

1 Refactoring in Delphi.NET

Als Refactoring betrachtet man allgemein ein Vereinfachen, Verbessern und Stabilisieren einer bestehenden Codestruktur, welche keine Änderung auf das „beobachtbare“ Verhalten der Applikation und dessen Ergonomie bewirken soll.

Wie man auch mit Delphi 8 diese sogenannten Technischen Anforderungen (Skalierbarkeit, Wartbarkeit und Erweiterbarkeit) erreichen kann, will ich mit dieser kompakten Übersicht schrittweise aufzeigen.

Mehrwert auf der Baustelle

Ein Redesign ist hingegen ein grösseres Umstellen von Code und deren Schichten innerhalb eines Projektes. Es ist auf der Stufe Architektur anzusiedeln und hat vielfach Einfluss auf das Systemverhalten, deren Schnittstellen bis hin zu Änderungen des GUI und zugehörigem Dialogfluss.

Hier gerät man zuweilen in Argumentationsnotstand wenn man dem Kunden erklärt, die 3 Wochen Refactoring Aufwand (ohne sichtbare Ergebnisse wohl bemerkt) reduzieren dafür die künftigen Ausbau- und Wartungskosten!

Wir haben kürzlich beim Testen eines Release urplötzlich beim Schliessen der Applikation einen massiven Speicherverbrauch mit anschliessender *kernel32.dll* Schutzverletzung hinnehmen müssen. Harter Schlag. Nach einigen Stunden Suchen inklusive Vergleich des letzten Release stellte sich heraus, dass unsere *infobox.dll* mit einer neuen Graphik im GIF-Format bestückt war, anstelle der alten Unit *jpeg* wurde die neu eingebundene Unit *GIFImage* als Sünder erkannt. Dieses Refactoring, im Sinne einer einheitlichen Grafikverwaltung, war also destruktiv und liess sich dank der Versionsverwaltung und dem Änderungsprotokoll wieder auf den Pfad der Tugend zurückführen. Moral der Geschichte: kein Refactoring ohne Änderungsprotokoll.

Heisst auch, ohne Tools wie Versionsverwaltung oder CASE kann eine Umstellung von Code zum Kreuzworträtsel werden. Ziel eines Refactoring sind robuste und stabile Systemarchitekturen mit klar gegliederten Einheiten von Klassen, Paketen und Komponenten. Da Modell und Code für diese Tools eins sind, fallen einige Refactoring Methoden ganz nebenbei ab. Ein Beispiel sei das Umbenennen von Klassen und Methoden. Nimmt man eine solche Änderung am Code vor oder ändert man das Modell entsprechend, aktualisieren viele Tools die jeweils andere Seite automatisch.ⁱ

Anhand von 5 Refactoring Techniken und deren Bedeutung, die auch in Reviews oder einer Code Inspektion nützlich sind, wollen wir uns nun der .NET Welt nähern.

Ein Interface extrahieren und bilden

Warum und wie Interfaces nützlich sind, da muß ich Sie leider auf mein Buch „Patterns konkret“ verweisen. Ein Interface aus bestehenden Klassen oder Strukturen zu bilden, ist eine der häufigsten Refactoring Aktionen. Auch in Delphi 8 wie in C# ist es möglich, sich mit dem Schlüsselwort `as` einen Zeiger auf ein Interface eines bekannten Objektes zu beschaffen:

```
myIDbConnection As System.Data.IDbConnection
```

.NET kennt jedoch keine Typbibliothek mehr, und auch die Interfaces `IUnknown` und `IDispatch` sowie `CLSID` etc. (Zugriff auf ein COM-Objekt) sind in der „reinen“ .NET-Welt unbekannt. Ein Interface ist somit ein geordneter Referenztyp wie übrigens ein Pointer auch! Klassen können von beliebig vielen Schnittstellen erben, im wesentlichen bauen Sie Klassen, welche die Schnittstellen implementieren.

