Modulare Anwendungsentwicklung mit Microsoft Access

Ein Vortrag von Paul Rohorzka

bei der 10. Access Entwickler Konferenz

Nürnberg, 6./7. und 20./21.Oktober 2007

Inhalt

A. Einführung

1. Warum modulare Anwendungsentwicklung?

Modular zu entwickeln bedeutet eine Anwendung auf verschiedenen Ebenen in möglichst klar abgegrenzte und verständliche Einheiten zu gliedern.

Vorteile durch modulare Anwendungsentwicklung

a. Bessere Wartbarkeit

Modular entwickelte Anwendungen lassen sich leichter warten, weil Abhängigkeiten zwischen den Teilen der Anwendung minimiert werden.

b. Wiederverwendbarkeit

Die Einheiten von Anwendungen, die in Richtung Modularität entwickelt wurden, beinhalten idealerweise funktional klar abgegrenzte Aufgaben. Daher bieten sich solche Einheiten dafür an, in anderen Anwendungen wieder verwendet zu werden.

Wiederverwendung durch Auslagern von Funktionalität in zentrale Methoden

Wenn ein Code-Fragment, dessen Funktionalität an mehreren Stellen in einer Anwendung benötigt wird, in eine Methode ausgelagert wird, kann der selbe Code von mehreren Programmteilen benützt werden.

Die Auslagerung in eine Methode erfordert die klare Trennung aus dem ursprünglichen Kontext und die Definition einer möglichst schlanken Schnittstelle über die die Methode ihre Eingangswerte erhält und die Ergebnisse zurückliefert.

Im Teil I dieses Textes geht es im Wesentlichen um diese Art von Modularisierung.

Wiederverwendung durch Auslagern in zentrale Komponenten

Benötigen mehrere Anwendungen teilweise die selben Elemente, können diese Elemente aus den einzelnen Anwendungen herausgelöst und in eine zentrale Komponente ausgelagert werden.

Im Teil II dieses Textes werden Wege diskutiert, wie eine Modularisierung im größeren Maßstab realisiert werden kann.

Nicht gemeint: Wiederverwendung durch das Kopieren von Code

Wenn Code aus einem einem Teil einer Anwendung in einem anderen Teil einer anderen oder der selben Anwendung kopiert und dort (eventuell nach einer Anpassung) wieder verwendet wird, spricht man von „Copy&Paste-Programmierung". Durch diese Vorgangsweise entstehen unweigerlich Code-Redundanzen, die die Wartbarkeit verschlechtern.

Diese Art von Wiederverwendung sollte möglichst nicht verwendet werden.

2. Was ist Access?

- Benutzeroberfläche für Jet-/ODBC-Datenbanken
  - Datenbankeigenschaften
  - div. Optionen
  - Tabellen
  - Abfragen
- RAD-Tool (Rapid Application Development) für Datenbankanwendungen
  - Formulare
  - Berichte
  - Makros und Module
  - Designer (grafische Entwurfsansichten)
  - Assistenten

3. Was sind Tabellen?

- Die (einzige) Stelle, an der tatsächlich Daten gespeichert sind
- Standardformulare zur Bearbeitung von Daten in einer Datenbank
  - Automatisch vorhanden
  - Datenblattansicht
  - Reiche Funktionalität
    - CRUD (Create, Read, Update, Delete)
    - Suche
    - Filter
    - Druckfunktionalität

4. Prinzipielle Strategien

a. Zuerst die Datenstruktur!

Bei vielen Access-Anwendungen beginnt das nicht modulare Chaos bereits bei der Vorgangsweise von Beginn an.

Daher:
Das Design einer Access-Anwendung beginnt bei der Datenstruktur.

Eine Vorgangsweise, die sich bewährt hat:

1. Die Datenstruktur auf Papier entwerfen
   - Welche Daten müssen gespeichert werden?
   - Welche Zusammenhänge bestehen zwischen den Daten
   - Großes Papier!
   - Viel Papier!
   - Unterlagen:
     - Pflichten-/Lastenheft
     - Use Cases
     - Formulare (Input)
     - Berichte (Output)

- § Testfälle
- § Altanwendung **L**

2. Die Datenstruktur im Backend implementieren
   - o Benennungsschema festlegen und strikt einhalten (`id` oder `KundenId` oder `idKunde` oder `pkKunde` oder ...)
   - o Tabellen
   - o Felder
     - § Datentypen
     - § Eingabe erforderlich
     - § Leere Zeichenfolge (geänderter Standard ab A2002!)
     - § Indizes (eindeutige Felder, Mehrfelderindizes)
     - § Gültigkeitsregeln (auch auf Tabellenebene)
   - o Referenzielle Integrität
   - o Von vielen verpönt bis verteufelt, IMHO dennoch zu empfehlen:
     - § Beschriftungen
       - Vorteil:
         - o Vorschlagswert für Beschriftungen von Steuerelementen
         - o Das ist oft praktisch, weil bei größeren Anwendungen ein Feldwert oft auf mehreren Formularen und Berichten benötigt wird (Arbeitserleichterung und Konsistenz!)
     - § Nachschlagefelder
       - Kritikpunkte (die der Autor nicht teilt), z.B. 10 Command-ments
         - o Es werden Daten angezeigt, die nicht in der Tabelle vorhanden sind.
         - o Man geht daher leicht von falschen Voraussetzungen (z.B. beim Formulieren von Abfragebedingungen) aus.
         - o Um statt der Fremdschlüsselwerte sprechende Informationen zu sehen, kann man sich sehr schnell mittels Autoformular und Kombinationsfeldassistenten entsprechende Funktionalität zusammenklicken. Gegenargument: Aber warum sollte ich, wenn ich das ganz automatisch haben kann?
       - Trotzdem für Profis vorteilhaft:
         - o „Ich bin Mensch und will keine Fremdschlüsselwerte sehen, weil sie sagen mir nichts!"
         - o „Ich weiß, dass ich meine Daten sehr gut normalisiere und mir ist daher klar, dass ich hier nicht ausschließlich die in dieser Tabelle tatsächlich gespeicherten Daten sehe."
         - o Vorbelegung von Kombinations-/Listenfeldern in Formularen
         - o Aussagekräftige Daten bei der Entwicklung bzw. Fehlersuche

3. Mithilfe einiger möglichst realistischer Testdatensätze typische Szenarien auf Tabellenebene durchspielen
    - Fragen, die hier beantwortet werden können:
        - Kann ich jede Information eingeben?
          (Nicht, ob es auch praktisch ist!)
        - Sind unsinnige Werte möglichst verhindert?
          (Datentypen, Gültigkeitsregeln auf Feld- und Tabellenebene, Referenzielle Integrität)
        - Muss ich weiter normalisieren?
          (Gebe ich oft identische Daten ein? Müsste ich manche Informationen mehrfach eingeben können? Lassen sich Werte aus anderen Feldern ableiten?)
    - Damit Validierung von
        - Datenstruktur
        - Datentypen
        - Gültigkeitsregeln

Überlegungen zum UI sind hier ABSOLUT FEHL am Platz!

Solche deplatzierten Überlegungen spiegeln sich Aussagen wider wie:

„Ich hab das Feld deshalb angelegt, weil es mir eine einfachere Darstellung im Endlosformular/Listenfeld ermöglicht."

b. Features möglichst deklarativ implementieren

Beispiel: Constraints in der Datenstruktur:

- Nicht:
  bei jedem Zugriff auf ein Feld kontrollieren, dass es nicht leer gelassen oder nachträglich geleert wird
- Sondern:
  das Feld auf Tabellenebene erforderlich machen.

Aber auch: SQL

- Nicht:
  „Mache eine Schleife über alle Datensätze der Kundentabelle. Wenn du bei einem Datensatz siehst, dass der Nachname mit E beginnt und die Postleitzahl eine Wiener Postleitzahl ist, dann füge Nachname, Vorname und Adresse an die Ergebnistabelle an."
- Sondern:
  Liefere mir Nachname, Vorname und Adresse aller Wiener Kunden deren Nachname mit E beginnt:

```
SELECT Nachname, Vorname, Adresse
WHERE Nachname LIKE "E*"
  AND PLZ BETWEEN 1000 AND 1999
```

B. Teil I - Modularität im Code

1. Code kleinräumig – Statement, Methode, Eigenschaft

a. Fehler sollen gar nicht passieren können

„Freiheiten möglichst beschränken"

Kleinst möglicher Gültigkeitsbereich

Die Komplexität und Wartbarkeit einer Software hängt u.a. sehr stark davon ab,

- wie klar die Teile von einander zu unterscheiden sind
- dass die Teile ihre Implementierungsdetails voreinander verbergen
- dass die Schnittstellen zwischen den Teilen möglichst eng gestaltet und gut dokumentiert sind.

Daher

- Felder (Variablen auf Modulebene) immer `Private` deklarieren
  - Empfehlung: Zugriffsmodifizierer explizit angeben (außer in Methoden kein `Dim` verwenden (auch kein `Const` ohne Zugriffsmodifizierer)
- Die Sichtbarkeit von Methoden und Eigenschaften möglichst stark einschränken
  - Interne Details `Private` deklarieren
    - Auch private Eigenschaften können sinnvoll sein!
  -
- Alle Zugriffsmodifizierer explizit angeben (Default ist meist `Public`!)
- Bei Systemen aus mehreren Komponenten kann auch der Modifizierer `Friend` (sichtbar innerhalb der Komponente) sinnvoll sein.

Parameter nur bewusst und explizit `ByRef` definieren

Funktion mit einem Parameter (implizite Übergabe als Referenz):

```
Private Function GetSteuer(Summe As Currency) As Currency
   Summe = ...      ' Hier kann der Wert der übergebenen Variable geändert werden!
End Function
```

Aufruf:

```
   Dim curSumme As Currency

   curSumme = ...
   ... = GetSteuer(curSumme)
   ' Hier könnte curSumme bereits verändert worden sein!
```

Stattdessen:

```
Private Function GetSteuer(ByVal Summe As Currency) As Currency
   Summe = ...   ' Hier kann nur der Wert der lokalen Variable geändert werden,
                 ' der Wert der übergebenen Variable kann nicht verändert werden.
End Function
```

Da `ByRef` in Visual Basic Classic der Standard ist, aber meistens nur `ByVal` benötigt wird, sollte `ByVal` explizit angeben. Nur wenn tatsächlich eine Übergabe per Referenz benötigt wird, das Schlüsselwort `ByRef` angeben.

Optionale Parameter nur mit Maß und Ziel weglassen

Im Auge behalten, ob bzw. wie sich das Weglassen von optionalen Parametern auf die Funktionalität auswirkt.

Typisches Beispiel in Access:

```
DoCmd.Close  ' Was wird hier geschlossen?
```

Es ist hier nicht klar ersichtlich, was (welches Objekt) geschlossen werden soll.

Und - wesentlich schlimmer - es ist auch nicht klar, welches tatsächlich geschlossen wird. Ohne Angabe von Parametern wird das aktive Objekt (das den Fokus besitzt) geschlossen.

Ggflls. kann das aber ein anderes als das gewünschte sein, weil der Benutzer mittlerweile schon den Fokus auf ein anderes Objekt verschoben hat (siehe Beispiel im Demo)!

Stattdessen:

```
DoCmd.Close acForm, Me.Name      ' Aha! Ein Formular, und zwar „ich“ bin dran!
```

Klammern für Methoden, keine Klammern für Eigenschaften

Die Art des Element-Typs explizit angeben:

- Methode (!) `CurrentDb()`
  vs.
- `CurrentDb` – sieht aus wie Eigenschaft, ist es aber nicht!

Hinweis: Diese Unterscheidung ist aufgrund der eigenwilligen Klammernsetzungsregeln in VB Classic allerdings nicht konsequent durchzusetzen, sofern man nicht `Call` verwenden möchte.

Einschub:

Kritisch gegenüber der Verwendung von `CurrentDb()` sein:

- Bei `CurrentDb()` wird jedes Mal ein neues Objekt zurückgeliefert (aber nicht die Datenbank kopiert, wie auf einer Website zu lesen ist **J**).
- Abhilfe:
  - In zusammenhängenden Codeabschnitten eine Hilfsvariable verwenden oder
  - Singleton-Pattern verwenden (Newsgroup-Sprech: „CurrentDbC“ nach einem Vorschlag von Michael Kaplan - siehe Google)

b. Vorhandene Fehler vom Compiler erkennbar machen

Vermeidung von Literalen

Triviale Literale abhängig vom Kontext

Je nach Kontext können triviale Literale sein:

- 0, 1, -1
- 2 (als Faktor oder Divisor)
- "" (leerer String) - hier kann in VBA-Methoden aber auch vbNullString verwendet werden (ist aber nicht das selbe!). Vorsicht bei Aufruf von Funktionen aus C-DLLs (z.B. Win-API)!

Nichttriviale Literale

In nichttriviale Literale stecken Informationen über Access-Objekte und problemspezifische Gegebenheiten:

- Tabellennamen
- Formularnamen
- Dateiendungen
- Meldungstexte
- ...

