AEK 10 - 10. Access Entwickler Konferenz
Nürnberg, 6./7. und 20./21. Oktober 2007

Modulare Anwendungsentwicklung in Microsoft® Access

Prinzipielle Strategien

Modularität im Code

Modularität im Design

softconcept, Wien
www.softconcept.at

Paul Rohorzka
paul.rohorzka@softconcept.at

Ziele

- Reflektion über typische Vorgehensweisen
- Strategien zur Vermeidung typischer Fehler
- Sensibilisierung für Anwendungsdesign
- Bessere Wartbarkeit der Anwendungen
- Erhöhte Qualität der Anwendungen

Themen

- **Teil I - Modularität im Code**
  - Code kleinräumig
  - Wohin mit dem Code?
  - Fehlerbehandlung
  - SQL im Code vs. gespeicherter Abfragen
- **Teil II - Modularität im Design**
  - Bibliotheken und Komponenten
  - Vorschlag für eine modulare Architektur
  - Aufbau einer Komponente
- **Verstreut: Grundsätzliche Überlegungen**
  - Warum modular?
  - Was ist Access, was sind Tabellen?
  - Prinzipien und Strategien

Teil I
Modularität im Code

Workshop: Evolution eines Stückchens Code

- Aufgabenstellung:
  Wiedereinbinden verknüpfter Jet-Tabellen
- Pseudocode:

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

⇒ Los geht's!

Code!

ReconnectJetBackend()
Kritik an Variante 1

- Kritikpunkte:
  - Sichtbarkeit der Methode nicht klar
    - `Public`?
  - Parameterübergabe
    - `ByRef`/`ByVal`?
  - Zweimalige Verwendung von `CurrentDb`
    - Konsequenz?
  - `CurrentDb` ist eine Methode, keine Eigenschaft
    - ?
  - Literal `";DATABASE="` kommt zweimal vor
    - Konsistenz?
  - Literal `10` ist abhängig von `";DATABASE=„`
    - Konsistenz?

⇒ Suche nach Abhilfe

Code!

ReconnectJetBackend()
Kritik an Variante 2

- Kritikpunkte:
  - Zu „technisch“
  - Die eigentliche Funktionalität ist
    - versteckt in vielen Details
    - nicht gut erkennbar
- Lösungsansatz:
  - Teilaspekte der Funktionalität in Methoden auslagern:
    - Kleine, klar abgegrenzte Teile
    - Sprechende Namen!
- ⇒ „Refactoring“

Code!

ReconnectJetBackend()
Vorteile von Variante 3

- Status für Variante 1:
  - Kleine und leicht überschaubare Einheiten
  - Die Hauptmethode ist
    - wieder nahe am Pseudo-Code
    - recht nahe an natürlicher Sprache

- $\Rightarrow$ Zeit für eine erste Zwischenbilanz

Wiedereinbinden verknüpfter Tabellen
Erste Zwischenbilanz

- Möglichst wenig implizites Wissen voraussetzen
  - Zugriffsspezifizierer (`Public`, `Private`, `Friend`) explizit angeben
  - Art der Parameterübergabe (`ByVal`/`ByRef`) explizit angeben
  - Art des Element-Typs (`CurrentDb()` vs. `CurrentDb`) explizit angeben
- Kritisch gegenüber der Verwendung von `CurrentDb()`:
  - Jedes Mal ein neues Objekt!
  - ⇒ Hilfsvariable in zusammenhängenden Codeabschnitten
  - Eventuell Singleton-Pattern (Newsgroup-Sprech: „`CurrentDbC`“)
- Aufteilung des Codes in mehrere klar abgegrenzte Einheiten
  - Klarere Verständlichkeit der einzelnen Code-Teile (meist Methoden)
  - Code liest sich leichter
  - „fast wie Prosa“

Prinzipielle Strategien:
Fehler sollen gar nicht passieren können

- Kleinst möglicher Gültigkeitsbereich
- Sichtbarkeit möglichst stark einschränken
  - interne Details: `Private`!
- Alle Zugriffsmodifizierer explizit angeben
  - Default ist meist `Public`!
- Systeme aus mehreren Komponenten:
  - Modifizierer `Friend` (sichtbar innerhalb der Komponente)