Da in .NET mit dem CTS (Common Type System) alles auf Objekten basiert und ein Typ besitzt (sogar ein Integer ist ein Objekt und hat einen Werttyp `System.Int32`) gehören auch Patterns und OO-Techniken zum Standard dieser mächtigen Klassenbibliothek.ⁱⁱ

Delphi 8 nutzt aber auch eigene Alias-Namen für Typen, damit für uns bei der praktischen Arbeit nur bekannte Datentypen sichtbar sind. Interessant ist, dass für ältere Delphi Versionen 6 oder 7 eine .NET-Klasse nur ein COM-Objekt darstellt. Wenn also jetzt schon Zugriffe in die neue Welt nötig sind, sind die Methoden einer .NET-Klasse für Delphi W32 nur altbekannte Interface-Methoden eines COM-Objekts.

Superklasse extrahieren

Eine Superklasse muß nicht unbedingt eine Vererbung aufweisen. Mit dem Konzept der Delegates als typsichere Methodenzeiger sind in .NET auch zusätzliche Klassen und Implementierungen möglich und nutzbar. Delegates entsprechen von der Idee den Events aus Delphi W32, hingegen haben events in .NET eine abweichende Bedeutung.

Mit dem Schlüsselwort `events` ist ein Lokalisieren und Verwalten von Delegates gemeint. D.h. ein öffentlicher Zugriff ist nur noch über spezielle Operatoren möglich. Durch diese Nutzung läßt sich sicherstellen, daß nur der Teil einer Anwendung den Event auslöst, der den Event auch definiert hat. Ein Auslösen von Events durch andere Objekte ist somit in .NET nicht erlaubt.

Im Prinzip ermöglicht .NET mit diesen 2 Konstrukten und dem Multicast Event, das Observer Design Pattern und das MVC Architektur Pattern zu implementieren.

Klasse und Unterklasse extrahieren

Mit dieser Refactoring Technik bietet Delphi 8 in der Tat ein neues OO-Konstrukt an. Heißt aber nicht, daß in Delphi W32 diese Technik nicht auch schon möglich war. Das neue Konstrukt heißt `Class Helper` und ist eine spezielle Klasse, welche eine andere Klasse mit Methoden oder Eigenschaften zur Laufzeit erweitert, ohne daß eine Vererbung nötig ist. Auch wird für das Überschreiben von Methoden benötigen wir durch die `Class Helper` nur eine Instanz. Es lassen sich also Methoden aufrufen, die in der Basisklasse gar nicht vorhanden sind.

Als Design Pattern ist dieses Konstrukt ein idealer Kandidat um den Decorator zu implementieren. Erweitere ein Objekt dynamisch mit Methoden und Zuständigkeiten ohne viele neue Unterklassen bilden zu müssen. Ein `Class Helper` hat in Delphi 8 folgenden Klassenkopf:

```
TDecoClassHelper = class helper for TBasicDecorator
```

Klassen in Paketen verschieben

Ein Paket kann eine technische, logische oder Fachliche Gruppierung von Klassen sein und deckt sich mit dem Begriff `Packages` aus der UML.

Für den Architektorentwurf bietet das Paketdiagramm sehr weitreichende Unterstützung an: Die einfache Darstellung hilft, IT-Systeme zu gliedern, Komplexität zu reduzieren und so robuste Systemarchitekturen zu bauen. In den Diagrammen definieren Sie grafisch die Abhängigkeiten und Hierarchiebeziehungen zwischen Paketen. Auf diese Weise kann man unter anderem vorgeben, welche Schnittstellen eines Paketes sich benutzen lassen und welche zu implementieren sind.

In .NET existiert ein in Java angenähertes Paketkonzept, das aber nicht auf gleichnamige Verzeichnishierarchien und Dateinamen beschränkt ist. Die Definition eines Paketes in .NET erfolgt durch das Schlüsselwort `namespace` und beinhaltet nebst Klassen auch die Abgrenzung zu den Typen, muß aber wie gesagt, keine hierarchische Ordnung aufweisen. Betrachten wir einen `Namespace` mal als `Unit-Name`, d.h. jede `Unit` deklariert einen eigenen Namensraum. Ein `Namespace` (Verweis) soll vor allem Namenskollisionen verhindern und wird in Delphi 8 weiterhin mit einem `uses` eingebunden:

```
Uses System.Reflection,  
      System.Runtime.CompilerServices,  
      Borland.Delphi.SysUtils;
```

In Delphi W32 läßt sich ja ein Bezeichner mehrmals verwenden, wobei die Reihenfolge der Unit-Namen in der Uses-Liste die Sichtbarkeit festlegt. Wobei sich Borland entschieden hat, daß auch in Delphi 8 eine Unit die Gültigkeit festlegt und der Namensraum „nur“ die Sichtbarkeit der Unit-Namen regelt.