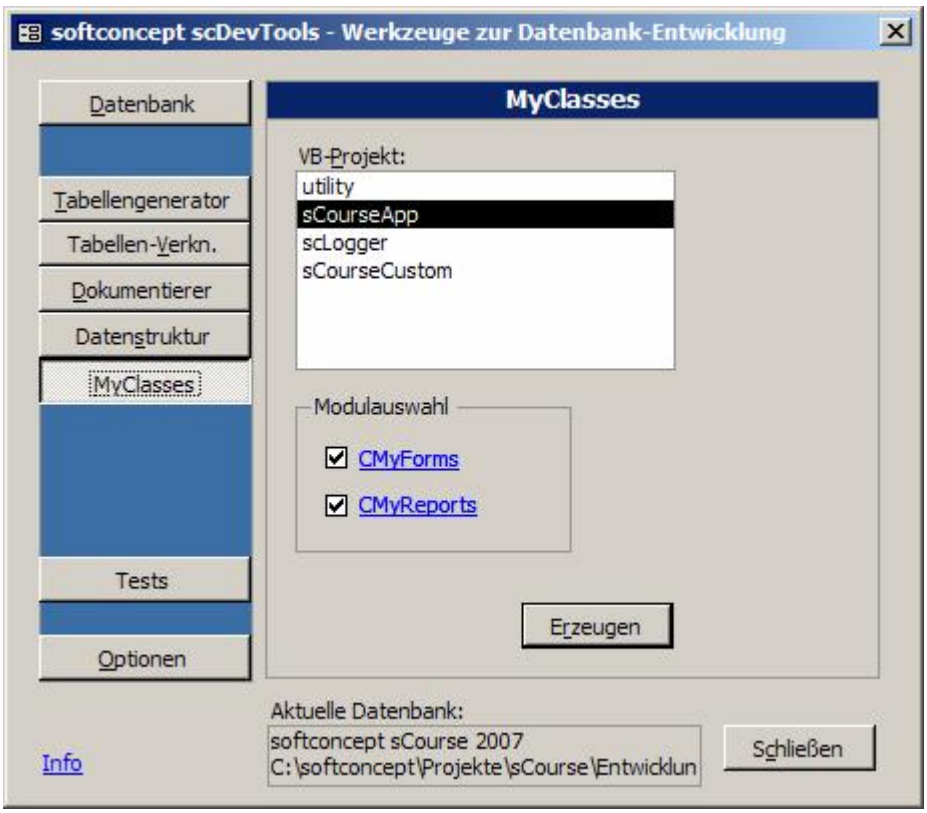
Solche Literale wenigstens in Form von Konstanten zentral definieren, und zwar dort wo sie hingehören (als Eigenschaft betrachten):

- Eigenschaft der gesamten Anwendung (z.B. Titel der Anwendung für `MsgBox` - noch besser natürlich: eigene `MsgBox()` Ersatz-Methode):
  Als globale Konstante oder als Eigenschaft in einer Singleton-Instanz einer Anwendungs-Klasse.
- Eigenschaft eines Moduls (z.B. Name des Moduls für Fehlermeldungen):
  Als Modul-Konstante
- Nur in seltenen Fällen Konstanten auf Methodenebene definieren (ev. für Konstanten, die nur in dieser Methode Bedeutung haben – siehe Demo).

Tool: MyClasses

Die Situation bei diesen spezifischen Literalen lässt sich ev. auch durch die Verwendung von Tools entschärfen, die Werte für solche Konstanten auf Knopfdruck z.B. aus der AllForms-Auflistung der Datenbank erzeugen.

Beispielsweise erzeugt das Tool „MyClasses" (ein Teil der softconcept scDevTools) auf Knopfdruck Kapselklassen, durch die Zugriff auf die Formulare und Berichte mithilfe strenger Typisierung möglich ist.



Das Tool erzeugt die beiden Klassen CMyForms und CMyReports (und jeweils eine Kapselklasse für Formulare und eine für Berichte), die jeweils für jedes Formular oder jeden Bericht eine Eigenschaft besitzt, deren Name dem Namen des Access-Objekts entspricht. Vorsilben „frm" und „rpt" werden dabei automatisch entfernt.

Die literalen Strings mit den Namen der Formulare und Berichte stehen nun nur mehr in den vom Tools jederzeit neu generierbaren Modulen.

Nach dem das Tools die Klasse `CMyForms` und die Eigenschaft `MyForms` in einem Standardmodul erzeugt hat, können Codes wie dieser

```
DoCmd.OpenForm "frmKundenDetails“
...
DoCmd.Close acForm, "frmKundenListe"
...
DoCmd.Close acForm, Me.Name
```

durch folgende Literal-freie Codes abgelöst werden:

```
MyForms.KundenDetails.OpenIt()
...
MyForms.KundenListe.CloseIt()
...
MyForms.Item(Me).CloseIt()
```

Damit kann also sowohl auf die fehleranfälligen Literale als auch auf die Verwendung von Methoden des `DoCmd`-Objekts in Zusammenhang von Öffnen und Schließen von Formularen verzichtet werden.

Diese Vorgangsweise bietet zwei große Vorteile:

- Vertipper in Formularnamen werden vom Compiler erkannt, nicht erst zur Laufzeit
- Intellisense kennt plötzlich auch meine Formulare (weswegen der erste Punkt kaum mehr auftreten wird)!

Literale lassen sich eventuell zur Laufzeit aus anderen Informationen ableiten:

Beispiel:

Ursprüngliche Variante mit zwei von einander abhängigen Literalen (die sich potenziell widersprechen können):

```
If Left(tdf.Connect, 10) = ";DATABASE=" Then
   …
End If
```

Abgeänderte Variante mit einem Literal zentral an einer Stelle definiert und sprechend benannt:

```
Const connectPrefixJet As String = ";DATABASE="
If Left(tdf.Connect, Len(connectPrefixJet)) = connectPrefixJet Then
   …
End If
```

Auch triviale Literale lassen sich manchmal vermeiden

Beispiel (`For`-Schleifen über alle Elemente eines Arrays):

```
   For i = 0 To mc_intAnzahl - 1
      ... = ... * aZahlen(i)
   Next i
```

Aussage: Eine Schleife „von einer Grenze bis zu einer anderen Grenze".

Stattdessen:

```
   For i = LBound(aZahlen) To UBound(aZahlen)
      ... = ... * aZahlen(i)
   Next i
```

Aussage: Eine Schleife von „über alle Elemente" → wesentlich klarer und weniger fehleranfällig.

Im Speziellen: Dot (.) statt Bang (!)

Zugriff auf Steuerelemente und Felder in Formularen nicht über die Controls-Auflistung, sondern über die jeweilige Eigenschaft:

Beispiel:

```
   Me!lblInfo = ...
```

ist nichts anderes als eine Kurzschreibweise für

```
   Me.Controls.Item("lblInfo").Caption = …
```

Da ist also ein String-Literal versteckt! Wenn man sich hier vertippt oder das Steuerelement nicht mehr existiert oder umbenannt wurde, fällt der Fehler erst zur Laufzeit auf. Weiters ist der Typ des Steuerelements erst zur Laufzeit bekannt (was ist, wenn das angesprochene Steuerelement kein Label (mehr) ist, und daher auch keine Eigenschaft Caption hat? → Laufzeitfehler).

Stattdessen:

```
   Me.lblInfo.Caption = ...
```

Warum funktioniert das?

Weil Access für jedes zur Entwurfszeit bereits vorhandene Steuerelement und Feld der Datensatzherkunft der Formularklasse (Form_<Formularname>) eine streng typisierte Eigenschaft verpasst. Wenn sich nun der Name oder der Typ ändern, oder das Steuerelement umbenannt oder gelöscht wird, tritt bereits beim Kompilieren ein Fehler auf.

Hinweis:

Diese Variante funktioniert nach intensivster Erfahrung des Autors bei Formularen unter Access 2000 absolut zuverlässig. Mit Access 2000 erstellte MDEs laufen absolut problemlos auch unter Access 2002 und Access 2003.

Probleme dürfte es mit dieser Eigenschafts-Notation bei der Entwicklung im Access 2000-Dateiformat unter höheren Access-Versionen geben. Hier wird von häufigen und reproduzierbaren Abstürzen berichtet, die sich durch Verwendung von `Me!` statt `Me.` lösen ließen.

Konsequenz des Autors: Immer in der Access-Version (nicht nur Dateiformat!) entwickeln, die später die MDE haben wird.

Strenge Typisierung

Möglichst den „engsten" Datentyp verwenden.

Boolean statt Integer

Schlechtes Beispiel aus Access: Der Parameter Cancel in Form_Unload, Control_BeforeUpdate, etc.:

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
   ...
End Sub
```

Warum hier Integer?

Klarer wäre:

```
Private Sub Form_BeforeUpdate(Cancel As Boolean)
   ...
End Sub
```

Currency statt Double

Currency kann vier dezimale Nachkommastellen *exakt* abbilden, Double und Float können das nur in Spezialfällen (Linearkombination von Zweier-Potenzen). In Double oder Float gespeicherte Werte sind also im Allgemeinen inhärent falsch!

Currency kann daher z.B. auch gut für Prozentsätze verwendet werden.

Dass Currency keinerlei Währungsinformation mit sich führt, sei hier nur der Vollständigkeit halber erwähnt.

Zugriff auf anderes Formular

Beispiel: Bezug auf ein Steuerelement eines Unterformulars aus dem Hauptformular

Statt untypisiert bei jedem Zugriff:

```
   ... = Me.frmsBestellungen.Form.Anzahl.Value            ' Schwach typisiert
   ... = Me.frmsBestellungen.Form.Artikel.Value           ' Schwach typisiert
   ... = Me.frmsBestellungen.Form.Rabatt.Value            ' Schwach typisiert
```

Streng typisierter Zugriff (schwach typisiert nur an einer einzigen Stelle):

```
   Dim objfrmsBestellungen As Form_frmKunden_frmsBestellungen

   ' Folgende Zuweisung ist die letzte nicht streng typisierte Stelle:
   Set objfrmsBestellungen = Me.frmsBestellungen.Form      ' Schwach typisiert
   ... = objfrmsBestellungen.Anzahl.Value                  ' Streng typisiert
   ... = objfrmsBestellungen.Artikel.Value                 ' Streng typisiert
   ... = objfrmsBestellungen.Rabatt.Value                  ' Streng typisiert
   Set objfrmsBestellungen = Nothing
```

Noch besser:

Keinen direkten Zugriff auf Steuerelemente eines anderen Formulars, stattdessen eine Eigenschaft vorsehen (logische statt technische Ebene).

Im Unterformular (Beispiel von oben):

```
   Public Property Get Artikelname() As String
      Artikelname = Nz(Me.Artikel.Value, "")
   End Property
```

Im Hauptformular:

```
   Dim objfrmsBestellungen As Form_frmKunden_frmsBestellungen

   Set objfrmsBestellungen = Me.frmsBestellungen.Form
   ... = objfrmsBestellungen.Artikelanzahl
   ... = objfrmsBestellungen.Artikelname
   ... = objfrmsBestellungen.Artikelrabatt
   Set objfrmsBestellungen = Nothing
```

c. Für die eigentliche Problemlösung möglichst wenig programmieren

Stattdessen:

- Designer verwenden
  - Voreinstellungen für Formulare in der Entwurfsansicht vornehmen
  - Abfragen nicht per SQL in VBA zusammenbasteln, sondern gespeicherte Abfragen verwenden – siehe unten
- Code generieren lassen
  - Mithilfe von Code-Generatoren gefährliche Teile generieren lassen
  - auf eine vom Compiler überprüfbare Ebene bringen
- Frameworks anwenden
  - „gefährliche" Codeteile zentralisiert spezifizieren
  - und dann per Eigenschaft o.ä. darauf zugreifen – siehe CMyClasses oben

d. Benennungen

Deutsch (lokale Sprache) oder Englisch?

Bei Benennungen können zumindest zwei Gruppen unterschieden werden:

- Allgemein bekannte Begriffe in Englisch belassen:
  - `GetXxx`, `SetXxx`, `CreateXxx`, `ListXxx`, `RemoveXxx`, `DeleteXxx`, `InsertXxx`, `AddXxx`, `RereshXxx`, `UpdateXxx`, `InitializeXxx`, `SetupXxx`, `DoXxx`, `MakeXxx`, ...
- Domänenspezifische (fachspezifische) Begriffe in der Originalsprache belassen:
  - Fachbereich, Antragsteller, Ausschuss, Dienststelle, Ware, ...

Die dadurch entstehenden englisch/deutschen Mischungen sind vielleicht gewöhnungsbedürftig aber durchaus sinnvoll (`GetAntragsteller()`, `AddBuerger()`, `UpdateFoerderfall()`, ...).

Die Übersetzung nicht sehr geläufiger domänenspezifischer Begriffe auf Biegen und Brechen ins Englisch birgt im Detail großes Potenzial für eventuell fatale Begriffsverwirrungen!

Konsistente Benennung von Datenbankobjekten

> Mit „Datenbankobjekte" sind die Tabellen, Abfragen, Formulare, Berichte und Code-Module einer Access-Anwendung (also alles, was im Datenbankfenster zu sehen ist) gemeint, nicht VB-Objekte.

Überlegungen für alle Typen

- Präfixe ja/nein?
- Wenn Präfixe: wie detailliert?
  - Lokale vs. temporäre Tabellen vs. Code-Tabellen
  - Auswahl vs. Aktionsabfragen
  - Unterformulare vs. Hauptformulare, etc.)
- Unterstriche?
- Nur Buchstaben ohne Umlaute und Ziffern

Tabellen

- Einzahl oder Mehrzahl (tblKunde oder tblKunden?)

Benennung von Methoden und Eigenschaften

Entscheidend ob Code in eine Eigenschaft oder Methode kommt ist die unterschiedliche Semantik:

- Eigenschaften
  - geben Auskunft über einen bestimmten Datenwert
  - und/oder legen diesen fest.
  - Änderungen von anderen Eigenschaften sollten nicht vorkommen.
- Methoden
  - stoßen eine Aktion an.
  - Eventuell wird der komplette Zustand des Objekts geändert
  - oder die Aktion hat noch eine andere, weiter reichende Auswirkung

Auswirkung auf die Benennung:

- Eigenschaften nennen den Datenwert (z.B. `Nachname`, `Value`,)
- Methoden benennen eine Aktion (z.B. `RechneRabatt()`, `GetKunde()`)

Übliche Konventionen beibehalten

- `GetXxx()` – oder doch eine schreibgeschützte Eigenschaft verwenden?
- `SetXxx()` – oder doch eine Eigenschaft nur zum Schreiben verwenden?
- `CreateXxx()`

2. Wohin mit dem Code? – Module

a. Allgemeine Punkte

Minimale Schnittstellen zwischen den Code-Teilen

Die Code-Teile in einer Anwendung sollten durch möglichst genau definierte (Sprachelemente!) und möglichst „schmale" Schnittstellen miteinander kommunizieren:

Gut geeignet:

- Parameterübergabe
- Rückgabewerte
- Ereignisse (siehe Demo-Beispiele)
- Interfaces (siehe Demo-Beispiele)

Schlecht wartbar:

- Globale Variablen
- Referenzierungen „aus dem Off"

Konkret: Keine Referenzierungen „aus dem Off"

... wie `Forms!frmKunden!Titel`

Problem dabei:

- Es entstehen sehr enge Abhängigkeiten zwischen den Code-Teilen und Access-Objekten.
- Das „KnowHow" über Access-Objekte (Formular- und Steuerelementnamen und -typen) ist über die ganze Anwendung verstreut.
- Änderungen (z.B. an Objektnamen) haben Auswirkungen an eigentlich nicht betroffenen Stellen.

Stattdessen:

- Wenn der Code das UI steuert, dann den Code CBF schreiben, dort Referenzierungen direkt über `Me` (wie `Me.Titel.Value`) vornehmen
- Wenn Code in Modul ausgelagert ist, und dort nur einzelne Werte aus Steuerelementen des Formulars benötigt werden, dann die Werte als Parameter übergeben.
- Wenn der Code tatsächlich ein Steuerelement- oder Formularobjekt benötigt, dann die Referenz auf das Steuerelement bzw. Formular übergeben (`Me.MeinSteuerelement` bzw. `Me`)

Beispiel:

Statt:

```
' In Standard-Modul
Public Sub MachWasMitMeinemKuhlenFormular()
...
   Forms!frmIchBinKuhl!SuperTextBox.Value = ...
   Forms!frmIchBinKuhl!SuperTextBox.BackColor = ...
...
End Sub
```

```
' In Form_frmIchBinKuhl
   ...
   MachWasMitMeinemKuhlenFormular
   ...
```

So

```
' In Standard-Modul
Public Sub MachWasMitMeinemKuhlenFormular(ByVal Textfeld As Access.TextBox)
...
   Textfeld.Value = ...
   Textfeld.BackColor = ...
...
End Sub
```

```
' In Form_frmIchBinKuhl
   ...
   MachWasMitMeinemKuhlenFormular Me.SuperTextBox
   ...
```

b. Trennung logischer Einheiten

Technische Ebene - fachliche Ebene

Auf technischer/implementatorischer Ebene:

- „Rufe die Methode `XYZ` aus, wenn jemand auf die Schaltfläche `btnSave` klickt."

Auf logischer Ebene

- „Führe die Aktion ABC aus, wenn der Benutzer speichern will."

Umsetzung in Access:

- Möglichst rasch die technische Ebene verlassen und auf die fachliche Ebene kommen
- Mittel:
  - Technische Details in Methoden kapseln

    Beispiel:
    Statt folgendem Code (technische Ebene):

```
Private Sub btnEinfuegen_Click()
   Dim strSQL As String

   strSQL = "INSERT ... """ & Me.Nachname.Value & """"
   CurrentDb().Execute strSQL, dbFailOnError
End Sub
```

... die technischen Details in eine Methode extrahieren und der Methode einen fachlich sinnvollen Namen geben:

```
Private Sub btnEinfuegen_Click()
' Fachliche Perspektive
   AktuellenDatensatzEinfuegen Me.Nachname.Value
End Sub

...

Private Sub AktuellenDatensatzEinfuegen(ByVal Nachname As String)
   Dim strSQL As String

' Technische Perspektive
   strSQL = "INSERT ... """ & Nachname & """"
   CurrentDb().Execute strSQL, dbFailOnError
End Sub
```

  - Ereignisse
  - Tool-Methoden und -Klassen

- Ergebnis
  - Die einzelnen Code-Abschnitte (meist Methoden) sind klarer verständlich
  - Code liest sich leichter – „fast wie Prosa"

Verschieden fachliche Bereiche

Eventuell kann Code je nach Art der Anwendung auch nach verschiedenen fachlichen Gesichtspunkten gruppiert werden.

Funktionalität – UI

Code nur dann in CBF, wenn er unmittelbar mit der Steuerung des UI zu tun hat. Jeglicher Businesscode unabhängig von Formularen in externe Module unterbringen.

Als Nagelprobe kann folgende Überlegung gelten:
Könnte ich die gesamte Funktionalität meiner Anwendung ohne Formulare, nur durch geeignete Aufrufe von Code aus Nicht-UI-Elementen (Standard- und Klassenmodule) anbieten?

3. Fehlerbehandlung

a. Wo sollte eine Fehlerbehandlung durchgeführt werden?

Die Frage wo überall eine Fehlerbehandlung eingesetzt werden soll, wird meistens zu wenig betrachet:

- Fehlerbehandlung in keiner Methode?
  - Es wird schon nix passieren (wird es aber doch!)
  - An manchen Stellen ist Fehlerbehandlung essenziell (siehe unten)!
- In jeder Methode/Eigenschaft?
  - Sicher ist sicher (?)
  - Viel zu viel unnötiger Code-Overhead!

Fehlerbehandlung mit Maß und Ziel eingesetzt:

- Dort wo
  - auch im Fehlerfall wieder aufgeräumt werden muss
    - Connection oder Datei schließen
    - `DoCmd.Echo True/False`
    - `DoCmd.SetWarnings True/False`
    - ...
  - ein Fehler mit Zusatzinformationen angereichert werden soll
  - ein eigener Fehler geworfen werden soll
  - in Ereignisprozeduren nicht triviale Aufrufe stehen

Was passiert mit nicht abgefangenen Fehlern?

Nicht durch einen Fehlerhandler abgefangene Laufzeitfehler werden an den aufrufenden Code weitergereicht. Ist dort auch kein Fehlerhandler vorhanden, wird der Fehler wiederum nach oben gereicht, bis ein Fehlerhandler gefunden wird. Wird bis an die oberste Stelle kein Fehlerhandler gefunden, zeigt Access den Fehler in einer Standard-Fehlermeldung an.

Daher sollten zumindestens in jeder Ereignisprozedur, eine Fehlerbehandlung stattfinden. Ausnahme sind Ereignisprozeduren, die einen trivialen Aufruf einer Methode beinhalten, wenn in der aufgerufenen Methode eine Fehlerbehandlung durchgeführt wird.

b. Benutzerfeedback in der Fehlerbehandlung

Auch die Fehlerbehandlungs-Codes dürfen die Trennung zwischen Funktionalität und Benutzerinteraktion nicht verletzen.

Fehlerbehandlungen in Nicht-UI Modulen dürfen daher keine MessageBoxen anzeigen!

c. Form der Fehlerbehandlung

Klassische Form

Die klassische Form eines Fehlerhandlings sieht folgendermaßen aus:

```
Private Sub MyMethod()
On Error Goto Err_

   ' Eigentlicher Nutzcode

Exit_:
   Exit Sub

Err_:
   ' Fehler anzeigen, protokollieren, ...
   Resume Exit_
End Sub
```

Alternative Fehlerbehandlung I

Da im eigentlichen Fehlerabschnitt (Label `Err_`) oftmals auf bestimmte Fehlercodes (`Err.Number`) reagiert werden muss, kann folgende alternative Variante verwendet werden. Damit wird auch die Umkehrung der Programmflussrichtung (durch `Resume Exit_`) vermieden werden:

```
Private Sub MyMethod()
On Error Goto Exit_

   ' Eigentlicher Code

Exit_:
   Select Case Err.Number
      Case 0:
         ' Eventuell noch Aktionen im Erfolgsfall
      Case 2501:
         ' Behandlung spezifischer Fehler
      Case Else:
         ' Nicht spezielle Fehler behandeln
   End Select
End Sub
```

Alternative Fehlerbehandlung II

Ein Problem bei Fehlerhandlern ist eventuell das zentralisierte Aufräumen sowohl im Erfolgs- wie auch im Fehlerfall. Mit Hilfe von Gosub/Return lässt sich auch dieses Problem lösen:

```
Private Sub MyMethod()
On Error Goto Exit_

   ' Eigentlicher Code

Exit_:
   Select Case Err.Number
      Case 0:
         ' Eventuell noch Aktionen im Erfolgsfall
         Gosub CleanUp
      Case 2501:
         ' Behandlung spezifischer Fehler
      Case Else:
         ' Nicht spezielle Fehler behandeln
         Gosub CleanUp
         Err.Raise ...    ' Fehler wieder aufwerfen
   End Select
   Exit Sub

CleanUp:
   ' Aufräumarbeiten hier durchführen
   Return
End Sub
```

4. Abfragen: SQL im Code vs. gespeicherte Abfragen

Vorteile gespeicherter Abfragen:

- Interaktive Manipulation (Joins!)
- Interaktiver Test
- Leichter verstehbar
- Leichter wartbar
- Abfrageplan ist schon gespeichert

Nachteile gespeicherter Abfragen:

- SQL-Code ist (mit Boardmitteln) nicht durchsuchbar
- Mehr Code zum Ausführen einer Aktionsabfrage
    - lässt sich aber leichter verallgemeinern

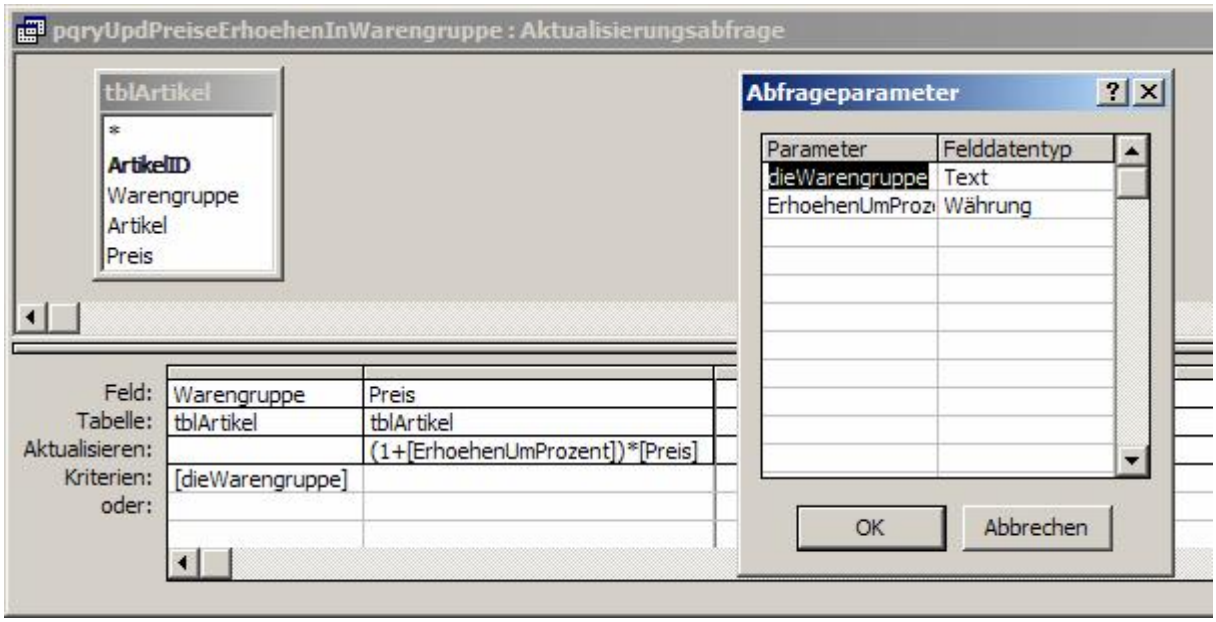
Beispiel (Ausführen einer Aktionsabfrage zur Erhöhung der Preise aller Artikel einer bestimmten Warengruppe):

Herkömmliche Variante:

```
Dim strSQL As String
Dim dbs As DAO.Database

strSQL = "UPDATE tblArtikel " & _
         "SET Preis = " & Str(1 + curErhoehung) & " * Preis " & _
         "WHERE Warengruppe = """ & strWarengruppe & """"
Set dbs = CurrentDb()
dbs.Execute strSQL, dbFailOnError
... = dbs.RecordsAffected
Set dbs = Nothing
```

Stattdessen Aktionsabfrage mit Parameter erstellen:



Aufruf im Code:

```
Dim dbs As DAO.Database
Dim qdf As DAO.QueryDef

Set dbs = CurrentDb()
Set qdf = dbs.QueryDefs("pqryUpdPreiseErhoehenInWarengruppe")
qdf.Parameters("dieWarengruppe").Value = strWarengruppe
qdf.Parameters("ErhoehenUmProzent").Value = curErhoehung

qdf.Execute dbFailOnError
... = qdf.RecordsAffected

If Not (qdf Is Nothing) Then qdf.Close
Set qdf = Nothing
Set dbs = Nothing
```

Auch hier kann eine Generierung von Zugriffsklassen, die für jeden Parameter der Abfrage eine eigene entsprechend benannte Eigenschaft zur Verfügung stellt in Betracht gezogen werden.

Da dabei aber für jede Abfrage eine eigene Klasse generiert würde, muss bei großen Anwendungen das Mengengerüst und die Maximalanzahl von Modulen in einer Access-Datei im Auge behalten werden.

C. Teil II - Modularität im Design

1. Bibliotheken und andere Komponenten

a. Warum Bibliotheken?

Die Entwicklung unter Verwendung von Bibliotheken hat folgende Auswirkungen:

Vorteile

- Wiederverwendbarkeit von Code
  - Allgemein verwendbarer Code muss nicht in jedes Projekt hineinkopiert werden.
- Nachliefern von Funktionen
  - Plugins, AddIns
  - Programm-Module
- Verwendung fremder Funktionalität ohne Code in der eigenen Anwendung zur Verfügung haben zu müssen (MDE/ACCDE).

Nachteile

- Mehr als eine Datei muss verteilt werden
  - Durch die Auslagerung von Code in eine oder mehrere Bibliotheken reicht es nicht mehr, die eigentliche Anwendung auszuliefern. Stattdessen müssen zusätzlich zur eigentlichen Frontend-Datei auch noch alle Bibliotheksdateien mit ausgeliefert werden.
- Verweis-Problematik
  - Um eine Bibliothek mit Early-Binding verwenden zu können, muss die Hauptanwendung einen Verweis auf die Bibliothek haben. Diese Verweise können aber verloren gehen, wenn z.B. der Pfad der Bibliothek am Kunden-Rechner nicht der selbe wie der am Entwicklungs-Rechner ist. In diesem Fall versucht Access, den nicht mehr zu findenden Verweis auf den neuen Ort der Bibliothek umzulenken. Dazu sucht es im System-Verzeichnis, im Access-Installationsverzeichnis und im Verzeichnis, in dem auch die Datenbank mit dem kaputten Verweis liegt.

Erkenntnisse

- Änderungen an einer MDB-Bibliothek
  - Wenn in einem Projekt ein Verweis auf eine im MDB/ACCDB- (oder MDA-) Format vorliegenden Bibliothek gesetzt wird, kann der Code der Bibliothek auch im referenzierenden Projekt bearbeitet werden.
  - Diese Änderungen werden aber *nicht abgespeichert!*
- `CurrentDb()` vs. `CodeDb()`
  - Wenn in der Bibliothek `CurrentDb()` verwendet wird, ist damit immer das „Hauptmodul" gemeint. Wenn z.B. Access-Objekte aus der Bibliothek angesprochen werden sollen, muss stattdessen `CodeDb()` verwendet werden!
    - Eigentlich könnte man sich immer angewöhnen `CodeDb()` statt `CurrentDb()` zu verwenden, außer genau bei Bibliotheken und AddIns, die auf Access-Objekte der Hauptanwendung referenzieren müssen.

- `DoCmd`-Methoden funktioniert oft nicht
    - Einige Methoden des `DoCmd`-Objekts, die direkt auf Datenbankobjekte zugreifen, verwenden aus einer Bibliothek nicht wie erwartet, weil intern offenbar `CurrentDb()` verwendet wird.

2. Vorschlag für eine modulare Anwendungsarchitektur in Access

Im Folgenden wird beschrieben, wie eine komponentenbasierte Anwendungsarchitektur unter Microsoft Access implementiert werden kann.

a. Merkmale

Diese Struktur bietet folgende Merkmale:

- Komponentenbasiert
    - Einzelne Bestandteile der Anwendung können in unterschiedlichen Dateien implementiert werden. Diese Aufteilung bietet Vorteile wie Wiederverwendung von Code in Form von Bibliotheken, Auslagerung von Teilen der Funktionalität in MDB-Dateien (Entwurfsmodus für den Endbenutzer) und das Nachrüsten von Funktionalität in Form von Modulen.
- Modularisierung
    - Verschiedene Teile der Anwendung, die eventuell vom Endkunden getrennt lizensiert werden müssen, können in eigenen Dateien implementiert werden.
- Großteils MDE-basiert
    - Alle Code-intensiven Komponenten werden zum Schutz des Codes kompiliert als MDE-Dateien ausgeliefert.
- MDB-Module möglich
    - Es können auch Komponenten im MDB-Format eingebunden werden. Diese bietet die Möglichkeit, beispielsweise dem fortgeschrittenen Endbenutzer den Berichtseditor von Access zur Anpassung seiner Berichte zur Verfügung zu stellen.

Alle Komponenten folgen in diesem Szenario einem einheitlichen Aufbau. Siehe dazu den nächsten Abschnitt über den Aufbau einer Komponente.

b. Grundlegender Gedanke

Damit es möglich ist

- den Hauptteil der Anwendung („MyApplication") als MDE/ACCDE auszuliefern,

- die Berichte dennoch in einer MDB/ACCDB bleiben können und

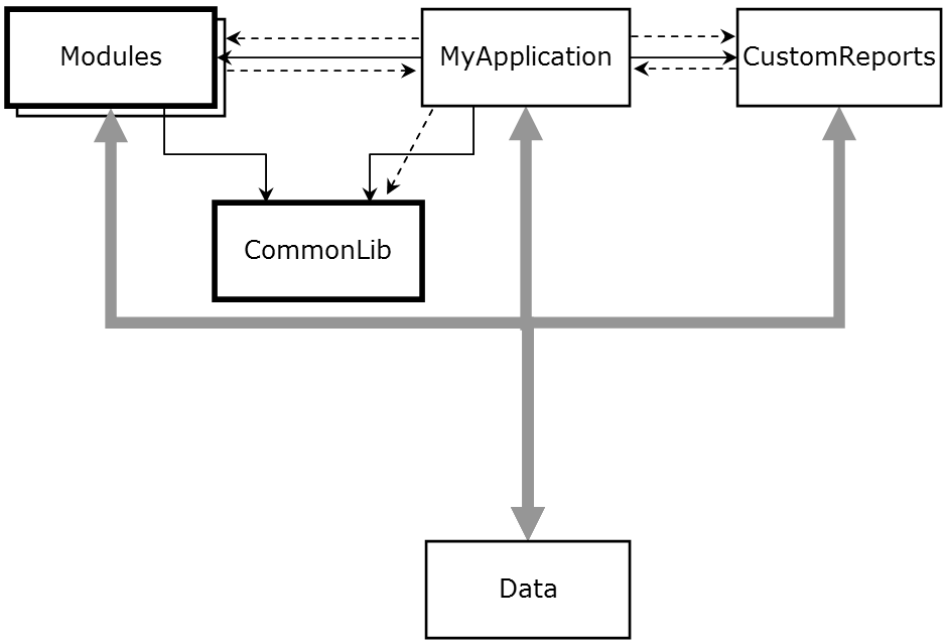
- dennoch die volle Integration der benutzeradaptierbaren Berichte („CustomReports") in die Hauptanwendung zu erhalten,

müssen alle Verweise in einer zentralen MDB/ACCDB („Starter" – siehe unten) auf alle Komponenten der Anwendung gesetzt werden. Über diese Verweise kann nun der Starter von jeder Komponente eine Instanz einer Fassadenklasse erzeugen. Diese Instanzen kann der Starter an die Instanz der Fassadenklasse der Hauptkomponente „MyApplication" übergeben werden. Damit hat „MyApplication" jeweils eine Objektreferenz auf alle externen Komponenten der Anwendung und kann so die weiteren Codes dieser Komponenten ausführen lassen.

Ruft „MyApplication" nun z.B. in „CustomReports" Code auf, der dort einen Bericht öffnet (`DoCmd.OpenReport()`), wird der Bericht wie gewohnt im Access-Hauptfenster geöffnet und bietet dort auch die Möglichkeit, in die Entwurfsansicht zu wechseln.

c. Struktur bei der Entwicklung

Während der Entwicklung ist der oben beschriebene Mechanismus mit dem Starter nicht erforderlich („MyApplication" liegt auch als MDB/ACCDB vor). Die Struktur sieht daher wie folgt aus:



(Legende siehe Abschnitt „Legende Zu den Architekturdiagrammen" auf Seite 53)

Bestandteile

MyApplication

MyApplication stellt die eigentliche Anwendung dar. Im Falle einer Modularisierung der Anwendung befindet sich in MyApplication die Hauptanwendung mit den standardmäßig freigeschalteten Modulen, während die optionalen Module in eigene Komponenten ausgelagert sind (siehe Komponente "Module").

Modules

Dieser Block stellt in obiger Grafik stellvertretend alle Komponenten dar, die optional verfügbare Module implementieren. Die Hauptanwendung befindet sich in MyApplication während zusätzliche Module in eigene Komponenten ausgelagert sind.

CustomReports

Diese MDB enthält alle vom Endbenutzer zu verändernden oder neu zu erstellenden Berichte.

Per Arbeitsgruppensicherheit kann ggflls. sichergestellt werden, dass auf die Daten in Data nur lesend zugegriffen werden kann. Mit dem Access 2007-Dateiformat besteht diese Möglichkeit jedoch nicht mehr.

CommonLib

In der Komponente CommonLib befindet sich Code, der in vielen Anwendung Verwendung finden kann (allgemeine Tool-Klassen).

Data

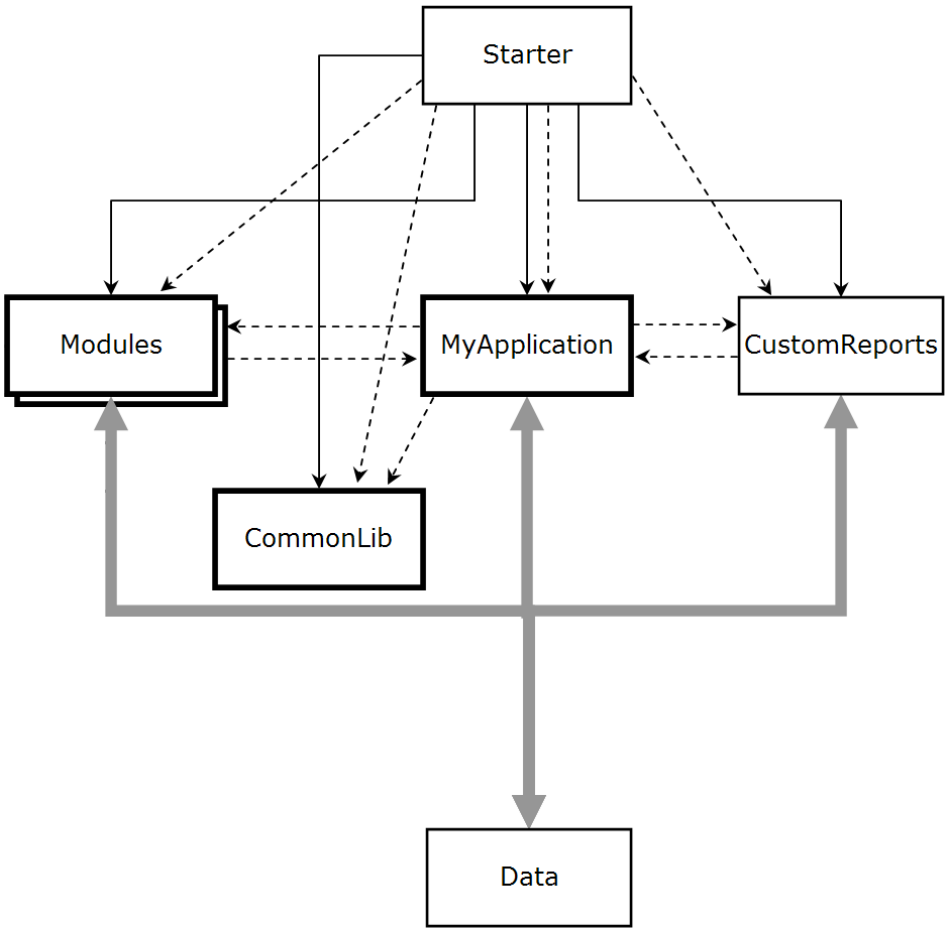
Das ist die Datendatei ("Backend") der Anwendung. Alle Komponenten, die Zugriff auf die Daten benötigen, benötigen Zugriff auf die Datenbank. Dies kann entweder über verknüpfte Tabellen geschehen, oder per Code über eine Database-Instanz.

Funktionsweise zur Entwicklungszeit

Da während der Entwicklung auch die Hauptanwendung („MyApplication") als MDB/ACCDB vorliegt, können alle anderen Komponenten der Anwendung inklusive der CustomReports per Bibliotheksverweis eingebunden werden. Der Zugriff auf den Code der Komponenten kann damit wie gewohnt erfolgen.

d. Struktur in Produktion

Im Auslieferungszustand, wenn also „MyApplication" bereits als MDE/ACCDE vorliegt, funktioniert das System nur mehr mithilfe des oben angesprochenen Starters:



(Legende siehe Abschnitt „Legende Zu den Architekturdiagrammen" auf Seite 53)

Zusätzlicher Bestandteil

Starter

Diese Komponente im MDB/ACCDB-Format übernimmt grundlegende logistische Aufgaben:

- Auffinden der anderen Komponenten der Anwendung (hier: CommonLib, Modules, MyApplication, CustomReports)

  Das Auffinden kann über den Access-eigenen Algorithmus (Suche in folgenden Verzeichnissen: (a) System (b) Access-Installation (c) Datenbank) erfolgen. Da

der Speicherort aller Komponenten absolut frei wählbar sein soll, wird der Access-interne Algorithmus nicht immer funktionieren. Aus diesem Grund können die Pfade zu den anderen Komponenten aus einer Konfiguration (z.B. Registry) gelesen werden oder vom Benutzer mittels Dateiauswahl-Dialog geöffnet werden.

Es ist also wichtig, dass der Starter durch geeignete Verfahren die absoluten Pfade aller Komponenten kennt.

- Bereitstellen von Objektverweisen für alle davon abhängigen Module.

  Beispielsweise benötigt MyApplication eine Instanz einer Fassadenklasse von CommonLib. Starter erzeugt diese Instanz und reicht eine Referenz darauf an MyApplication weiter.
- Starten der eigentlichen Anwendung (hier: MyApplication) durch Aufruf einer Methode der Instanz der Fassadenklasse von MyApplication.

e. Vorsicht: In Produktion ist der Starter der Host für alle anderen Komponenten!

Im Produktionsszenario startet der Benutzer die Starter-Anwendung, die dann über das oben beschriebene Verfahren alle anderen Komponenten mit einbindet. Die Hauptkomponente „MyApplication" hat in diesem Zusammenhang keine gesonderte Bedeutung.

Auswirkung 1: CodeDb() statt CurrentDb()

Wenn Code aus einer der eingebundenen Komponenten läuft, wird er im Kontext der Starter-Anwendung ausgeführt. Die Angabe `CurrentDb()` im Code einer Komponente bezieht sich also auf die Starter-Anwendung. Da die Starter-Anwendung mit der eigentlichen Anwendungs-Funktionalität nichts zu tun hat, und daher im Speziellen auch keine Tabellen des Backends eingebunden hat, können über `CurrentDb()` keine Datenzugriffe erfolgen. Im Code der Komponenten muss daher immer CodeDb() verwendet werden, um die Daten aus einer in der jeweiligen Komponente eingebundenen Tabelle bearbeiten zu können.

DoCmd.TransferSpreadsheet() und ähnliche Funktionen

Die Funktionen des `DoCmd`-Objekts, die auf Daten zugreifen funktionieren in diesem Szenario nicht. Diese Funktionen dürften intern `CurrentDb()` verwenden und damit in der Starter-Anwendung nach den Daten suchen, was natürlich fehlschlägt.

Für diese Funktionalitäten müssen also Alternativen gesucht werden. Im Fall von `DoCmd.TransferSpreadsheet()` kann mit `Excel.Range.CopyFromRecordset()` recht komfortabel ein Workaround implementiert werden.

Auswirkung 2: Symbolleisten

Symbolleisten müssen daher in der Starteranwendung vorhanden sein. Symbolleisten aus MyApplication sind im Produktionsszenario nicht verfügbar.

Änderungen an Symbolleisten

Da während der Entwicklung aber die Anwendung immer direkt über MyApplication gestartet wird, ist es erforderlich, dass alle Symbolleisten sowohl in „MyApplication" wie in „Starter" vorhanden sind. Es darf daher nicht vergessen werden, Änderungen an den Symbolleisten nicht nur während der Entwicklung in „MyApplication" durchzuführen, sondern sie auch in „Starter" nachzuziehen. Die Starter-Komponente muss dann natürlich auch mit deployt werden.

Aufruf von Code aus Symbolleisten-Elementen

Da das Menü aus der Starter-Datei verwendet wird, muss der Code-Verweis des Symbolleisten-Elements so gestaltet werden, dass dennoch Code aus MyApplication (i.A.) aufgerufen wird.

Das kann ganz einfach dadurch sichergestellt werden, indem den Methodenaufrufen in der Eigenschaft „Bei Aktion" der Bibliotheksname von MyApplication vorangestellt wird. Statt `ShowKundenDetails` muss dann also bspw. `MyApplication.ShowKundenDetails` im entsprechenden Symbolleistenelement eingetragen werden. Diese Syntax funktioniert sowohl zur Entwicklungszeit in MyApplication als auch in Produktion in der Starter-Datei. Damit können Menüeinträge leicht von MyApplication in die Starter-Datei kopiert werden.

3. Aufbau einer Komponente

a. Allgemeines

Der im Folgenden beschriebene Aufbau ist die Basis für die im vorangegangenen Kaptiel vorgestellte Struktur einer komponentenbasierten Anwendung in Microsoft Access und bedient folgende Ideen:

- Jede Nutzfunktionalität in einer Klasse

  Keines der Code-Elemente (Methoden und Eigenschaften) der Komponente befindet sich in einem Standardmodul. Dadurch wird der Zugriff auf jede Nutzfunktionalität per Objektreferenz ermöglicht. Das ist die Voraussetzung für die Verwendung der Komponente aus einer anderen Komponente, die keinen Bibliotheks-Verweis zur referenzierten Komponente aufbauen kann.
- Verbergen aller Interna

  Die Interna der Komponente sollen von außerhalb der Komponente nicht zugreifbar sein. Das betrifft vor allem intern verwendete Konstante, Variablen und Objekte, die komponentenweit sichtbar sein sollen. Das ist eine der Aufgaben der Klasse CMain (siehe unten).
- Bereitstellen von Factory-Funktionen

  Da Klassen aus referenzierten Komponenten im MDB- oder MDE-Format nicht direkt per New instanziiert werden können, müssen Instanzen innerhalb der Komponente erstellt werden. Das ist die zweite Aufgabe der Klasse CMain (siehe unten).
- Identifikation der Komponente

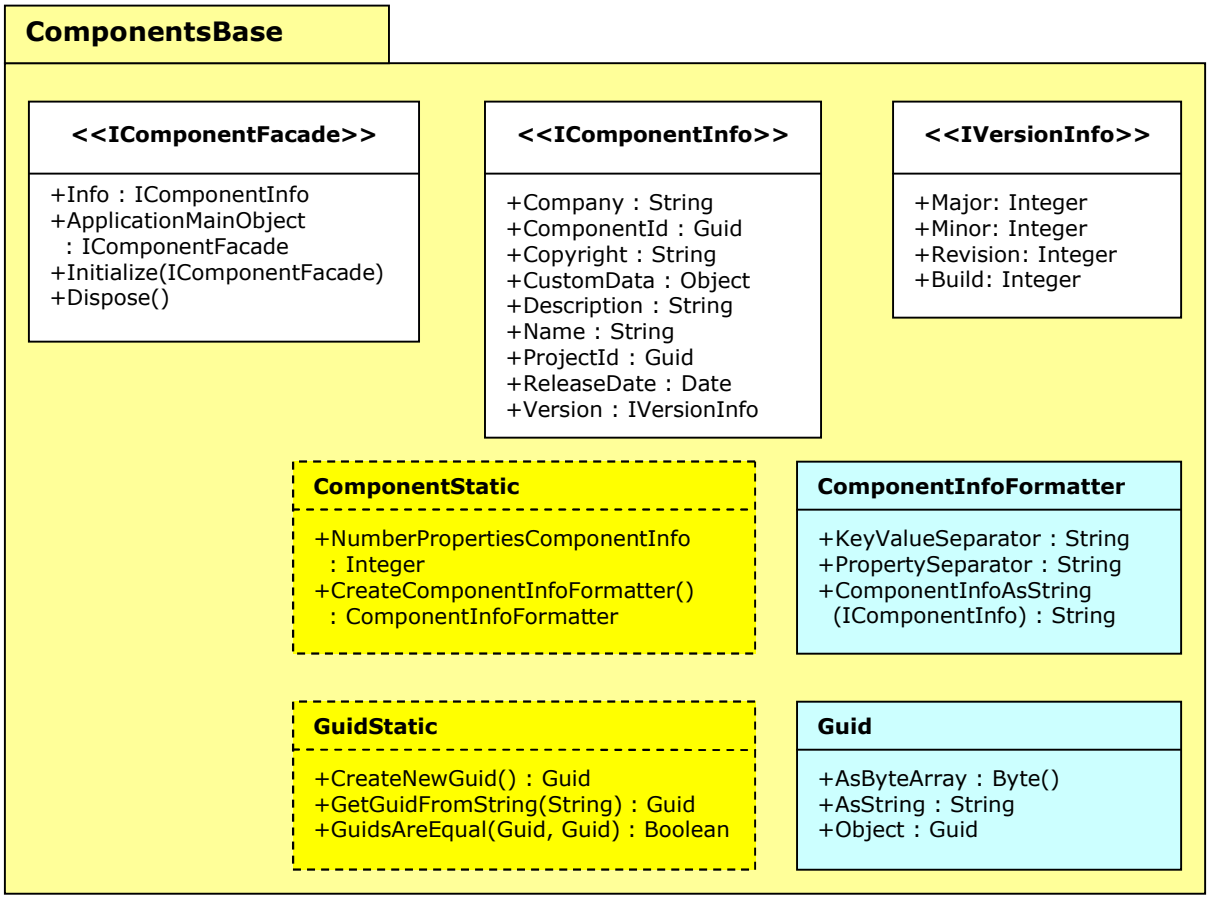
  Jede Komponente soll grundlegende Informationen über sich preisgeben. Dazu dient die Klasse CComponentInfo (siehe unten).

b. Komponenten-Infrastruktur in ComponentsBase (MDE/ACCDE)

Um die Handhabung der Komponenten nach dem vorliegenden Modell zu vereinfachen, ist etwas Infrastruktur von Nöten. Die Basis für diese Infrastruktur befindet sich in der Komponente ComponentsBase.

Beschreibung ComponentsBase

Die Komponente ComponentsBase bildet die Basis für alle konkreten Komponenten. Im Wesentlichen deklariert sie drei Interfaces (für die Fassadenklasse, eine Infoklasse und Versionsklasse) sowie zwei Hilfsklassen (Guid und ComponentInfoFormatter):



Interface IComponentFacade

Das Interface IComponentFacade definiert die Elemente, die die Fassadenklasse einer Komponente haben muss:

- Eigenschaft Info As IComponentInfo
  Liefert eine Referenz auf eine IComponentInfo-Implementierung. Über sie können allgemeine Informationen über die Komponente abgefragt werden.
- Eigenschaft ApplicationMainObject As Object
  Liefert die in der Methode Initialize übergebene Referenz auf das Fassadenobjekt der Hauptanwendung.
- Methode Initialize(IComponentFacade)
  Initialisiert die Komponente. Bekommt Referenz auf das Fassadenobjekt der Hauptanwendung als Parameter mitgeliefert.
- Methode Dispose()
  Weißt die Komponente an ihre Ressourcen freizugeben um eine ordnungsgemäße Beendigung der gesamten Applikation zu ermöglichen. Im Speziellen muss in Dispose() die Referenz auf die Hauptfassade gelöscht werden (Zirkelreferenz).

Neben dieser erforderlichen Elemente der Komponentenfassade (die Klasse, die IComponentFacade implementiert) bietet die Fassade auch Zugriff auf die eigentliche Funktionalität der Komponente.

Interface IComponentInfo

Das Interface IComponentInfo definiert die Elemente, die eine Infoklasse einer Komponente bereitstellen muss:

- Eigenschaft Company As String
  Name der Herstellerfirma der Komponente
- Eigenschaft ComponentId As Guid
  Id der Komponente (um Unabhängig vom Namen zu sein)
- Eigenschaft Copyright As String
  Copyright-Information zur Komponente
- Eigenschaft CustomData As Object
  Bietet die Möglichkeit zusätzliche komponentenspezifische Informationen bereitzustellen
- Eigenschaft Description As String
  Beschreibung der Komponente
- Eigenschaft Name As String
  Name der Komponente
- Eigenschaft ProjectId As Guid
  Id des Projekts, zu dem diese Komponente gehört
- Eigenschaft ReleaseDate As Date
  Datum der Freigabe dieser Komponente
- Eigenschaft Version As IVersionInfo
  Versionsinformationen zur Komponente

Interface IVersionInfo

Das Interface IVersionInfo definiert die Informationen für eine vollständige Versionsangabe der Komponente:

- Eigenschaft Major As Integer
  Hauptversionsnummer
- Eigenschaft Minor As Integer
  Nebenversionsnummer
- Eigenschaft Revision As Integer
  Revisionsnummer
- Eigenschaft Build As Integer
  Nummer des Builds

Klasse Guid

Die Klasse Guid repräsentiert eine Guid (Globally Unique Identifier) zur weltweit eindeutigen Kennzeichnung von Projekten und Componenten (IComponentInfo.ProjectId und IComponentInfo.ComponentId)

Klasse ComponentInfoFormatter

Der ComponentInfoFormatter bietet die Möglichkeit die Daten aus IComponentInfo als String zu erhalten. Der Trenntext zwischen Name und Wert einer Eigenschaft und der Trenntext zwischen den Eigenschaften lässt sich konfigurieren. Damit kann die Versionsinformation relativ leicht als einfache ASCII-Tabelle, HTML- oder XML-Struktur oder CSV-Liste ausgegeben werden.

Standardmodul ComponentStatic

Dieses Standardmodul enthält eine Factory-Funktion für den ComponentInfoFormatter und eine Eigenschaft für die Anzahl der Eigenschaften in IComponentInfo.

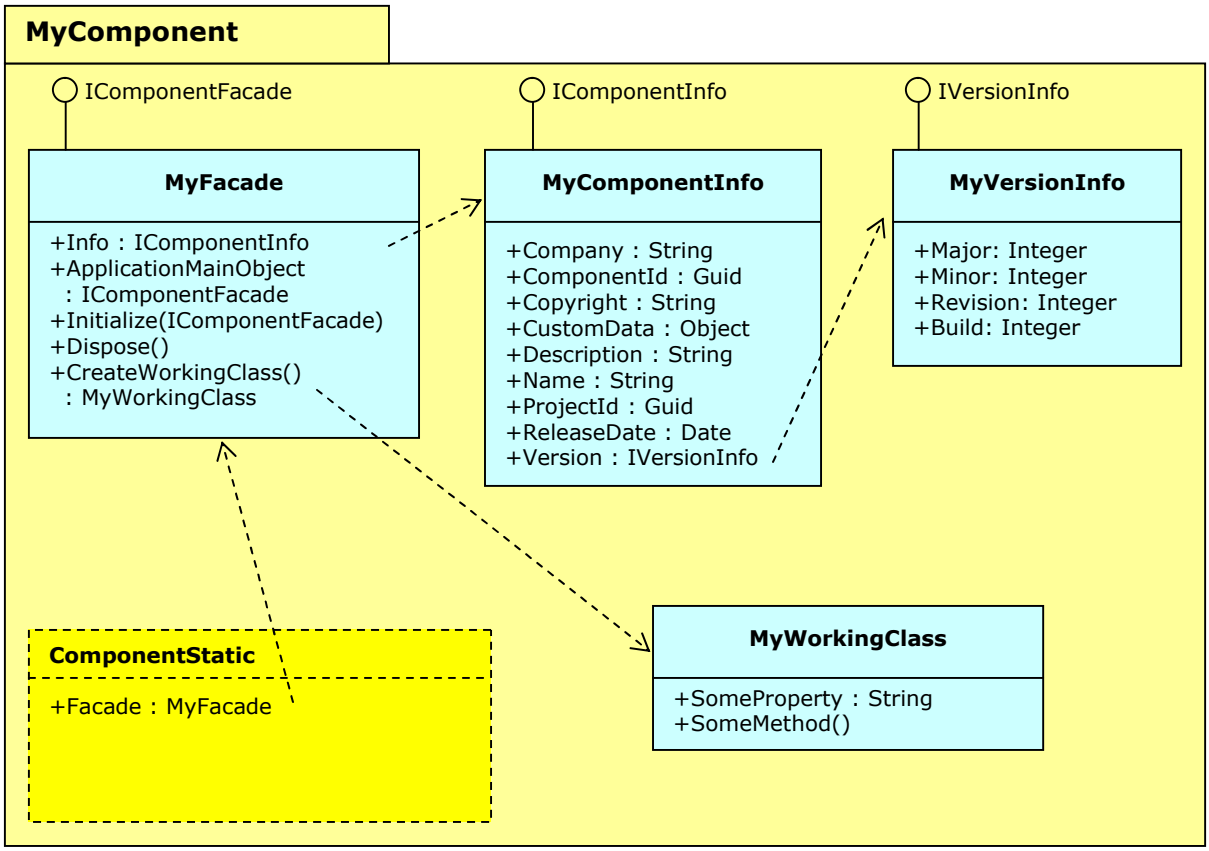
Standardmodul GuidStatic

Dieses Standardmodul beinhaltet die Factory-Funktion für die Klasse Guid sowie zwei Hilfsfunktionen für die Arbeit einer Guid.

c. Entwickeln einer Komponente

Zur Erstellung einer Komponente für die hier vorgestellte Architektur müssen folgende Schritte durchgeführt werden:

1. Einen Verweis auf die ComponentsBase setzen
2. Die drei Interfaces der ComponentsBase (IComponentFacade, IComponentInfo und IVersionInfo) implementieren. Ev. kann die Implementierung von IComponentInfo und IVersionInfo von ein und der selben Klasse übernommen werden.
3. In einem Standardmodul (Name ev. ComponentStatic) eine öffentliche Eigenschaft namens Facade anlegen, die eine Referenz auf ein Singleton-Instanz der Klasse zurückliefert, die IComponentFacade implementiert.



d. Verwenden der Komponenten in der Anwendung

Der Starter muss in Produktion die Referenzen auf die Komponentenfassaden an die Fassade der Hauptanwendung übergeben. Was die Hauptanwendung mit diesen Referenzen tut, bleibt ihr überlassen. Im Allgemeinen wird sie diese Referenzen validieren und danach in einem Feld speichern. Die konkrete Funktionalität kann zur Produktionslaufzeit dann nur mittels Late-Binding erreicht werden.

Validierung der übergebenen Komponente

Anhand von IComponentFacade.Info kann die prinzipielle Tauglichkeit der übergebenen Komponenten überprüft werden:

- Die ProjectId muss bei anwendungsspezifischen Komponenten gleich der eigenen ProjectId sein.

- Die ComponentId muss eventuell überprüft werden – die Hauptanwendung kennt ihre Module anhand deren ComponentId.
- Eventuell muss die Komponente einen gewissen Versionsstand aufweisen.

Beispiel:

```
Public Sub TestComponent(ByVal ComponentFacade As IComponentFacade)
   Dim ComponentFacade As IComponentFacade
   Dim Info As IComponentInfo
   Dim Version As IVersionInfo
   Dim objCustomData As Object
   Dim customer As Object

   Set Info = ComponentFacade.Info

   ' Versionsinformation anzeigen
   Set Version = Info.Version
   Debug.Print Version.AsString
   Set Version = Nothing

   ' Komponenteninfo anzeigen
   With ComponentBase.ComponentStatic.CreateComponentInfoFormatter()
'      .KeyValueSeparator = ";"
'      .PropertySeparator = ";"
      Debug.Print .ComponentInfoAsString(ComponentFacade.Info)
   End With

   ' Benutzerdefinierte Komponenteninfodaten anzeigen
   Set objCustomData = Info.CustomData
   Debug.Print objCustomData.History    ' Implizites Wissen über die Komponente
   Set objCustomData = Nothing

   ' Nutzfunktionalitäten der Komponente aufrufen (implizites Wissen)
   ComponentFacade.OpenReport "Customer", acViewPreview
   For Each customer In ComponentFacade.GetVipCustomers()
      Debug.Print customer
   Next customer

   Set Info = Nothing
End Sub
```

In diesem Beispiel wird angenommen, dass die Methode über den Starter mit einer Referenz auf die Fassadenklasse der Komponente bekommen hat.

Weiters existiert implizites Wissen darüber, dass in IComponentInfo.CustomData ein Objekt zurückgeliefert wird, dass eine Eigenschaft History besitzt, die vom Typ String ist, oder automatisch nach String konvertiert werden kann.

D. Demos aus dem Vortrag

1. Demos aus Teil I

a. Demoblock 1: Tabellen wiedereinbinden – erster Code und Refactorings

Ziel: Code zum Wiedereinbinden der Tabellen aus einem Backend, dessen Pfad als String zur Verfügung steht (kommt aus Konfiguration, Benutzerfrage, oä)

Schritt 1: Pseudocode:

```
' Pseudocode
Für alle Tabellen:
      Wenn eingebundene Tabelle aus Jet-Backend ist:
            Verknüpfungseigenschaften mit neuem Pfad aktualisieren
Aktualisieren des Tabellencontainers
```

Schritt 2: Schreiben des Codes in einem Modul:

```
' Variante 1
Sub ReconnectJetBackend(BackendPath As String)
   Dim tdf As DAO.TableDef

   For Each tdf In CurrentDb.TableDefs
      If Left(tdf.Connect, 10) = ";DATABASE=" Then
         tdf.Connect = ";DATABASE=" & BackendPath
         tdf.RefreshLink
      End If
   Next tdf
   CurrentDb.TableDefs.Refresh
End Sub
```

Kritikpunkte an dieser Vorgangsweise:

- Sichtbarkeit der Methode nicht klar (`Public`?)
- Parameterübergabe (`ByRef`/`ByVal`?)
- Zweimalige Verwendung von `CurrentDb`
- `CurrentDb` ist Methode, nicht Property → `CurrentDb()`
- Literal `";DATABASE="` kommt zweimal vor
- Literal `10` lässt sich mittels `Len()` aus `";DATABASE="` ableiten

Schritt 3: Verbesserter Code:

```
' Variante 2
Public Sub ReconnectJetBackend(ByVal BackendPath As String)
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database
   Const connectPrefixJet As String = ";DATABASE="

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If Left(tdf.Connect, Len(connectPrefixJet)) = connectPrefixJet Then
         tdf.Connect = connectPrefix & BackendPath
         tdf.RefreshLink
      End If
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub
```

Kritikpunkte an dieser Variante:

- Zu „technisch"
- Die eigentliche Funktionalität ist nicht gut erkennbar

Schritt 4: Fortgesetztes Refactoring:

```
' Variante 3
Private Const connectPrefixJet As String = ";DATABASE="

Public Sub ReconnectJetBackend(ByVal BackendPath As String)
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         UpdateJetConnect tdf, BackendPath
      End If
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub

Private Function IsJetConnect(ByVal Connect As String) As Boolean
   IsJetConnect = (Left(Connect, Len(connectPrefixJet)) = connectPrefixJet)
End Function

Private Sub UpdateJetConnect(ByVal tdf As DAO.TableDef, _
                             ByVal BackendPath As String)
   tdf.Connect = connectPrefixJet & BackendPath
   tdf.RefreshLink
End Sub
```

Vorteil dieser Variante:

- Kleine und leicht überschaubare Einheiten
- Hauptmethode ist wieder
    - sehr nahe am Pseudo-Code
    - recht nahe an der natürlichen Sprache

b. Demoblock 2: Erweiterung der Anforderungen – Benutzerfeedback

Ziel: Feedback für den Benutzer über den Fortschritt der Einbindung

Schritt 1: Formular zur Anzeige der bearbeiteten Tabelle in einem Textfeld erstellen
- Soll automatisch geöffnet und geschlossen werden

Schritt 2: Direkte Übergabe des Tabellennamens an das Textfeld:

```
' Variante 4
Public Sub ReconnectJetBackend(ByVal BackendPath As String)
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         Forms!frmReconnectionProgress!txtTabelle.Value = tdf.Name
         DoEvents
         UpdateJetConnect tdf, BackendPath
      End If
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub
```

Schritt 3: Aufruf (z.B. in Startup-Formular):

```
' Variante 1
Public Sub Reconnect()
   DoCmd.OpenForm "frmReconnectionProgress"
   DoEvents

   Tools.ReconnectJetBackend "…"

   DoCmd.Close
End Sub
```

Kritikpunkte an dieser Vorgangsweise:
- Implizite Voraussetzungen für das Funktionieren von `ReconnectJetBackend()`:
  - Das Formular muss „frmReconnectionProgress" heißen
  - Das Formular muss geöffnet sein
  - Das Steuerelement muss „txtTabelle" heißen
  - Das Steuerelement muss ein Textfeld sein (Eigenschaft Value – funktioniert nicht bei Umstellung auf Bezeichnungsfeld, dort müsste die Eigenschaft Caption verwendet werden)
- Alle oben angeführten Punkte können im vorliegenden Szenario nicht vom Compiler überprüft werden.
  - → Fehler treten daher erst zur Laufzeit auf!
- Diese Vorbedingungen müssen an verschiedenen Stellen im Code erfüllt werden.
- → Schlechte Wartbarkeit
- Das `DoCmd.Close()` schließt eventuell das falsche Access-Objekt.

- Schritt 4: Verbesserter Code:

```
‘ Variante 5
Public Sub ReconnectJetBackend(ByVal BackendPath As String, _
                               ByVal TableNameTextBox As Access.TextBox)
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         TableNameTextBox.Value = tdf.Name
         DoEvents
         UpdateJetConnect tdf, BackendPath
      End If
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub
```

- Schritt 5: Geänderter Aufruf:

```
' Variante 2
Public Sub Reconnect()
   Const FormName As String = "frmReconnectionProgress"

   DoCmd.OpenForm FormName
   DoEvents

   Tools.ReconnectJetBackend "…", Forms(FormName)!txtTabelle

   DoCmd.Close acForm, FormName
End Sub
```

- Vorteile dieser Vorgangsweise:
  - Eigentlicher Code in `ReconnectJetBackend()` macht keinerlei Annahmen über die Namen von Formular oder Steuerelement
  - Die eigentliche Funktionalität und die Darstellung sind wesentlich klarer getrennt.
  - Bessere Wartbarkeit
    - § Änderung des Formularnamens
    - § Änderung des Steuerelements

- Was immer noch stört:
  - Potenzielle Fehlerquellen (vom Compiler nicht überprüfbar):
    - § Name des Formulars
    - § Name und Typ des Textfeldes

- Schritt 6: Abermals geänderter Aufruf:

```
' Variante 3
Private Sub Reconnect()
   Const FormName As String = "frmReconnectionProgress"
   Dim ReconnectInfoForm As Form_frmReconnectionProgress

   DoCmd.OpenForm FormName
   Set ReconnectInfoForm = Forms(FormName)
   DoEvents

   Tools.ReconnectJetBackend "…", ReconnectInfoForm.txtTabelle

   DoCmd.Close acForm, ReconnectInfoForm.Name
   Set ReconnectInfoForm = Nothing
End Sub
```

Hinweis:
- Das CodeBehind-Modul `Form_frmReconnectionProgress` existiert nur, wenn die Eigenschaft „Enthält Modul" auf „Ja" steht.
- Es muss also ein ev. leeres Code-Modul für das Formular existieren.

Verbleibende Kritik an dieser Variante:
- Vom Compiler nicht überprüfbar:
  - Name des Formulars
- Was vom VB6-Compiler leider nicht überprüft wird (damit müssen wir leben):
  - Typ des Steuerelements
  - Der Compiler könnte es aber überprüfen!
- Nach wie vor Vermischung von Funktionalität und Darstellung
  - Alternative Verwendung des Tabellennamens erfordert immer Anpassung von `ReconnectJetBackend()`:
    Zum Beispiel ...
    - § zur Anzeige in Bezeichnungsfeld (`Caption` statt `Value`)
    - § zur Auflistung in Listfeld (Manipulation der RowSource oder temporäre Tabelle)
    - § zur Darstellung in einem Endlosformular (Schreiben in temporäre Tabelle)
    - § zur Darstellung eines Fortschrittsbalkens
    - § zum Protokollieren in Log-Datei
    - § ... und das obwohl die eigentliche Funktionalität unverändert bleibt!

c. Demoblock 3: Trennung von Funktionalität und GUI

Anforderung: Vollständige Trennung von eigentlicher Funktionalität und User Interface

Ansatz 1: Ereignisse

Idee/Wunsch:
- `ReconnectJetBackend()` signalisiert nach außen das Einbinden einer neuen Tabelle
- Andere(r) Programmteil(e) reagieren darauf

Technisches Vehikel dazu:
- Ereignissen
- Vorbedingung
  - Code liegt in einer Klasse
  - Standardmodul kann keine Ereignisse auslösen

Schritt 1: Auslagern des Codes in eine Klasse:

- Neue Klasse JetReconnector
- (funktional identisch zur vorangegangenen Implementierung)

```
' This is JetReconnector
' Project: TabellenWiedereinbinden
' Last change: 16.09.2007
'              by Paul Rohorzka (paul.rohorzka@softconcept.at)
' ShortDescr:  Wiedereinbindung aller verknüpften Jet-Tabellen
' Description: Verknüpft alle an ein Jet-Backend verknüpften Tabellen
'              mit einem neuen Backend. Liefert Fortschrittsinformation
'              an eine TextBox
Option Compare Database
Option Explicit

' ___ Constants ___

Private Const connectPrefixJet As String = ";DATABASE="

' ___ Fields for Property Values ____

Private mp_strBackendPath As String
Private mp_txtProgressInfo As Access.TextBox

' ___ Public Properties ___

Public Property Let BackendPath(ByVal NewBackendPath As String)
   mp_strBackendPath = NewBackendPath
End Property
Public Property Get BackendPath() As String
   BackendPath = mp_strBackendPath
End Property

Public Property Set ProgressInfoTextBox(ByVal NewProgressInfoTextBox _
                                        As Access.TextBox)
   Set mp_txtProgressInfo = NewProgressInfoTextBox
End Property
Public Property Get ProgressInfoTextBox() As Access.TextBox
   Set ProgressInfoTextBox = mp_txtProgressInfo
End Property

' ___ Public Methods ___

Public Sub Reconnect()
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         mp_txtProgressInfo.Value = tdf.Name
         DoEvents
         UpdateJetConnect tdf, BackendPath
      End If
```

```
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub

' ___ Private Methods ___

Private Function IsJetConnect(ByVal Connect As String) As Boolean
   IsJetConnect = (Left(Connect, Len(connectPrefixJet)) = connectPrefixJet)
End Function

Private Sub UpdateJetConnect(ByVal tdf As DAO.TableDef, _
                             ByVal BackendPath As String)
   tdf.Connect = connectPrefixJet & BackendPath
   tdf.RefreshLink
End Sub
```

Aufruf:

```
' Variante 4
Private Sub Reconnect()
   Const FormName As String = "frmReconnectionProgress"
   Dim ReconnectInfoForm As Form_frmReconnectionProgress
   Dim Reconnector As JetReconnector

   DoCmd.OpenForm FormName
   Set ReconnectInfoForm = Forms(FormName)
   DoEvents

   Set Reconnector = New JetReconnector

   With Reconnector
      .BackendPath = "…"
      Set .ProgressInfoTextBox = ReconnectInfoForm.txtTabelle

      .Reconnect
   End With

   Set Reconnector = Nothing

   DoCmd.Close acForm, ReconnectInfoForm.Name
   Set ReconnectInfoForm = Nothing
End Sub
```

Schritt 2: Umstellung auf Ereignisse

- JetReconnector

```
' This is JetReconnector
' Project: TabellenWiedereinbinden
' Last change: 16.09.2007
'              by Paul Rohorzka (paul.rohorzka@softconcept.at)
' ShortDescr:  Wiedereinbindung aller verknüpften Jet-Tabellen
'              mit einem neuen Backend. Löst zur Fortschrittsinformation
'              das Ereignis ReconnectingTable aus.
Option Compare Database
Option Explicit

' ___ Events ___

Public Event ReconnectingTable(ByVal Table As DAO.TableDef)

' ___ Constants ___

Private Const connectPrefixJet As String = ";DATABASE="

' ___ Fields for Property Values ____

Private mp_strBackendPath As String

' ___ Public Properties ___

Public Property Let BackendPath(ByVal NewBackendPath As String)
   mp_strBackendPath = NewBackendPath
End Property
Public Property Get BackendPath() As String
   BackendPath = mp_strBackendPath
End Property

' Keine Eigenschaft ProgressInfoTextBox mehr!

' ___ Public Methods ___

Public Sub Reconnect()
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database

   Set dbs = CurrentDb()

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         RaiseEvent ReconnectingTable(tdf)
         UpdateJetConnect tdf, BackendPath
      End If
   Next tdf
   dbs.TableDefs.Refresh

   Set dbs = Nothing
End Sub
```

```
' ___ Private Methods ___

Private Function IsJetConnect(ByVal Connect As String) As Boolean
   IsJetConnect = (Left(Connect, Len(connectPrefixJet)) = connectPrefixJet)
End Function

Private Sub UpdateJetConnect(ByVal tdf As DAO.TableDef, _
                             ByVal BackendPath As String)
   tdf.Connect = connectPrefixJet & BackendPath
   tdf.RefreshLink
End Sub
```

Verwendung:

- Zum Abfangen von Ereignissen muss die Objektvariable Reconnector
  - auf Modulebene
  - mit dem Schlüsselwort WithEvents

  deklariert werden.
- Z.B. in Start-Formular:

```
Private WithEvents Reconnector As JetReconnector
Private ReconnectInfoForm As Form_frmReconnectionProgress

' Variante 5
Private Sub Reconnect()
   Const FormName As String = "frmReconnectionProgress"

   DoCmd.OpenForm FormName
   Set ReconnectInfoForm = Forms(FormName)
   DoEvents

   Set Reconnector = New JetReconnector

   With Reconnector
      .BackendPath = "…"

      .Reconnect
   End With

   Set Reconnector = Nothing

   DoCmd.Close acForm, ReconnectInfoForm.Name
   Set ReconnectInfoForm = Nothing
End Sub

Private Sub Reconnector_ReconnectingTable(ByVal Table As DAO.TableDef)
   ReconnectInfoForm.txtTabelle.Value = Table.Name
End Sub
```

Wesentlicher Vorteil dieser Variante:

- Vollkommene Trennung zwischen Funktionalität und Darstellung im User Interface
  - Klasse `JetReconnector`
    - § kann Tabellen mit Jet-Datenbanken verbinden
    - § Löst bei jeder Tabelle ein Ereignis aus
  - Aufrufende Klasse
    - § stößt den Vorgang an
    - § reagiert auf das Ereignis
- JetReconnector muss überhaupt nicht angefasst werden, wenn an das User Interface andere Anforderungen gestellt werden:
  - Textfeld
  - Bezeichnungsfeld
  - Listfeld
  - Endlosformular
  - Log-Datei
  - Protokoll-Tabelle
  - gar keine Reaktion
  - ...

  (Alles ohne die kleinste Änderung in `JetReconnector` machbar!)

Erweiterung des Prinzips:

- Reicheres Ereignismodell durch zusätzliche Parameter und Ereignisse:
  - `BeginReconnect`
  - `ReconnectingTable` mit zusätzlichem Parameter `Skip`
  - `ReconnectedTable`
  - `ReconnectingTableSkipped`
  - `EndReconnect`

```
' This is JetReconnector

...

' ___ Events ___

Public Event BeginReconnect(ByVal Database As DAO.Database)
Public Event ReconnectingTable(ByVal Table As DAO.TableDef, _
                               ByRef Skip As Boolean)
Public Event ReconnectedTable(ByVal Table As DAO.TableDef)
Public Event ReconnectingTableSkipped(ByVal Table As DAO.TableDef)
Public Event EndReconnect(ByVal Database As DAO.Database)

...
```

```
' ___ Public Methods ___

Public Sub Reconnect()
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database
   Dim blnSkip As Boolean

   Set dbs = CurrentDb()

   RaiseEvent BeginReconnect(dbs)

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         blnSkip = False
         RaiseEvent ReconnectingTable(tdf, blnSkip)
         If blnSkip Then
            RaiseEvent ReconnectingTableSkipped(tdf)
         Else
            UpdateJetConnect tdf, BackendPath
         End If
         RaiseEvent ReconnectedTable(tdf)
      End If
   Next tdf
   dbs.TableDefs.Refresh

   RaiseEvent EndReconnect(dbs)

   Set dbs = Nothing
End Sub
```

Verwendung:

```
' Variante 6
Private Sub Reconnect()
   Set Reconnector = New JetReconnector

' Kein Code zum Öffnen des Fortschrittsformulars

   With Reconnector
      .BackendPath = "…"

      .Reconnect
   End With

' Kein Code zum Schließen des Fortschrittsformulars

   Set Reconnector = Nothing
End Sub
```

```
Private Sub Reconnector_BeginReconnect(ByVal Database As DAO.Database)
   Const FormName As String = "frmReconnectionProgress"

   DoCmd.OpenForm FormName
   Set ReconnectInfoForm = Forms(FormName)
   DoEvents
End Sub

Private Sub Reconnector_ReconnectingTable(ByVal Table As DAO.TableDef, _
                                          Skip As Boolean)
   ReconnectInfoForm.txtTabelle.Value = Table.Name & " wird verbunden ..."
End Sub

Private Sub Reconnector_ReconnectedTable(ByVal Table As DAO.TableDef)
   ReconnectInfoForm.txtTabelle.Value = Table.Name & " wurde verbunden."
End Sub

Private Sub Reconnector_ReconnectingTableSkipped(ByVal Table As DAO.TableDef)
   ReconnectInfoForm.txtTabelle.Value = Table.Name & " wurde übersprungen."
End Sub

Private Sub Reconnector_EndReconnect(ByVal Database As DAO.Database)
   DoCmd.Close acForm, ReconnectInfoForm.Name
   Set ReconnectInfoForm = Nothing
End Sub
```

Verbleibende Kritik:

- An mehreren Stellen wird auf Interna des Formulars, nämlich den Namen (und implizit auch den Typ) des Textfeldes.

Lösung:

- Bereitstellen einer öffentlichen Methode `ShowProgressInfo()` im Formular, die von der tatsächlichen Art der Darstellung der Fortschrittsinformation abstrahiert:

```
' This is Form_frmReconnectionProgress
…
' ___ Public Methods ___

Public Sub ShowProgressInfo(ByVal Message As String)
   Me.txtTabelle.Value = Message
End Sub
```

Verwendung:

```
Private Sub Reconnector_ReconnectingTable(ByVal Table As DAO.TableDef, _
                                          Skip As Boolean)
   ReconnectInfoForm.ShowProgressInfo Table.Name & " wird verbunden ..."
End Sub

Private Sub Reconnector_ReconnectedTable(ByVal Table As DAO.TableDef)
   ReconnectInfoForm.ShowProgressInfo Table.Name & " wurde verbunden."
End Sub

Private Sub Reconnector_ReconnectingTableSkipped(ByVal Table As DAO.TableDef)
   ReconnectInfoForm.ShowProgressInfo Table.Name & " wurde übersprungen."
End Sub
```

Ansatz 2: Interface

Alternative des Benutzerfeedbacks zur Variante mit Ereignissen:

- Verwendung eines Interface (z.B. IReconnectionListener), das die Methoden definiert, die von der Methode Reconnect() des JetReconnectors aufgerufen werden können.
- Eine Klasse (z.B. auch ein Formular!) implementiert das Interface
- Eine Instanz dieser Klasse (z.B. des Formulars) wird an den JetReconnector übergeben.
- Der Code in Reconnect ruft die Methoden des übergebenen Klasse (z.B. des Formulars) auf.

Interface `IReconnectionListener`:

```
' This is IReconnectionListener
' Project: TabellenWiedereinbinden
' Last change: 23.09.2007
'              by Paul Rohorzka (paul.rohorzka@softconcept.at)
' ShortDescr:  Interface eines Konsumenten von Fortschrittsinformationen
'              beim Wiedereinbinden von Tabellen
' Description: Definiert die Methoden, die ein Konsument von
Fortschrittsinformationen
'              beim Wiedereinbinden von Tabellen bereitstellen muss.
Option Compare Database
Option Explicit

' ___ Methods ___

Public Sub BeginReconnect(ByVal Database As DAO.Database)
End Sub

Public Sub ReconnectingTable(ByVal Table As DAO.TableDef, _
                             ByRef Skip As Boolean)
End Sub

Public Sub ReconnectedTable(ByVal Table As DAO.TableDef)
End Sub

Public Sub ReconnectingTableSkipped(ByVal Table As DAO.TableDef)
End Sub

Public Sub EndReconnect(ByVal Database As DAO.Database)
End Sub
```

Umstellung von `JetReconnector` auf die Verwendung des Interface `IReconnectionListener`:

```
' This is JetReconnector
' Project: TabellenWiedereinbinden

...

' Keine Events mehr deklariert


' ___ Fields for Property Values ____

Private mp_strBackendPath As String
Private mp_objListener As IReconnectionListener

' ___ Public Properties ___

Public Property Let BackendPath(ByVal NewBackendPath As String)
   mp_strBackendPath = NewBackendPath
End Property
Public Property Get BackendPath() As String
   BackendPath = mp_strBackendPath
End Property

Public Property Set Listener(ByVal NewListener As IReconnectionListener)
   Set mp_objListener = NewListener
End Property
Public Property Get Listener() As IReconnectionListener
   Set Listener = mp_objListener
End Property
```

```
' ___ Public Methods ___

Public Sub Reconnect()
   Dim tdf As DAO.TableDef
   Dim dbs As DAO.Database
   Dim blnSkip As Boolean

   If mp_objListener Is Nothing Then
      Err.Raise 5, TypeName(Me) & ".Reconnect()", _
                "Die Eigenschaft Listener muss gesetzt sein."
   End If

   Set dbs = CurrentDb()

   mp_objListener.BeginReconnect dbs

   For Each tdf In dbs.TableDefs
      If IsJetConnect(tdf.Connect) Then
         blnSkip = False
         mp_objListener.ReconnectingTable tdf, blnSkip
         If blnSkip Then
            mp_objListener.ReconnectingTableSkipped tdf
         Else
            UpdateJetConnect tdf, BackendPath
         End If
         mp_objListener.ReconnectedTable tdf
      End If
   Next tdf
   dbs.TableDefs.Refresh

   mp_objListener.EndReconnect dbs

   Set dbs = Nothing
End Sub

...
```

Implementierung von `IReconnectionListener` im Formular `frmReconnectionProgress`:

```
' This is Form_frmReconnectionProgress
' Project: TabellenWiedereinbinden
' Last change: 23.09.2007
'              by Paul Rohorzka (paul.rohorzka@softconcept.at)
' ShortDescr:  Fortschrittsanzeige bei Wiedereinbindung von Tabellen
' Description: Formular zur Fortschrittsanzeige beim Wiedereinbinden von
'              Tabellen mittels Interface IReconnectionListener
Option Compare Database
Option Explicit

Implements IReconnectionListener

Private Sub IReconnectionListener_BeginReconnect(ByVal Database _
                                                 As DAO.Database)
   ' Im Formular nichts zu tun.
End Sub

Private Sub IReconnectionListener_ReconnectingTable(ByVal Table _
                                                    As DAO.TableDef, _
                                                    Skip As Boolean)
   Me.txtTabelle.Value = Table.Name & " wird verbunden ..."
End Sub

Private Sub IReconnectionListener_ReconnectedTable(ByVal Table _
                                                   As DAO.TableDef)
   Me.txtTabelle.Value = Table.Name & " wurde verbunden."
End Sub

Private Sub IReconnectionListener_ReconnectingTableSkipped(ByVal Table _
                                                           As DAO.TableDef)
   Me.txtTabelle.Value = Table.Name & " wurde übersprungen."
   Debug.Print Table.Name & " wurde übersprungen."
End Sub

Private Sub IReconnectionListener_EndReconnect(ByVal Database As DAO.Database)
   DoCmd.Close acForm, Me.Name
End Sub
```

Verwendung:

```
' Variante 7
Private Sub Reconnect()
   Const FormName As String = "frmReconnectionProgress"
   Dim ReconnectInfoForm As Form_frmReconnectionProgress
   Dim Reconnector As JetReconnector

   DoCmd.OpenForm FormName
   Set ReconnectInfoForm = Forms(FormName)
   DoEvents

   Set Reconnector = New JetReconnector

   With Reconnector
```

```
      Set .Listener = ReconnectInfoForm
      .BackendPath = "…"

      .Reconnect
   End With

   Set Reconnector = Nothing
   Set ReconnectInfoForm = Nothing
End Sub
```

Unterschied zur Implementierung mit Ereignissen:

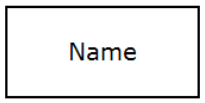
- Ereignisse können an beliebig vielen Stellen konsumiert werden
- Die im Interface definierten Methode wird für genau eine Implementierung aufgerufen.
- Wenn keine Referenz auf eine implementierende Klasse (hier: ein Listener) übergeben wurde, dürfen die Methodenaufrufe nicht durchgeführt werden!
- Verschwindender Performancevorteil (2‰ … 6‰ bei nicht sehr seriöser Messung)

E. Glossar und Abkürzungsverzeichnis

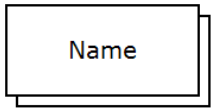
Begriff (Deutsch)	Begriff (Englisch)	Bedeutung/Verwendung in diesem Text
-	CRUD	Create, Read, Update, Delete Standardoperationen auf Daten in einer Datenbank-basierten Anwendung
Domäne	domain	Fachgebiet, in dem mit Hilfe einer Anwendung Aufgaben bearbeitet werden (z.B. Automobilbranche, öffentliche Verwaltung, Controlling, biotechnologische Forschung, Bildungswesen,...)
Eigenschaft	Property	Mittels Kombination aus `Property Let` bzw. `Property Set` (schreiben) / `Property Get` (lesen) zur Verfügung gestellter Wert
Feld	Field	Variable auf Modulebene
Methode	Method	Funktion (Schlüsselwort `Function`) oder Prozedur (Schlüsselwort `Sub`)
-	Refactoring	Interne Änderungen (Code aber auch Struktur) einer Anwendung ohne Änderungen der Funktionalität. Ziel von Refactoring ist eine bessere Wartbarkeit, Herausarbeiten von Gemeinsamkeiten, Skalierbarkeit, Wiederverwendbarkeit (Auslagerung von Funktionalitäten in Methoden, Module oder Komponenten)
-	RAD	Rapid Application Development RAD-Tools ermöglichen unter Zuhilfenahme von grafischen Benutzeroberflächen, Designer, Assistenten und anderen Hilfsmitteln die schnelle Entwicklung von Anwendungen. Im Gegensatz dazu stehen rein text-basierte Systeme.

F. Legende

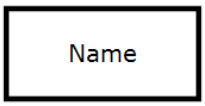
1. Zu den Architekturdiagrammen



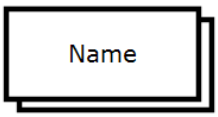
Eine Komponente im MDB/ACCDB-Format



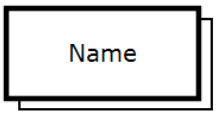
Mehrere Komponenten im MDB/ACCDB-Format



Eine Komponente im MDE/ACCDE-Format



Mehrere Komponenten im MDE/ACCDE-Format



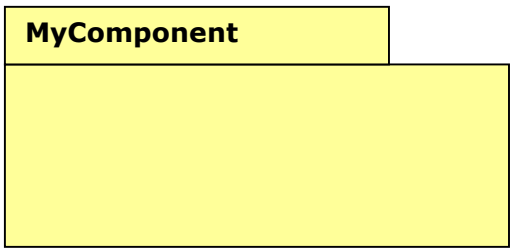
Mehrere Komponenten, tlw. im MDB/ACCDB-, tlw. Im MDE/ACCDE-Format

(Bibliotheks-)Verweis

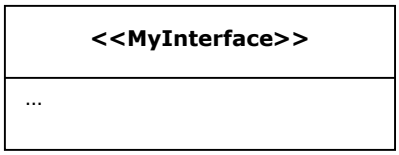
Objektreferenz

Datenfluss (verknüpfte Tabelle)

2. Zu den Klassendiagrammen

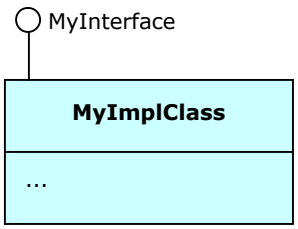


Komponente

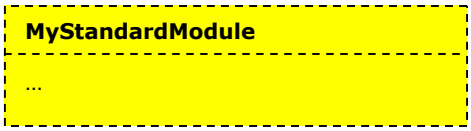


Interface

Klassenmodul



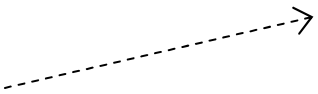
Klassenmodul das Interface implementiert



Standardmodul

+Info : IComponentInfo
+Initialize(IComponentFacade)

Memberliste:
+ ... Öffentliches Element
() ... Methode (Function oder Sub)
ohne Klammer ... Eigenschaft



Realisierung
(Objektreferenz)

G. Literatur

- Steve McConnell, Code Complete, 2nd Edition, Microsoft Press, 0-7356-1967-0, http://www.cc2e.com
- Sönke Petersen, Benutzerdefinierte Berichte, Skriptum AEK5, http://donkarl.com/AEK/AEKDownloads/AEK5_Berichte.zip
- Josef Pötzl, AccEPT - Access Error Prevention Table, http://access.joposol.com/
- Karsten Pries, Klassenmodule, Skriptum AEK2, http://donkarl.com/AEK/AEKDownloads/AEK2_Klassen.zip
- Paul Rohorzka, Klassenmodule konkret, Skriptum AEK8, http://donkarl.com/AEK/AEKDownloads/AEK8_Klassen_konkret.zip
- Paul Rohorzka, Praxiseinsatz von Klassen, Skriptum AEK7, http://donkarl.com/AEK/AEKDownloads/AEK7_Klassen.zip