Prinzipielle Strategien:
Fehler sollen gar nicht passieren können

- 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 wird der Wert
                  ' der übergebenen Variable geändert!
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
                  ' des (lokalen) Parameters geändert werden.
End Function
```

Prinzipielle Strategien:
Auf logischer Ebene arbeiten

- Auf technischer/implementatorischer Ebene:
  - „Führe Statements X, Y und Z aus wenn jemand auf die Schaltfläche btnSave klickt."
- Auf logischer Ebene
  - „Führe die Aktion Abc aus, wenn der Benutzer speichern will."
- Umsetzung in VB(A) und Access:
  - Logisch benannte Konstanten statt Literale
  - Nur eine klar abgegrenzte Teilfunktion pro Methode
  - Tool-Methoden und –Klassen
    - rasch die technische Ebene verlassen und
    - auf die logische Ebene kommen.

Wiedereinbinden verknüpfter Tabellen
Erweiterung der Anforderungen

- Aufgabenstellung
  - Benutzerfeedback über den Fortschritt der Einbindung
  - Anzeigen der aktuell verarbeiteten Tabelle in einem Textfeld
- Vorarbeit
  - Formular
  - mit Textfeld
- Erster Ansatz
  - Ausgabe direkt aus der Schleife im Textfeld
  - ⇒ Los geht's!

Code!

ReconnectJetBackend()
Kritikpunkte an Variante 4

- ♦ Kritikpunkte
  - Implizite Voraussetzungen
    - Das Formular muss `frmReconnectionProgress` heißen
    - Das Formular muss geöffnet sein
    - Das Steuerelement muss `txtTabelle` heißen
    - Das Steuerelement muss ein Textfeld sein (Eigenschaft `Value`!)
  - Keiner dieser Punkte kann vom Compiler überprüft werden!
    - Fehler treten also erst zur Laufzeit auf ☹
  - Diese Vorbedingungen müssen an verschiedenen Stellen (Entwurf, Code) erfüllt werden.
  - `DoCmd.Close()` schließt eventuell das falsche Access-Objekt!
  - Schlechte Wartbarkeit
    - Alle Zusammenhänge müssen bedacht werden

⇒ Los geht's!

Code!

ReconnectJetBackend()
Bemerkungen zu Variante 5

- Vorteile
  - Keinerlei Annahmen in `ReconnectJetBackend()` über Namen von Formular oder Steuerelement
  - Wesentlich klarere Trennung von Funktionalität und Darstellung
  - Bessere Wartbarkeit

    Bei
    - Änderung des Formularnamens
    - Änderung des Steuerelements

    Anpassungen nur mehr im Code des Formulars
- Was immer noch stört:
  - Potenzielle Fehlerquellen (vom Compiler nicht überprüfbar):
    - Name des Formulars
    - Name und Typ des Textfeldes

Wiedereinbinden verknüpfter Tabellen
Zweite Zwischenbilanz

- Vom Compiler nicht überprüfbar:
  - Name des Formulars
- Vom VB6-Compiler nicht überprüft wird (obwohl er könnte):
  - Typ des Steuerelements
- Nach wie vor Vermischung von Funktionalität und Darstellung
- Alternative Verwendung des Tabellennamens erfordert immer Anpassung von `ReconnectJetBackend()`:
  - 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!

Prinzipielle Strategien:
Fehler vom Compiler erkennbar machen

- Vermeidung nicht trivialer Literale
- Triviale Literale (je nach Kontext):
  - 0, 1, -1
  - 2 (als Faktor oder Divisor)
  - "" (leerer String)
- Nichttriviale Literale
  - Tabellennamen
  - Formularnamen
  - Dateiendungen
  - ...
- Wenigstens in Form von Konstanten zentral definieren
- Dort definieren, wo sie hingehören (als Eigenschaft betrachten):
  - Eigenschaft der gesamten Anwendung (z.B. Titel der Anwendung für `MsgBox`): Als globale Konstante
  - Eigenschaft eines Moduls (z.B. Name des Moduls für Fehlermeldungen): Als Modul-Konstante
  - Nur in seltenen Fällen Konstanten auf Methodenebene definieren

Prinzipielle Strategien:
Fehler vom Compiler erkennbar machen

- 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
```

