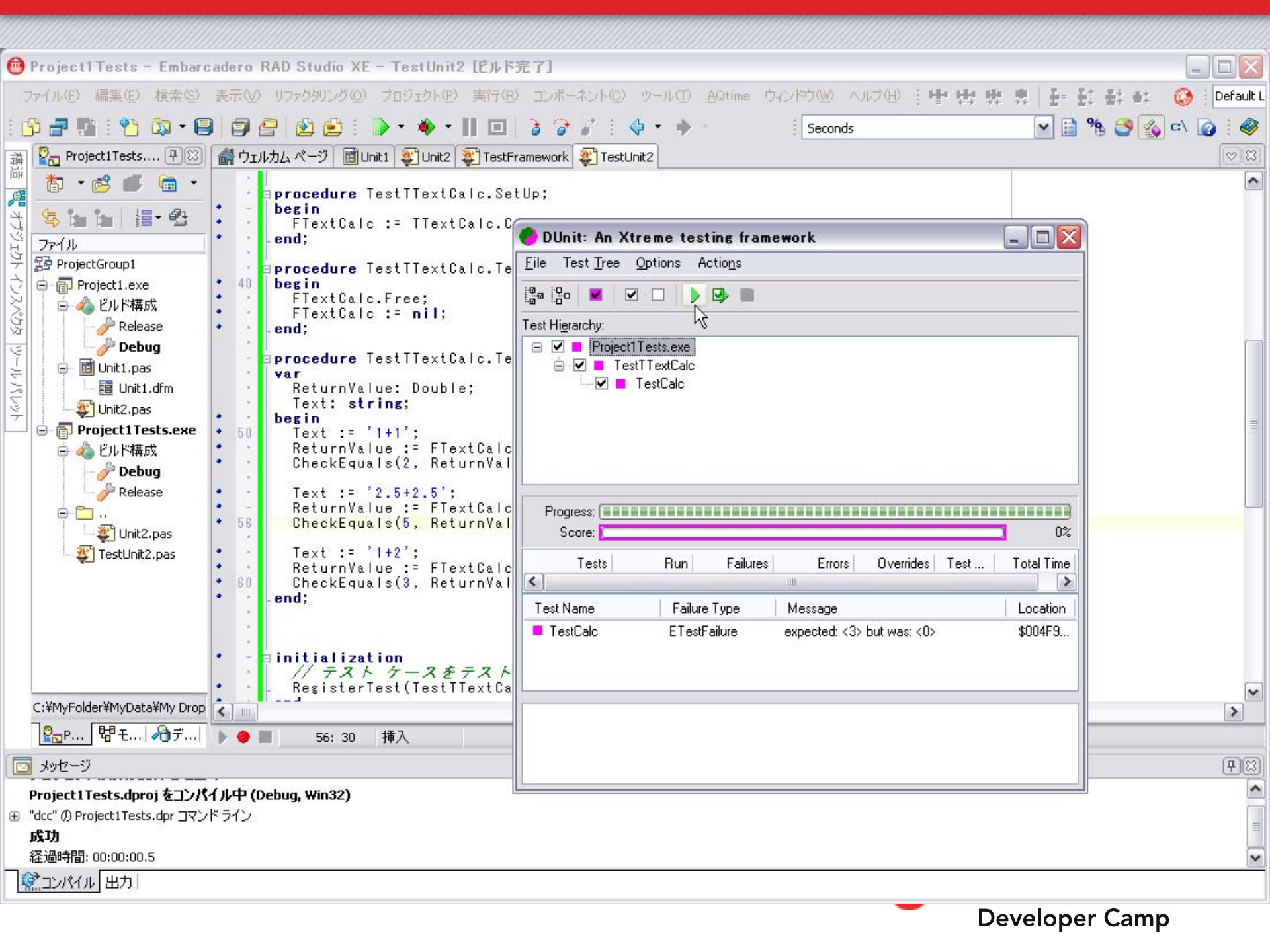

Project1Tests - Embarcadero RAD Studio XE - TestUnit2
ファイル(F) 編集(E) 検索(S) 表示(V) リファクタリング(O) プロジェクト(P) 実行(R) コンポーネント(C) ツール(T) AQtime ウィンドウ(W) ヘルプ(H)
Seconds
Project1Tests.dproj - プロジェクト マネー...
ウェルカム ページ
Project1Tests
TestUnit2
ファイル
ProjectGroup1
Project1.exe
ビルド構成
Release
Debug
Unit1.pas
Unit1.dfm
Unit2.pas
Project1Tests.exe
ビルド構成
Debug
Release
..
Unit2.pas
TestUnit2.pas
C:¥MyFolder¥MyData¥My Dropbox¥MyHomeFolder¥仕事
Project1T...
モデル ビュー
データエク...
unit TestUnit2;
interface
uses
TestFramework, Unit2;
type
// クラスのテスト メソッド TTextCalc
TestTTextCalc = class(TTestCase)
strict private
FTextCalc: TTextCalc;
public
procedure SetUp; override;
procedure TearDown; override;
published
procedure TestCalc;
end;
implementation
procedure TestTTextCalc.SetUp;
begin
FTextCalc := TTextCalc.Create;
end;
procedure TestTTextCalc.TearDown;
begin
FTextCalc.Free;
FTextCalc := nil;
end;
procedure TestTTextCalc.TestCalc;
var
ReturnValue: Double;
Text: string;
begin
// TODO: メソッド呼び出しパラメータのセットアップ
ReturnValue := FTextCalc.Calc(Text);
// TODO: メソッド結果の検証
end;
initialization
// テスト ケースをテスト ランナーに登録する
RegisterTest(TestTTextCalc.Suite);
end.
5: 1
挿入
変更あり
コード
履歴

```
procedure TestTTextCalc.TestCalc;
var
  ReturnValue: Double;
  Text: string;
begin
  // TODO: メソッド呼び出しパラメータのセットアップ
  ReturnValue := FTextCalc.Calc(Text);
  // TODO: メソッド結果の検証
end;
```

```
begin
  Text := '1+1';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(2, ReturnValue);

  Text := '2.5+2.5';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(5, ReturnValue);

  Text := '1+2';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(3, ReturnValue);
end;
```



Developer Camp

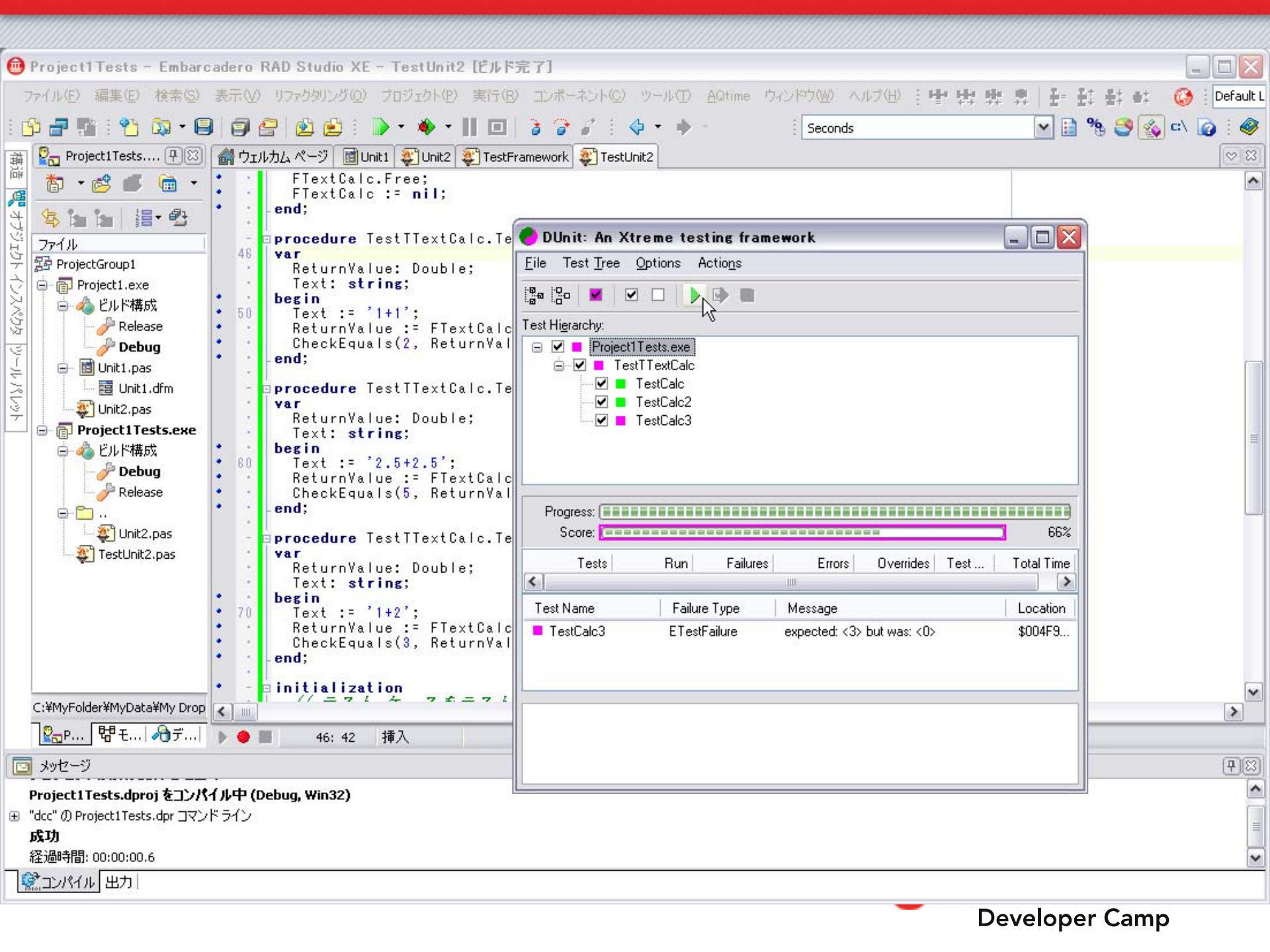


Developer Camp

```
procedure TestTTextCalc.TestCalc;
var  ReturnValue: Double;  Text: string;
begin
  Text := '1+1';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(2, ReturnValue);
end;
```

```
procedure TestTTextCalc.TestCalc2;
var  ReturnValue: Double;  Text: string;
begin
  Text := '2.5+2.5';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(5, ReturnValue);
end;
```

```
procedure TestTTextCalc.TestCalc3;
var  ReturnValue: Double;  Text: string;
begin
  Text := '1+2';
  ReturnValue := FTextCalc.Calc(Text);
  CheckEquals(3, ReturnValue);
end;
```



Developer Camp

embarcadero
Developer Camp

構造化例外を超えて

例外 try ... finally ... end;

- 構造化例外、使っていますか？
  使いこなせていますか？

```
オブジェクト := Tクラス.Create;
try
  処理
finally
  オブジェクト.Free;
end;
```

- オブジェクトCreate時に例外が発生したらどうするの？
  - メモリ確保はやっていないとみなせるから、Freeはしなくていいよな...
- Free時にも例外が発生したらどうするんだ？
  - 所有しているオブジェクトが連続破棄時の途中で例外発生したらどこまで再度破棄すればいいんだ？
- ・・・

try … except on on e:Exception do ... end

```
try
  オブジェクト := Tクラス.Create;
  try
    処理
  finally
    try
      オブジェクト.Free;
    on E:E-Freeの時に想定される例外do
      例外処理
    end;
  end;
except
  on E: E-Createの時の例外 do
    例外処理
  end;
end;
```

```
try
  オブジェクト := Tクラス.Create;
  try
    処理
  finally
    オブジェクト.Free;
  end;
except
  on E: E-Createの時の例外 do
    例外処理
  end;
  on E: E-Freeの時に想定される例外 do
    例外処理
  end;
end;
```

- どちらにしても何か息苦しさを感じる。
  - 例外が起きても問題が起きないようにするには
  - 毎回こんな処理を書かなければならない、のか？

Joel On Software
「間違ったコードは間違って見えるようにする」

- もう読んでるよね…さすがに…
- まさか…
  - 読んでないなら、ここで1時間、私の話を聞いている場合じゃなくてインターネットに接続して読もう！
  - なぜBorlandがMSに敗北して社名を失ったかがわかるぞ！(おっと失言)

- あなたの技術者スキルとキャリアに影響する、一冊だと思います。

- そこでは「**例外は実質的に見えないgotoであり、目に見えるgotoよりいっそう悪い**」と書かれていて、
- 「**もちろん何百万という人々が私ののど元に飛びかかってきた。私の擁護に立ち上がった唯一の人間は、もちろんレイモンド・チェンで、彼は世界最高のプログラマなわけだ**」という話が載っています。
  - レイモンド・チェンは、元Windowsコア開発者

- これ読んで私は気がつき、長年の・積年の謎が解けました。

品質の高いコードを書きたいのなら
構造化例外など窓から捨ててしまいなさい。

- アルファベットが1つのメソッドか関数か何かだと思ってください。

```
try A;B;C;D;E;F;G;H;I;J;K;L;M;N; … finally … end;
```

- Aの内部には、A1;A2;A3;A4;… などがあり、Bの内部には、B1;B2;B3…
- このような状況だと、どこかで例外が起きたときに
  じゃあ、Jの処理は行われたのか否か？と見分ける事は困難。
- 自分で自前の例外クラスを作成して呼び出したとしても、同じ問題が起こりえます。
- つまり「例外は起きないように察知して対応処理を書きましょう。」
  ということ。

どう書くべきか

```
try
  //処理A、100行あるとする
  IntValue := StrToInt(StringValue);
  Result := IntValue / 10;
  //処理B、100行あるとする
except
  on Exception do
    処理
  end;
end;
```

- このようには
  書くべきではありません
- try finally も使うべきではありません。

このように書くべきです

```
//処理A、100行あるとする
if TryToStrToInt(StringValue, IntValue) = False then
begin
  //StringValueが数値変換出来ない場合のエラー処理
end else
if IntValue = 0 then
begin
  //変換された値が0の場合に0除算エラーが起きないための処理
end else
begin
  Result := IntValue / 10;
end;
//処理B、100行あるとする
```

- 例外を発生させないように自分のコードで防御すれば安全。
- 例外は、実行時エラーとみなしコンパイルエラーと同じように開発中に見たら全て修正する。
- 例外をスローする必要もキャッチする必要もない。
- **という対応が望ましいです。**

「構造化例外を超えて」

- 『特定の問題が発生した場合、その特定の問題だと知らせて、それに対する処理を最後に一括で処理する。』

  という、
  構造化例外の持つ思想自体は間違っていません。

- 単に、例外発生したらジャンプする
  そしてキャッチしなければならないという仕様が
  間違っているだけです。

- では、構造化例外の持つ思想だけを実装してみましょう。

「構造化例外を超えて」

```
type  E_MyException1 = class(Exception);
      E_MyException2 = class(Exception);

  //処理A、100行あるとする
  ErrorList.Clear;
  if ErrorList.Count = 0 then
    if TryStrToInt(StringValue, IntValue) = False then
      ErrorList.Add(E_MyException1.Create('文字がIntegerに変換できないよ'));

  if ErrorList.Count = 0 then
    if IntValue = 0 then
      ErrorList.Add(E_MyException2.Create('数値が0除算エラーだよ'));

  if ErrorList.Count <> 0 then
  begin
    ErrorIndex := ErrorList.IndexOfClass(Exception, 0);
    begin
      while ErrorIndex <> -1 do
      begin
        ShowMessage(ErrorList.Items[ErrorIndex].Message);
        ErrorIndex := ErrorList.IndexOfClass(Exception, ErrorIndex + 1);
      end;
      //リスト中の例外を取得してループして出力している。
    end;
  end else
  begin
    Result := IntValue / 10;
    //処理B、100行あるとする
  end;
```

ErrorList

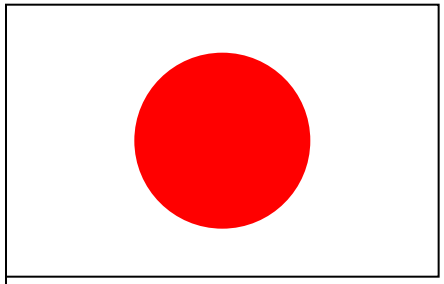
- ErrorListはシングルトンオブジェクト

- 例外を調査する前に.Clearする
- 例外を起こしたくなったら.Add(Error.Creat)する
- 例外が起きたかどうか確認したかったら.Countを調べる
- .IndexOfClass(例外)でその例外が含まれているかどうか調べる事ができる。
- .IndexOfClass(例外,Index)でループして例外クラスを調べれば、いくつでも該当例外クラスが発生しているか確認できる

- 近日、公開予定。 To DelFusaBlog

- ご静聴ありがとうございました。

- See You



復興がんばろう