Pakete bilden, aufbrechen und entkopplern

Logisch erscheint die Refactoring Tatsache, daß abhängige Pakete nicht von solchen abhängen sollen, die noch änderungsanfälliger sind als das Paket selbst.

Diese Technik ist der Skalierbarkeit und dem Deployment einer Anwendung gewidmet und hier kommt die Assembly ins Spiel. Dahinter steckt eine sich selbst beschreibende Gruppierung von gemeinsamen Dateien, die als atomare Einheit Daten in den Bereichen Version, Installation, Ressourcen und Sicherheit besitzt. Borland bringt selbst ein Runtime Package in Form einer Assembly mit: *Borland.Delphi.dll*

In UML ist eine Assembly als Component abzubilden. In den .NET Assemblies lassen sich auch mehrere öffentliche Klassen und weitere Namespaces verpacken. Auf die Gefahr hin den Sachverhalt zu vereinfachen, läßt sich sagen, ein Runtime Package entspricht einer Assembly und ein Namespace dient dazu, existierende Assemblies implizit oder explizit ohne Namenskonflikt einzubinden. Ein Namespace wird oft mit dem Namen der Datei (Assembly) identisch sein, muß aber nicht!

Was heißt nun implizit einbinden. Wenn ich den Namensraum nicht explizit deklariere, sondern durch Abhängigkeiten von anderen Assemblies implizit einbinden lasse, geschieht dies ohne meine Kenntnisse. Mit Delphi 8 läßt sich sogar mit der Einstellung `Link Units = True` eine Assembly auch statisch, explizit in die Anwendung linken!

Um existierende Pakete einzubinden, muß der Compiler über Direktiven (Switches) informiert werden, wo sich die benötigten Assemblies befinden. Somit gibt es grundsätzlich in Delphi 8 keine Unterschiede mehr zwischen einer DLL (Library) und einem Package. Wobei ich hier eine große Klammer öffne: Ein Delphi 8 Package besteht aus MSIL-Code und die CLR (Common Language Runtime) ist dann in der Lage, den Inhalt auszulesen und auf die jeweilige Umgebung hin optimierten Maschinencode zu erzeugen. Ein Delphi W32 Package besteht hingegen bereits nach dem ersten Compilat aus Maschinencode. Diese neue Sichtweise setzt enorme Kenntnisse der .NET Laufzeitumgebung voraus.ⁱⁱⁱ

Qualitätsmessung

Am effizientesten ist ein Refactoring, wenn die meisten Eigenschaften einer Klasse als Property angelegt sind. Dies ermöglicht ein gezieltes Ändern des Property selbst, so daß die abhängigen Methoden sich gekoppelt zwischen den Klassen verschieben lassen. Hier hat .NET eindeutig mit Hejlsberg das Konzept der Properties aus Delphi übernommen, so dass in Delphi 8 kein Zusatzaufwand entsteht.

Wie aber stellt man die Notwendigkeit des Refactoring fest? Denn je mehr Klassen man erhält desto übersichtlicher die Methoden, aber die Gesamtübersicht wird bei erhöhter Anzahl Klassen und Packages auch nicht besser. Die Praxis bestätigt aber: Besser mehr Klassen in den Paketen als zu viele Methoden in den Klassen. Durch die Befolgung des Gesetzes von Demeter können Entwickler ein Refactoring oder eine Dekomposition als nötig überprüfen, d.h. beim nicht Einhalten der 4 Sätze von Demeter sollten Sie mit einem Refactoring beginnen.^{iv}

Nach Demeter ist es also nicht statthaft, Instanzen aus einem aufgerufenen Objekt aufzurufen, die irgendwie global in einer anderen Unit deklariert sind.