  - Eine Schleife „von einer Grenze bis zu einer anderen Grenze“
- Stattdessen

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

  - Eine Schleife „über alle Elemente“
  - Wesentlich klarer und weniger fehleranfällig

Prinzipielle Strategien:
Fehler vom Compiler erkennbar machen

- 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 = ...
```

    - String-Literal versteckt!
    - Fehler erst zur Laufzeit ☹
    - Typ des Steuerelements erst zur Laufzeit
  - Stattdessen

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

Vorsicht!
Funktioniert nicht in
allen Access-Kombinationen.
Absolut problemlos mit A2k.

Prinzipielle Strategien: **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`?
  - Besser wäre

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

- `Currency` statt `Double`
  - Vier dezimale Nachkommastellen *exakt*
  - `Double` und `Float` können das nur in Spezialfällen!
  - Z.B. auch gut für Prozentsätze verwendbar

Prinzipielle Strategien: **Strenge Typisierung**

- Beispiel: Bezug auf ein Steuerelement eines Unterformulars aus dem Hauptformular
- Statt

```
... = Me.frmsBestellungen.Form.Artikel.Value
```

- Streng typisiert

```
Dim objfrmsBestellungen As Form_frmKunden_frmsBestellungen
Set objfrmsBestellungen = Me.frmsBestellungen.Form
... = objfrmsBestellungen.Artikel.Value
Set objfrmsBestellungen = Nothing
```

- Noch besser
  - Keinen direkten Zugriff auf Steuerelemente eines anderen Formulars, stattdessen eine Eigenschaft vorsehen.
  - Im Unterformular

