max@kleiner.com

1 Code Reviews mit dem Pascal Analyzer

Endlich ein Tool, daß bereits nach 10 Minuten zum Einsatz gelangt und Ihnen damit nächtelang mehr Qualität beschert. Pascal Analyzer (PAL) ist ein Werkzeug, welches nach gewissen Qualitätsregeln und Metriken ihren Source Code prüft, analysiert und dokumentiert, bspw. fehlende Ausnahmebehandlungen bemängelt oder Namenskonventionen überprüft. Die meisten der generierten Reports eignen sich auch für ein Systemhandbuch oder sonstige Dokumentation. Viel Expertenwissen bei einfacher Bedienung.

1.1 Die Quelle ist das Ziel

Wenn Sie immer schon mal wissen wollten, wo sich toter Code befindet, ob man Namenskonvention einhält oder die Rückgabe einer Funktion wirklich auswertet, dann auf zu PAL.

Pascal Analyzer (PAL) von Peganza ist ein Bewertungswerkzeug zum Messen, Überprüfen, Entwickeln und Dokumentieren von Pascal- und Delphi Code. Mit Hilfe des Tools lassen sich Softwareprojekte jeder Größe einfach analysieren und testen und das Tool ermöglicht damit einen zuverlässigen Code zu erstellen. Die Reports, die man durch PAL meistens in einem Durchgang erzeugt, enthalten Informationen zur Unterstützung und Bewertung des Codes und seiner Struktur beim Entwickeln des Systems. Die Reports lassen sich in einem Text- , XML- oder HTML-Format produzieren und sind einzeln oder kombiniert einsetzbar. PAL nimmt keine Änderungen am Code vor sowie man selbst keine Modifikationen am Code vornehmen muß. Es ist also kein Profiler oder Debugger.

Am häufigsten setzen wir PAL bei den Code Reviews oder als Vorbereitung zur Integration ein, da es das Fine-Tuning und Management des Entwicklungsprozesses erleichtert. So läßt sich beim Ändern einer Subroutine schnell mal die davon abhängigen Prozeduren aufzeigen (z.B. Bindings Report).

Viele Analytiker haben ja ein eigenes, automatisiertes Verfahren entwickelt, um Code auf Fehler hin zu durchsuchen. Damit wollen sie objektive und vergleichbare Erkenntnisse zu jeder untersuchten Software produzieren, und zwar unabhängig von Betriebssystem, Hersteller oder Programmierstil. Bei diesen Leuten hält man daher die Codeinspektion oder ein Review für die effektivste Maßnahme, um die Codequalität verbessern zu können. Ein Teil diese Wissens steckt im PAL.

Im Leistungsumfang der Version 2.1.9, die bei Lizenzierung mit rund 100 Euro zu Buche schlägt, ist auch eine Evaluationsedition frei durch Download beziehbar, die sich nachträglich mit einem Schlüssel freischalten läßt. Kurz vor Redaktionsschluß ist PAL 3.0 erschienen, der Delphi.Net, objektorientierte Metriken und später auch die neuen Sprachelemente von Delphi 2005 unterstützen wird. Obwohl PAL eine Win32 Applikation ist, läßt sich auch CLX Code der Analyse unterziehen. Innerhalb der Konfiguration ist nur ein Umschalten auf das Formformat xfm nötig, oder allenfalls ein Anpassen der Compilerdirektiven. Grundsätzlich wird jede Delphi Version, ja sogar das gute Borland Pascal vom Analyzer geparst. „Ich habe bewußt nicht das gute alte Borland Pascal gesagt, da ich im Sommer 04 in Kasachstan ganze Laufmeter von (Turbo) Pascal Büchern in aktuellen Auflagen gesichtet habe.“

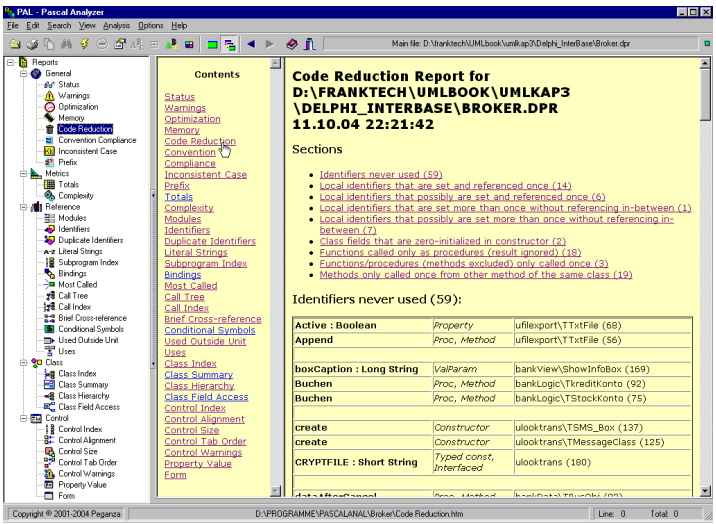

Abb. 1: PAL in der HTML Ansicht

Im Paket ist auch ein „command-line analyzer“ enthalten *(PALCMD.EXE*), der sich parametrisieren läßt und exakt die gleichen Reports wie die GUI Version (*PAL.EXE*) in Abb. 1 erzeugt. Aussagekräftige Barcharts zeigen zudem Aspekte der Code Komplexität und gewisse Profiler Aspekte, wie weiter unten noch zu sehen ist.

Kommen wir zu den wichtigsten Reports:

Der **Optimization Report** deckt Parameter auf, die zu besserer Performance führen können, wie das Setzen des Schlüsselwortes `const`, daß bei nicht modifizierten Referenz Parametern kompakteren Code generiert, eben optimaler ist. Einige Hinweise dienen auch zur Vereinfachung des Codes. Der von Delphi generierte Code benutzt in der Regel die Fast-Call Syntax (Aufrufkonvention ist register), will heißen, daß bei 2 Parametern der Compiler meistens kein Stack-Frame für die Parameter anlegen muß.

Beim **Memory Report** will man mögliche Speicherlöcher vermeiden, so daß alle Allozierungen ohne `try/ finally` angezeigt werden. Sobald PAL ein `Free` oder ein `FreeAndNil` ausgemacht hat, gilt dieser Abschnitt als ungeschützt:

```
procedure ListSocketHeader;
var
  SL: TStringList;
begin
  SL:= TStringList.Create;
  ..
  SL.Free; //missing try-finally block!
end;
```

Der **Conditional Symbols Report** listet die Orte auf wo `$DEFINE/$UNDEF` oder `$IFDEF/$IFNDEF` Direktiven unnötig wiederholt werden. Entfernen dieses Überflusses erhöht die Wartbarkeit des Codes:

```
(*$DEFINE Final*)
var
  X: integer;
  Y: integer;
(*$DEFINE Final*) <- unnecessary, Final is already defined
const
  sAppName = 'AppHexMax';
(*$ENDIF*)
..
(*$ENDIF*)
```

Interessant und enorm qualitätssteigernd ist der **Warnings Report** mit der Sektion “Ambiguous references in with-blocks”, der gleichnamige Bezeichner findet, die eine globale Variable mit einer Variablen innerhalb eines with-statements vergleicht und somit zu Verwechslungen führen kann, siehe Bsp.. Mit der strikten Anwendung von qualifizierten Bezeichnern könnte man diese Fallen umgehen, das würde aber zu Bandwürmern führen. Der Report soll generell die Gefahr von Seiteneffekten reduzieren:

```
var Title: string;
type
  TMyRec = record
    Title: string;
  end;

var Rec: TMyRec;
begin
  with Rec do begin
    ..
    Title:= ´be a big bug’;
  end;
```

```
end;
```

Weiter bei den Warnungen sind Variablen aufgelistet, die wohl referenziert aber nie gesetzt / initialisiert werden oder umgekehrt. Nicht immer hat aber PAL recht, z.B. muß man ein Objekt ohne Felder innerhalb der Klasse auch nicht allozieren.

Der **Code Reduction Report** hilft Codezeilen zu reduzieren indem man Bezeichner die nie referenziert sind (siehe nochmals Abb. 1) eigentlich entfernen oder aus kommentieren kann. Auch Funktionen / Prozeduren die nur einmal aufgerufen werden sind nicht unbedingt effizient und lassen sich durch PAL aufdecken. Bei Subprogrammen, die den Rückgabewert einer Funktion nicht auswerten, legt das Tool ebenfalls sein Veto ein. Hingegen bei virtuellen Methoden kann PAL schon mal eine Fehlanzeige liefern. Als Subprogramm bezeichnet PAL übrigens eine Funktion, Prozedur oder eine Methode.

Wirkungsvoll ist auch die Möglichkeit, alle Codepassagen aufzuzeigen, die eine Variable zweimal setzen obwohl zwischen den Zeilen keine weitere Auswertung derselben erfolgt (Local identifiers that are set more than once without referencing in-between)! :

```
procedure MaxBox;
var
i: integer;
begin
i:= 7;
..
i:= 10; // !! i is set again
..
if i = 10 then begin
..
```

Meistens hat man hier fortlaufend den Code entwickelt ohne Altlasten zu bereinigen. So nebenbei zeigt PAL auch die rote Karte bei initialisierten Strings (Longstrings), da hier der Compiler die Initialisierung schon implizit vornimmt:

```
var
s: string;
begin
s:= ''; // !! init unnecessary
```

Richtig komplex wird es nun mit dem **Complexitiy Report**, welcher in die Rubrik der Software-Metriken paßt. Hier kommt die Decision Point Analyse zum Zug die identisch mit der McCabe's Metrik ist, auch cyclometric complexity genannt. Die Berechnung ist transparent dokumentiert und jederzeit auch manuell nachvollziehbar. Nun, was ist der Nutzen? Diese in der Industrie anerkannte Metrik macht eine Aussage über die „Komplexität" und Verständlichkeit des Code, welcher dann teilweise im Widerspruch zur Wiederverwendbarkeit steht. Bei Werten >10 ist es angebracht, das Subprogramm in kleinere Teile zu zerlegen.

Ein weiterer **Literal String Report** zeigt alle Strings innerhalb von Subprogrammen oder die extern als Konstanten oder String-Ressourcen auftauchen, die vor allem bei der Internationalisierung zum Einsatz gelangen. String-Ressourcen werden ja nicht in die Formulardatei gebunden; sie lassen sich isolieren, indem man Mithilfe des reservierten Worts `resourcestring` Konstanten deklariert. Das Auslagern von Ressourcen oder String Literalen vereinfacht den Übersetzungsprozess oder die Wartbarkeit, denn wer favorisiert schon „festverdrahteten" Code.

Nützlich kann auch der **Uses Report** sein, der nicht benutzte Units aufdeckt. Obwohl der Smart Linker solche Units nicht einbindet, werden die „initialization sections" (oder „finalization sections") trotzdem in die EXE gebunden. Dieses nachträgliche Bereinigen anhand des Reports beschleunigt die Compilationszeit und reduziert Größe und Startzeit der Anwendung.

1.2 Visualisierung

Wie Steve McConnell 1993 schon in *Code complete* schrieb, sind Kontrollen und Tests eine Weise der Entschädigung der vorweggenommenen menschlichen Fehlbarkeit. Oder noch krasser: Nur eine tote Variable ist eine gute Variable, denn Sie sollten die Lebensdauer einer Variablen so kurz wie möglich halten! PAL hilft ihnen dabei. Es gibt 34 Reports (in der Version 3 sogar 40), die getrennt in die fünf Kategorien: General, Metrics, Reference, Class und Control, aufgeteilt sind.

Hervorragend ist auch die visuelle Unterstützung von einigen Reports, wie der **Call Tree Report** demonstriert. Analog einer Stackanalyse untersucht das Tool die Aufrufkaskade, die ähnlich einem Sequenzdiagramm, die Interaktion der Subprogramme auflisten. Beispielhaft ist die Tiefe der Verschachtelung anhand der Einrückung ablesbar:

```
[4]
bankLogic.TkreditKonto.Buchen (230i)
bankLogic.TKonto.Buchen (216i)
   bankLogic.TTransaction.SetTransactionHist (333i)
        bankData.TBusObj.setSP_TransData (264i)
        bankData.TBusObj.setSQLTransData (245i)
bankLogic.TkreditKonto.Bonitaet (312i)

[5]
bankLogic.TStockKonto.Buchen (222i)
bankLogic.TKonto.Buchen (216i)
..[4]
-----------------------------
procedure TKonto.Buchen(betrag: double);
begin
  FKontostand:= FKontostand + betrag;
  FTransakt_Obj.setTransactionHist(betrag);
end;
```

Die Aufrufkaskade ist zudem verlinkt, im Beispiel die [4], da Funktionen aus [5] wiederum mehrfach andere Subprogramme in [4] aufrufen können.

Im weiteren lassen sich in diversen Charts graphisch Abhängigkeiten, Nutzungsziffern oder Komplexitäten (siehe Abb. 2) darstellen. Die Änderung einer Funktion hat Einfluß auf die davon abhängigen Funktionen, die im Bindings Report aufgelistet werden, demzufolge ein erneuter Test der Funktion der Qualität nur zugute käme. Praktisch ist auch der **Controls Report** mit konkreten Hinweisen zu doppelten Tastaturkürzeln, überlappenden Formbereichen oder ein falscher Tab Order, wenn die Konvention von links nach rechts sein soll.

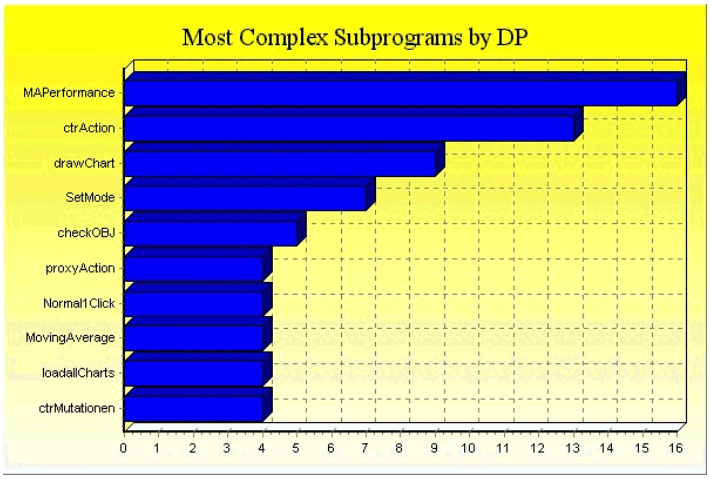

Abb. 2: Erhöhte Aussagekraft einer Softwaremetrik

1.3 Customizing

In den Properties von PAL (Abb. 3) sind viele Möglichkeiten der eigenen Arbeitsmethodik einstellbar. Einige Reports lassen sich bezüglich Geschwindigkeitsmessung (most called subprograms) oder Typensicherheit anpassen. Zwischen Performance und Sicherheit besteht aber ein ewiger Zielkonflikt, der folgendes Beispiel verdeutlicht: Wenn die Anwendung eine Routine über Gebühr in Anspruch nimmt, ist bezüglich Performance eine Optimierung angesagt. Ist bspw. in dieser Routine eine

Typumwandlung nötig, läßt sich ja mit `is` prüfen, ob die Typen verträglich sind. Ein Test mit `is` ist aber ähnlich teuer wie die Typumwandlung selbst.

Im Sinne der Performanzsteigerung hat man nun zwei Möglichkeiten. Entweder man vermeidet die Umwandlung oder man verzichtet auf den Test ob die Typen verträglich sind. Somit spart man die Zeit innerhalb der kritischen Routine und wenn die Typen nicht verträglich sind, meldet sich das Laufzeitsystem eh von sich aus. Mit den Eigenschaften in den Properties lassen sich passende Änderungen vornehmen, die anhand des von ihnen benutzen „Programming Guideline“ einen Einfluß auf die Reports und deren Maßnahmen haben.

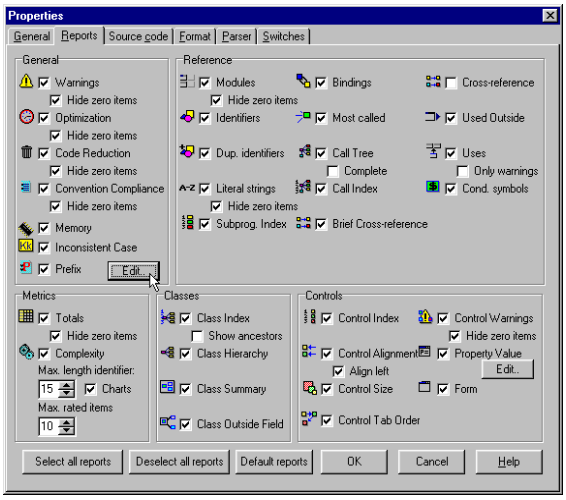

Abb. 3: Je nach Guideline sind Anpassungen nötig

PAL ist ein enormes Werkzeug. Es ist intuitiv, schnell installiert und zügig gestartet. Der Nutzen ist die bescheidene Investition der Zeit und des Geldes wert. Zu wünschen wäre ein Report über duplizierten Code (don’t repeat yourself) oder zusätzlich eine kleine Profiler Funktion. In der soeben erschienenen Version 3 lassen sich gezielter Reports in eigenen Projekten durchführen, ohne jedesmal den kompletten Code analysieren zu müssen. Als Fazit wird PAL zum unentbehrlichen Helfer im Softwarebau. PAL bedeutet im Englischen auch Kollege, was heißen will, sich bei Gelegenheit von einem weiteren digitalen Kollegen über die Schultern schauen zu lassen. Damit ihr Code eleganter und leistungsfähiger wird.

„May the source be with you.“

Literatur & Links:

Pascal Analyzer
http://www.peganza.com/

Namenskonvention
http://www.delphi1.com/Articles/Abbreviations.htm

Spezifikation eines Compilers
http://ag-kastens.upb.de/eli/examples/eli_pascalE.html

Max Kleiner, November 2004
http://max.kleiner.com