Bedenken Sie aber, daß ein Element, das an irgendeiner Position innerhalb einer Unit verfügbar ist, in der gesamten Datei zur Verfügung steht. Wenn Sie also zwei Klassen in derselben Unit definieren, können diese auf die als private deklarierten Methoden der jeweils anderen Klasse zugreifen (friend-Beziehung).

Das Gesetz von Demeter besagt, dass ein Objekt O als Reaktion auf eine Nachricht m, weitere Nachrichten nur an die folgenden Objekte senden sollte:

Refactoring in Delphi.NET

1. [M1] an Objekt O selbst
Bsp.: `self.initChart(vdata);`
2. [M2] an Objekte, die als Parameter in der Nachricht m vorkommen
Bsp.: `visitor.visitElementChartData;`
3. [M3] an Objekte, die O als Reaktion auf m erstellt
Bsp.: `visitor := TChartVisitor.create(cData, madata);`
4. [M4] an Objekte, auf die O direkt mit einem Member zugreifen kann
Bsp.: `perfList.add(period);`
`chartgenCtnr := visitor.TotalStatistic`

Ich habe diesen Sachverhalt mit einem Sequenzdiagramm in Abb. 1 dargestellt, die 4 Sätze entsprechen den Nachrichten M1 bis M4. Die Instanz `ChartGen` sendet an `O` die 4 gültigen Nachrichten und `O` reagiert gemäß Demeter darauf. Überprüfen Sie doch selbst mal ihre wichtigen Codepassagen, ob das Gesetz von Demeter eingehalten wird ;).

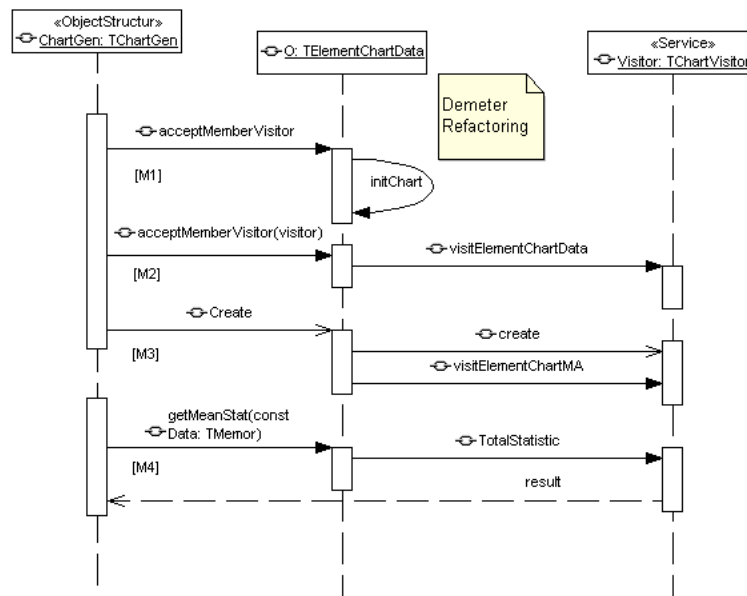


Abb. 1.1: Die erlaubten Zustandswechsel des Objekts

Eine Methode umfaßt nur in den seltensten Fällen die Ausführung einer einzigen Funktion. Wesentlich häufiger sind verkettete Aufrufe mehrerer Methoden und Kaskadierungen möglich, die aber nicht in derselben Klasse angesiedelt zu sein brauchen. Somit wird klar, daß diese Phase mit der Demeterprüfung und anschließendem Refactoring eben stark iterativ geprägt ist und einer wiederholten Selbstprüfung des Refactoring (zu zweit) nichts im Wege stehen sollte. Abschließend solle eine Übersicht der bekannten und nützlichen Refactoring Techniken als Checkliste dienen:

Einheit	Refactoring Funktion	Beschreibung
Package	Rename Package	Umbenennen eines Packages
Package	Move Package	Verschieben eines Packages

Qualitätsmessung

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. 1.2: Refactoring Checkliste

Max Kleiner, Juni 2004

ⁱ Delphi 8 Tools: <http://www.modelmakertools.com/>

ⁱⁱ Patterns konkret; Software & Support Verlag 2003

ⁱⁱⁱ delphi.net Sonderheft; Software & Support Verlag 2004

^{iv} <http://www.delphi-expert.com/castalia2/>