```
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.Artikelname
Set objfrmsBestellungen = Nothing
```

Prinzipielle Strategien:
Wenig programmieren!

- Stattdessen
  - Designer verwenden
    - Voreinstellungen für Formulare in der Entwurfsansicht vornehmen
    - Abfragen nicht per SQL in VBA zusammenbasteln, sondern gespeicherte Abfragen verwenden
  - Mit Code-Generatoren
    - gefährliche Teile generieren lassen und
    - auf eine vom Compiler überprüfbare Ebene bringen
  - Frameworks anwenden
    - „gefährliche" Codeteile zentralisiert spezifizieren und
    - dann per Eigenschaft oä darauf zugreifen

Prinzipielle Strategien:
Wenig programmieren!

- Tool: MyClasses
  - Codegenerator
  - Erzeugt Klasse `CMyForms` für alle Formulare
    - Property `<FormName>` für jedes Formular
    - Datentyp `CFormCapsule`
    - Singleton-Instanz `MyForms` (in Standard-Modul)
  - Methoden von CFormCapsule:
    - `OpenIt()`
    - `OpenAsDialog()`
    - `CloseIt()`
  - Entsprechend auch für Berichte

Demo

Wiedereinbinden verknüpfter Tabellen
Aus der zweiten Zwischenbilanz

- Nach wie vor Vermischung von Funktionalität und Darstellung
- Alternative Verwendung des Tabellennamens erfordert immer Anpassung von `ReconnectJetBackend()`:
  - Bezeichnungsfeld (`Caption` statt `Value`)
  - Listfeld (Manipulation der `RowSource` oder temporäre Tabelle)
  - Endlosformular (Schreiben in temporäre Tabelle)
  - Fortschrittsbalkens
  - ... und das obwohl die eigentliche Funktionalität unverändert bleibt!

⇒ Was tun?

Code!

Wiedereinbinden verknüpfter Tabellen
Ansatz 1: Entkopplung durch Ereignisse

- Idee/Wunsch:
  - `ReconnectJetBackend()` signalisiert nach außen das Einbinden einer neuen Tabelle
  - Andere(r) Programmteil(e) reagieren darauf
- Technisches Vehikel dazu:
  - Ereignisse
- Vorbedingung
  - Code liegt in einer Klasse (Standardmodul kann keine Ereignisse auslösen oder darauf reagieren)

Code!

JetReconnector – mit Ereignissen
Vorteile

- 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

Wiedereinbinden verknüpfter Tabellen
Ansatz 2: Entkopplung durch Interface

- Idee/Wunsch:
  - Callback-Prinzip (Hollywood)
  - `ReconnectJetBackend()`
    - erhält eine Referenz auf ein an sich beliebiges Objekt
    - das (zumindestens) ein definiertes Set von Methoden und Eigenschaften bietet
    - ruft die jeweils passende Methode des Objekts auf
- Technisches Vehikel dazu
  - Interface

Wiedereinbinden verknüpfter Tabellen
Entkopplung durch Interface – HowTo

- Interface (z.B. `IReconnectionListener`)
  - definiert Methoden, die von der Methode `Reconnect()` des JetReconnectors aufgerufen werden können.
- Eine Klasse implementiert das Interface
  - Schlüsselwort `Implements`
  - Kann auch die CBF-Klasse eines Access-Formulars sein
- Eine Instanz dieser Klasse wird an den JetReconnector übergeben.
- Der Code in `Reconnect()` ruft die Methoden der übergebenen Klasse auf.

Code!

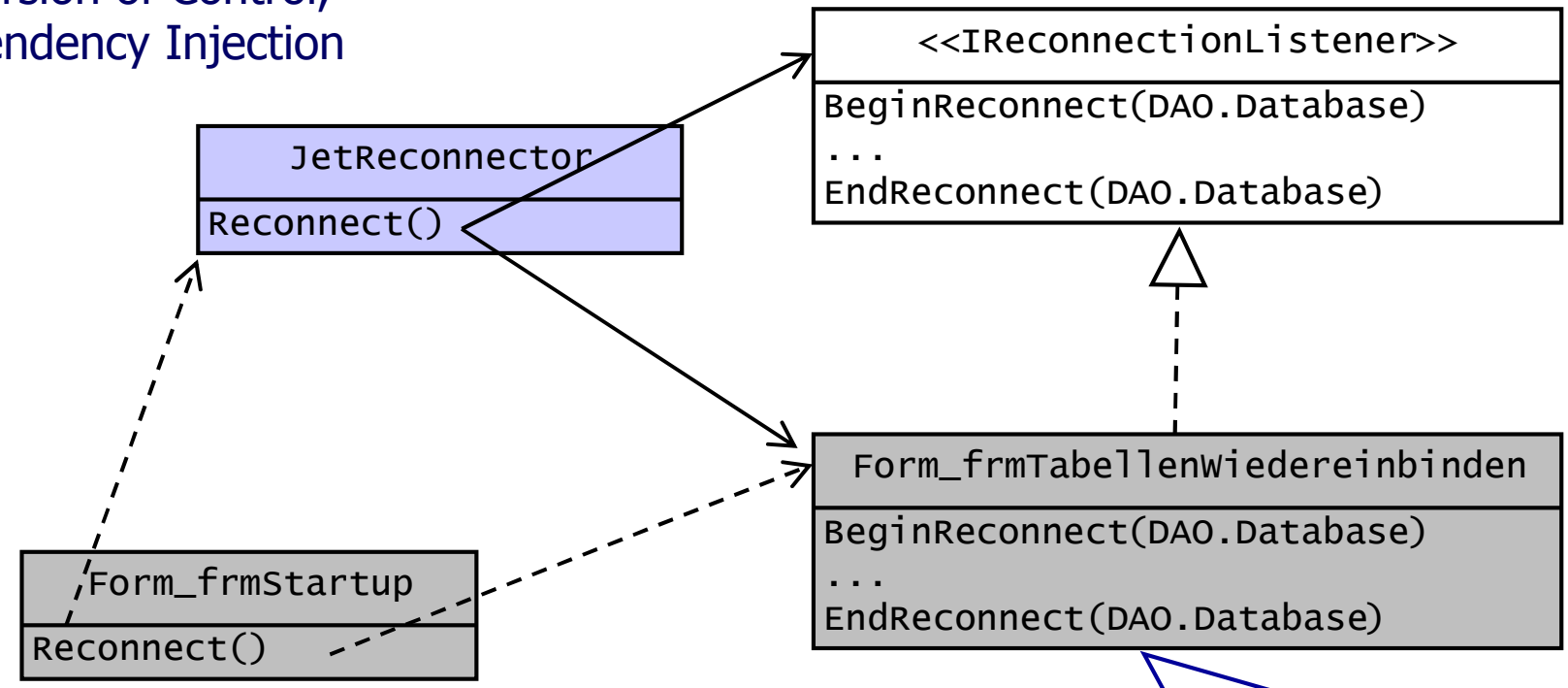
JetReconnector – unter Verwendung eines Interface
Unterschiede zur Arbeit mit Ereignissen

- Ereignisse können an *beliebig vielen* Stellen konsumiert werden
- Die im Interface definierte 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!
  - `If (Not ... Is Nothing) Then ...`
- Die Lösung mit Interface funktioniert auch late bound!
- Verschwindender Performancevorteil: 2‰ … 6‰
  - (bei nicht besonders seriöser Messung)

JetReconnector – unter Verwendung eines Interface
Die Struktur

- Zusammenhänge der beteiligten Elemente

Inversion of Control,
Dependency Injection



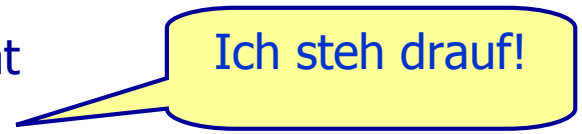
Kann
ohne Änderung am JetReconnector
durch anderen Listener ersetzt werden!

Grundsätzliche Überlegungen
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

Grundsätzliche Überlegungen
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
  - Nachschlagefelder



Prinzipielle Strategien
Zuerst die Datenstruktur!

- Die Datenstruktur auf Papier entwerfen
  - Welche Daten müssen gespeichert werden?
  - Welche Zusammenhänge bestehen zwischen den Daten
  - Großes Papier!
  - Viel Papier!

Prinzipielle Strategien
Zuerst die Datenstruktur!

- Die Datenstruktur im Backend implementieren
  - Benennungsschema festlegen und strikt einhalten
  - Tabellen
  - Felder
    - Datentypen
    - Eingabe erforderlich
    - Leere Zeichenfolge (geänderter Standard ab A2002!)
    - Indizes (eindeutige Felder, Mehrfelderindizes)
    - Gültigkeitsregeln (auch auf Tabellenebene)
  - Referenzielle Integrität
  - Von vielen verpönt, IMHO dennoch sehr zu empfehlen:
    - Beschriftungen
    - Nachschlagefelder

Prinzipielle Strategien
Zuerst die Datenstruktur!

- Mithilfe einiger Testdatensätze typische Szenarien auf Tabellenebene durchspielen
  - Kann ich jede Information eingeben?
  - Sind unsinnige Werte möglichst verhindert?
  - Muss ich ggflls. weiter normalisieren?

Überlegungen zum UI sind hier
ABSOLUT FEHL
am Platz!

Trennung Funktionalität – GUI:
In Formularen nur Code zur GUI-Steuerung

- Code nur dann in CBF
  - wenn er unmittelbar mit Steuerung des UI zu tun hat
- Jeglicher Businesscode unabhängig von Formularen
  - in Standardmodule
  - in Klassenmodule

Trennung Funktionalität – GUI:
Keine Referenzierungen „aus dem Off“

- ... wie

```
Forms!frmKunden!Titel
```

- Stattdessen
  - Code steuert GUI
    - den Code CBF schreibendort
    - Referenzierungen direkt über Me (wie Me.Titel.Value)
  - Code in Modul ausgelagert, nur einzelne Werte aus Steuerelementen
    - Werte als Parameter übergeben
  - Code benötigt tatsächlich ein Formularobjekt
    - Referenz auf das Formular übergeben (Me)

Strategien
Möglichst enge Schnittstellen

- Code-Teile
  - durch möglichst genau definierte und minimale Schnittstellen miteinander kommunizieren lassen
- Gut geeignet:
  - Parameterübergabe
  - Rückgabewerte
  - Ereignisse
- Schlecht wartbar:
  - Referenzierungen „aus dem Off“
  - Globale Variablen

Abfragen: gespeichert vs. SQL per Code: **Typisches Szenario einer Abfrage per Code**

- Preiserhöhung in einer Warengruppe:

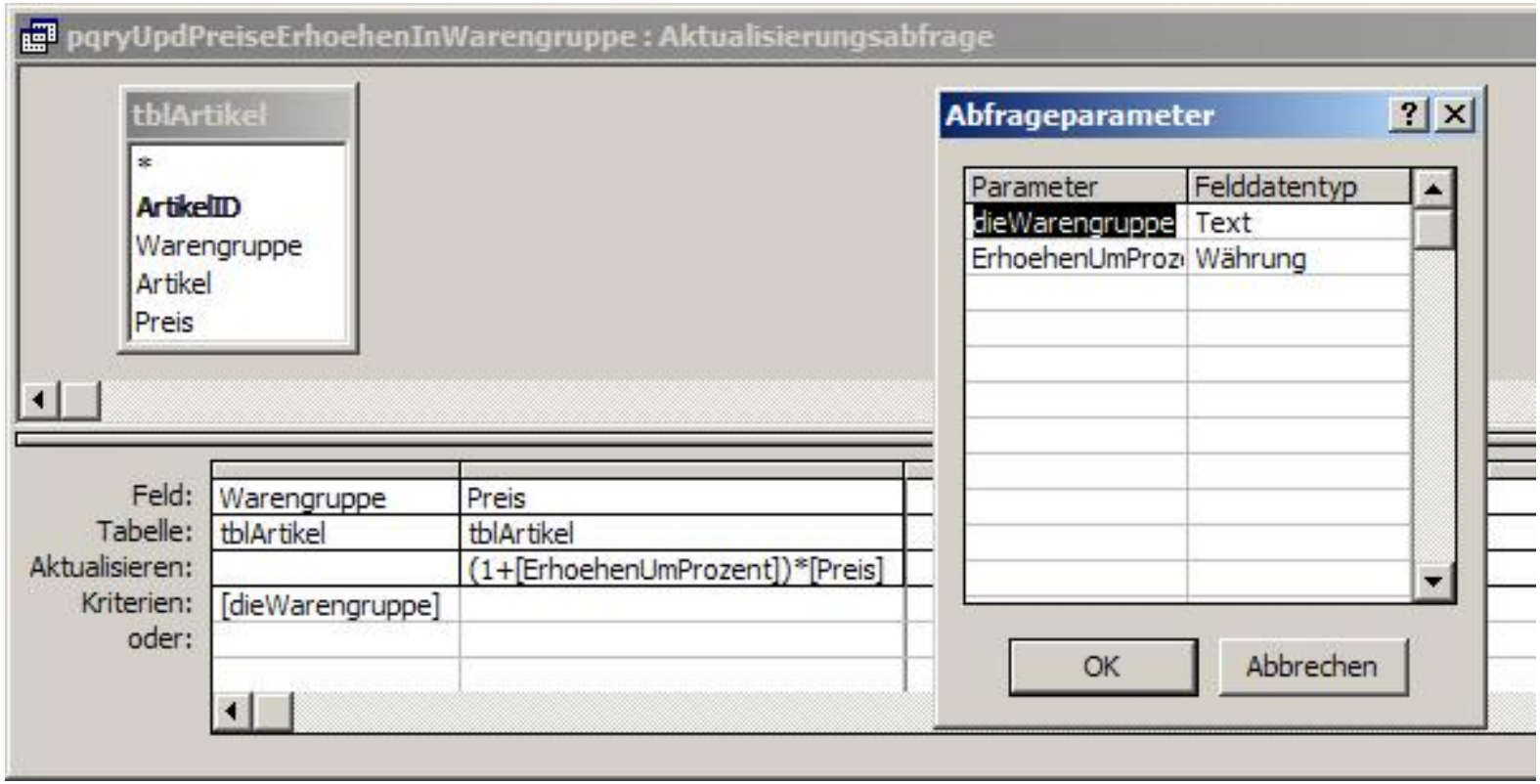
```
Dim strSQL As String
Dim dbs As DAO.Database

strSQL = "UPDATE tblArtikel " & _
         "SET Preis = " & Str(1 + curErhoehung) & _
         " * Preis " & _
         "WHERE Warengruppe = " & _
         """" & strWarengruppe & """"
Set dbs = CurrentDb()

dbs.Execute strSQL, dbFailOnError
... = dbs.RecordsAffected

Set dbs = Nothing
```

Abfragen: gespeichert vs. SQL per Code: **Nachteile von SQL per Code**

- Keine interaktive Manipulation (Joins!)
  - Kein interaktiver Test
  - Schwerer verstehbar
    - Intensive String-Konkatenierung
  - Schwerer wartbar
  - Abfrageplan ist nicht gespeichert
- Nachteile gespeicherter Abfragen
  - SQL-Code ist (mit Boardmitteln) nicht durchsuchbar
    - Lässt sich aber sehr leicht beheben
  - Mehr Code zum Ausführen einer Aktionsabfrage
    - Kann aber leicht verallgemeinert werden

Abfragen: gespeichert vs. SQL per Code: **Verwenden einer gespeicherten Abfrage**

- Parameterabfrage



Abfragen: gespeichert vs. SQL per Code: **Verwenden einer gespeicherten Abfrage**

- Ausführen der Paramterabfrage 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
```

Fehlerbehandlung
Wo Fehlerbehandlung vorsehen?

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

Fehlerbehandlung
Einige Gedanken

- Was passiert mit nicht abgefangenen Fehlern?
  - Werden nach oben weitergereicht
  - Wenn kein Handler gefunden wird: Std.-Access-Meldung

  Daher
  - Fehlerhandler in jeder Ereignisprozedur
    oder
  - in von Ereignisprozedur aufgerufener Methode
- Benutzerfeedback in der Fehlerbehandlung

  Auch hier gilt
  - Trennung zwischen Funktionalität und UI!
  - Keine MsgBoxen in Nicht-UI Code!
    - Stattdessen: Eigene Fehlercodes definieren

      z.B.: `vbObjectError + 12345`

Fehlerbehandlung
Form der Fehlerbehandlung

- Klassische Form

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

    ' Eigentlicher Nutzcode

Exit_:
    Exit Sub

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

Fehlerbehandlung
Alternative Form der Fehlerbehandlung I

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

    ' Eigentlicher Nutzcode

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
```

Fehlerbehandlung
Alternative Form der Fehlerbehandlung II

```
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
```

Teil II
Modularität im Design

Bibliotheken und andere Komponenten
Warum?

- Vorteile
  - Wiederverwendbarkeit von Code
  - Nachliefern von Funktionen
    - Programm-Module (Lizenzierung in Standard-Produkten)
    - AddIns, PlugIns
  - Verwendung bzw. Weitergabe von Funktionalität ohne Code freilegen zu müssen
- Nachteile
  - Mehr als eine Datei muss verteilt werden
  - Verweis Thematik

Bibliotheken und andere Komponenten
Was geht und was nicht geht

- Code von eingebundenen MDB/ACCDB-Komponenten
  - kann durchgesteppt werden (Debugging) ☺
  - kann verändert werden ☺
  - kann aber nicht abgespeichert werden! ☹
- `CurrentDb()` meint immer die gestartete Access-Datei
  - Zugriff auf die Komponenten-Bibliothek nur mittels `CodeDb()`
  - Warum standardmäßig meistens `CurrentDb()`?
  - Außer bei AddIns und Komponenten sollte man eigentlich immer `CodeDb()` verwenden.
- `DoCmd`- und `Application`-Methoden funktionieren oft nicht
  - `DoCmd.TransferSpreadsheet()`, …
  - `Application.DLookup()`, …
  - Intern wird offenbar `CurrentDb()` verwendet

Vorschlag für eine modulare Anwendungsarchitektur
Merkmale

- Komponentenbasiert
  - Einzelne Aspekte der Funktionalität in eigenen Dateien
  - Wiederverwendung
  - Auslagerung von Funktionalität in MDB/ACCDB-Dateien
  - Entwurfsmodus für den Endbenutzer!
  - Nachrüsten von Modulen
- Modularisierung
  - