1 Pattern Techniken

1.1 Refactoring

1.1.1 DUnit

Als letzten Teil des ersten Kapitels zu Pattern Techniken stelle ich eine Art musterhaftes Testen vor, dass im Zusammenhang mit dem Vorgehensmuster des Extreme Programing (XP) besser bekannt wurde und nun auch für Delphi und Kylix einsatzbereit ist.

Dunit ist ein Framework das ursprünglich aus der Java Welt stammt und in sich eine Technik wie ein Werkzeug vereinnt. Das faszierende an DUnit ist die Tatsache, dass es innerhalb des Framworks intensiv von Design Patterns gebrauch macht, vor allem Decorator, Command und Composite, und gleichzeitig einen Test-Rahmen fürs Refactoring vorgibt. Denn die Qualitätssicherung von Code bei Änderungen ist sehr mühselig, so dass sich immer häufiger automatische Testmethoden anbieten.

Die folgende Tabelle soll aufzeigen, wo sich innerhalb der Refactoring-Funktionen Fehler einschleichen können, die bei einer passenden, eingerichteten Testklasse automatisch ans Tageslicht kommen:

Einheit	**Refactoring Funktion**	**Beschreibung**
Package	Rename Package	Umbenennen eines Packages
Package	Move Package	Verschieben eines Packages
Class	Extract Superclass	Aus Methoden, Eigenschaften eine Oberklasse erzeugen und verwenden
Class	Introduce Parameter	Ersetzen eines Ausdrucks durch einen Methodenparameter
Class	Extract Method	Heraustrennen einer Codepassage
Interface	Extract Interface	Aus Methoden ein Interface erzeugen
Interface	Use Interface	Erzeuge Referenzen auf Klasse mit Referenz auf implementierte Schnittstelle
Component	Replace Inheritance with Delegation	Ersetze vererbte Methoden durch Delegation in innere Klasse
Class	Encapsulate Fields	Getter- und Setter einbauen
Modell	Safe Delete	Löschen einer Klasse mit Referenzen

Tab. 1. 1: Testen des Refactoring

Das Testen mit DUnit oder einem anderen Framework setzt auf drei Elementen auf.

- Das eigentliche Testobjekt, fixture genannt
- Der Testfall zum Objekt, action genannt
- Das Resultat nach dem Testfall, check genannt

Als einfaches Beispiel sei hier das Addieren von zwei Zahlen erwähnt. Das Resultat kann Positiv, Negativ oder ein unerwarteter Fehler sein. Der Check testet, ob das Resultat dem Originalwert plus 5 entspricht:

```
function TestAdd5ToNumber(n: integer):boolean;
var
  OrigVal: integer;
begin
  OrigVal := n;
  n := n + 5;
  //Check result of the action
  Result := n = OrigVal + 5;
end;
```

Einrichten des Framework

Als erstes gelangt man zum Framework durch folgenden Link:

http://dunit.sourceforge.net

Mindestens Delphi 4 wird benötigt. Achten Sie darauf, beim Auspacken der Dateien die Verzeichnisnamen beizubehalten. Als nächstes fügen Sie den Pfad der Sourcen (vor allem TestFramework.pas und GUITestRunner.pas) in den Delphi Optionen hinzu. Einfach den kompletten Pfad \dunit\src hinzufügen. Einen ersten Eindruck gewinnt man, indem im Verzeichnis \dunit\tests zuerst die Library dunittestlib.dpr und dann unittests.dpr compiliert wird, demzufolge beim Starten der Exe das ganze Spektrum der eingebauten Testmöglichkeiten ersichtlicht wird:

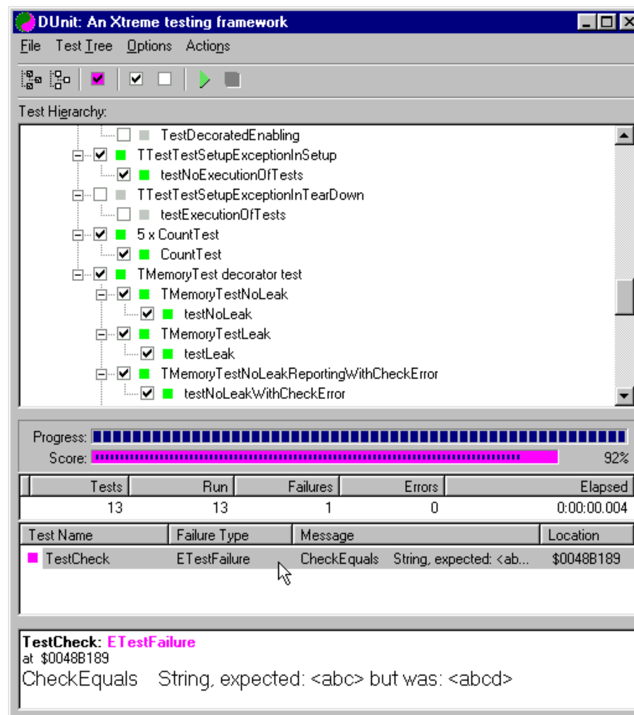

Abb. 1. 2: Das GUI innerhalb von DUnit

Wenn ich nun in der Datei UnitTestFramework die Zeile eines Vergleichs derart ändere:

```
CheckEquals('abc', 'abcd', 'CheckEquals  String');
```

So provoziere ich die logische Fehlermeldung in Abb. 60 im Framework, das nebst der Adresse auch die vergangene Zeit und die Anzahl der Testläufe protokolliert. Wenn man einen eigenen Testfall einrichtet, erbt man die Klasse TTestCase als Subclassing, die wiederum zur Laufzeit unseren registrierten Test mit RTTI auswertet. Demzufolge sind unsere Testmethoden als published zu deklarieren:

```
TFailuresTestCase = class(TTestCase)
  private
    procedure DoNothing;
  published
    procedure OneSuccess;
    procedure OneFailure;
    procedure SecondFailure;
    procedure OneError;
end;
```

Im initializaiton Teil der Unit lässt sich der Test beim Framework registrieren. Der initialization-Abschnitt enthält Anweisungen, die beim Programmstart in der angegebenen Reihenfolge ausgeführt werden. Arbeiten Sie beispielsweise mit definierten Datenstrukturen, können Sie diese im initialization-Abschnitt initialisieren.

```
Initialization
  RegisterTests('GUI Tests', [TGUITestRunnerTests.Suite]);
```

Damit das Framework weiss, welche registrierten Tests von Anfang an in der Startroutine geprüft werden, ist es zentral, die Projektdatei mit den Testklassen auf den Start des Frameworks umzulenken. Anstelle der eigentlichen Applicaition.Run tritt die des Frameworks. TestFramework.pas und GUITestRunner.pas sind in der uses Anweisung der Projektdatei anzufügen. Mit den entsprechenden Compilerdirektiven lassen sich diese zusätzlichen Testcodes bei der Auslieferung ja wieder bereinigen:

```
begin
{$IFDEF DUNIT CLX}
    TGUITestRunner.RunRegisteredTests;
{$ELSE}
    GUITestRunner.RunRegisteredTests;
{$ENDIF}
end.
```

Weil bedingte Direktiven jedoch den Quellcode schwer verständlich und komplizierter in der Wartung machen, sollten Sie wissen, wann es sinnvoll ist, diese zu verwenden. Dunit arbeitet intern auch intensiv mit den asserts oder assigned Methoden. In Delphi können Sie mit Assert testen, ob Bedinungen verletzt werden, die als zutreffend angenommen werden. Assert bietet eine Möglichkeit, eine unerwartete Bedingung zu simulieren und ein Programm anzuhalten, anstatt die Ausführung in unbekannter Konstellation fortzusetzen. Assert übernimmt als Parameter einen Booleschen Ausdruck und einen optionalen Meldungstext. Schlägt der Boolesche Test fehl, löst Assert eine EAssertionFailed-Exception aus. Auch die Kombination von beidem ist in Dunit anzutreffen:

```
procedure TTestResult.Run(test: ITest);
begin
  assert(assigned(test));
  if not ShouldStop then begin
    StartTest(test);
    try
      if RunTestSetUp(test) then
        RunTestRun(test);
      RunTestTearDown(test);
    finally
      EndTest(test);
    end;
```

```
  end;
end;
```

In der Regel werden Assertions nicht in Programmversionen verwendet, die zur Auslieferung vorgesehen sind. Deshalb wurden Compiler-Direktiven implementiert, mit denen die Generierung des zugehörigen Codes deaktiviert werden kann:

```
$ASSERTIONS ON/OFF  (Lange Form)
```

Workshop

Die Einarbeitungszeit eines solchen Frameworks beträgt einige Tage, bei wirlich zwingendem Gebrauch, sei es von der QS gefordert oder vom Kunden erwünscht, ist eine gute Woch für die Schulung zu planen, da vor allem das Wissen um die Testfälle zu erstellen, zu Buche schlägt. Neben dem eigentlichen Programmentwurf besteht die Entwicklung aus einem sich ständig wiederholenden Zyklus von Quelltexteingabe und anschließender Fehlerprüfung bei DUnit. Damit auch wirklich alle Aspekte des Programmverhaltens von diesem Test erfasst werden, müssen Sie planvoll vorgehen. Sobald Sie einen Programmierfehler gefunden haben, korrigieren Sie das Problem im Quelltext, compilieren erneut und fahren mit dem Test fort.

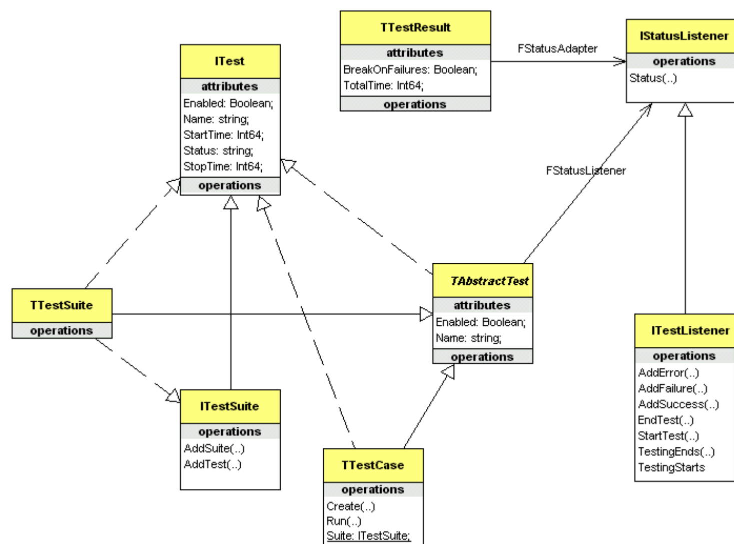

Abb. 1. 3: Das Framework von DUnit

Eine besondere Situation ist auch das Testen bei Vererbung. Eigentlich muss man die Methoden der Oberklasse nicht nochmals in den Unterklassen testen. Zu testen sind jedoch die überschriebenen Methoden! Auch bei Eigenschaften, die in der Unterklasse

mit neuen Werten gefüttert werden und dann von der Oberklasse gebraucht werden, ist Vorsicht zu walten. Ich beginne nun Schritt für Schritt mit einem Testfall einer StringList mit DUnit. Listen sind häufige Kandidaten, da ein verändertes Datenformat oder eine Typenkonversierung zu Seiteneffekten in den Listen führen kann.

Ich starte ein neues Projekt und füge in der zu testenden Unit `TestFramework` in der uses Klausel hinzu. Ich deklariere eine neue Klasse, die von TTestCase abstammt.

```
Interface
uses
 TestFrameWork, Classes;
type
 TTestStringList = class(TTestCase)
 private
   Fsl: TStringList;
 protected
   procedure SetUp; override;
   procedure TearDown; override;
 published
   procedure TestPopulateStringList;
   procedure TestSortStringList;
 end;
```

Diese Testklasse hat sich nun im Initialisierungsteil der Unit zu registrieren:

```
RegisterTest('CODESIGN suite', TTestStringList.Suite);
```

Als letztes der Vorbereitung des „Frameworkshop" ist das Einklinken in die Projektdatei zu bewerkstelltigen, so dass mit dem Start der Anwendung getestet werden kann:

```
Application.Run;
to
GUITestRunner.RunRegisteredTests;
```

Beim Starten des Programms sind die Testfälle im Baum des Framework ersichtlich, das die Aufgabe eines Regiezentrums übernimmt, bei uns auch mal gerinschätzig mit TestDirector tituliert wurde. Ich komme zur Implementation der beiden Testmethoden:

```
procedure TTestStringList.SetUp;
begin
 Fsl:= TStringList.Create;
end;

procedure TTestCaseFirst.TearDown;
begin
 Fsl.Free;
end;
```

Die Bezeichnung Setup und Teardown ist eine Konvention in der Testwelt, die besagt in diesen Methoden die Vor- und Nachbedingungen einer Testklasse zu implementieren, d.h. Initialisierungen, Bedingungen bis hin zu abschliessenden Aufräumarbeiten. Bei rein arithmetischen Tests fehlen in der Regel diese beiden Methoden.

Die eigentlichen Testfälle Population und Sortierung soll aufzeigen, wie repetitive Fälle in das Framework eingebunden werden und einen Eindruck vermitteln, dass sich Testen auch automatisieren lässt. Klar, diese Fälle sind noch trivial und stellen in keiner Art und Weise die Funktionalität einer TStringList in Frage, jedoch wie häufig scheitert man am Trivialen wenn es sich häuft. Check & Test soll hier die Devise sein.

```
procedure TTestCaseFirst.TestPopulateStringList;
var
 i: Integer;
begin
 Check(Fsl.Count = 0);
 for i:= 1 to 50 do // Iterate
   Fsl.Add('i');
 Check(Fsl.Count = 50);
end;

procedure TTestCaseFirst.TestSortStringList;
begin
  Check(Fsl.Sorted = False);
  Check(Fsl.Count = 0);
  Fsl.Add('Trade');
  Fsl.Add('Roboter');
  Fsl.Add('Love');
  Fsl.Sorted:= True;
  Check(Fsl[2] = 'Trade');
  Check(Fsl[1] = 'Roboter');
  Check(Fsl[0] = 'Love');
end;
```

Weitere Techniken wie das Verwalten von n-Testklassen in einer Testsuite, Starten aus der Konsole oder repetitive Tests ist in den Source Beispielen von DUnit ausreichend beschrieben. Ein letzte Frage die sich stellt, ist das Verwalten der Testklassen als eigene Unit oder direkt im produktiven Code. Dazu sind in den Sourcen wiederum zwei Projekte, die den Unterschied aufzeigen, vorhanden. Die Trennung in verschiedene Units hat zwei klare Vorteile: Das Release Management wie das Arbeiten in Teams lässt sich vereinfachen, bedingt aber durch Delegation einen höheren Anfangsaufwand, z.B. benötigt man auch zwei Projektdateien, die sich natütlich in einer Projektgruppe definieren lassen.

Sie können zusammengehörige Projekte zu Projektgruppen zusammenfassen. Projektgruppen ermöglichen es Ihnen, an miteinander verbundenen Projekten zu arbeiten und diese zu organisieren (z.B. Anwendungen und DLLs, die zusammen eine

mehrschichtige Anwendung bilden). Die Projektgruppendatei (Dateinamenserweiterung BPG) enthält Make-Befehle, um die Projekte der Gruppe zu erzeugen. Immer, wenn Sie einer Projektgruppe ein Projekt hinzufügen, wird ein Verweis auf dieses Projekt in die BPG-Datei eingefügt.

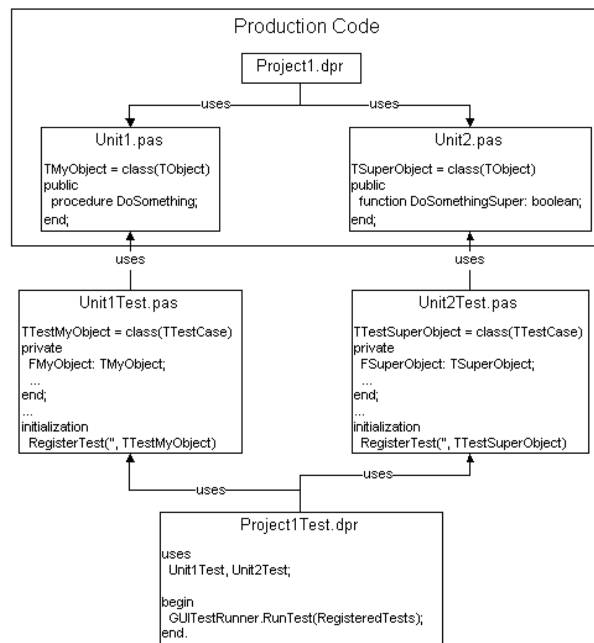

Abb. 1.4: Produktiver- und Testcode in DUnit

DUnit is an Xtreme testing framework for Borland Delphi programs. It was originally inspired on the JUnit framework written in Java by Kent Beck and Erich Gamma, but has evolved into a tool that uses much more of the potential of Delphi to be much more useful to Delphi developers.

DUnit Test

Wozu dient das folgende erneute Auslösen der Exception mit DUnit?

```
function TfrmTrans.GetFileList(const Path: string): TStringList;
  var i: Integer; SearchRec: TSearchRec;
begin
  result:= TStringList.Create;
  try
    i:= FindFirst(Path, 0, SearchRec);
```

```
    while i = 0 do begin
      result.Add(SearchRec.Name);
      i := FindNext(SearchRec);
    end;
  except
    result.Free;
    raise; //erneutes Auslösen
  end;
end;
```

In manchen Fällen muss eine Prozedur oder Funktion beim Auftreten einer Exception bestimmte Aufräum-Operationen durchführen, ist aber in Wirklichkeit nicht für die Behandlung der Exception vorbereitet. Ein Beispiel zeigt die Prozedur `GetFileList`, die ein TStringList-Objekt erzeugt und ihm die Dateinamen zuweist, die mit einem bestimmten Suchpfad übereinstimmen.

Die Funktion legt zuerst ein neues `TStringList`-Objekt an und initialisiert dann die Stringliste mit Hilfe der Funktionen `FindFirst` und `FindNext` aus der Unit `SysUtils`. Wenn die Initialisierung aus irgendeinem Grund fehlschlägt, etwa weil der angegebene Suchpfad ungültig ist oder weil zu wenig Speicherplatz verfügbar ist, muss `GetFileList` die neu angelegte Stringliste wieder entfernen, denn der Aufrufende weiss ja noch nichts von ihrer Existenz. Deshalb erfolgt die Initialisierung der Stringliste in einer try...except-Anweisung. Tritt nun eine Exception auf, so kann die Stringliste im except-Abschnitt der Exception-Behandlungsroutine entfernt und anschliessend die Exception erneut ausgelöst werden, um so die weitere Behandlung z.B. an den Aufrufer zu delegieren.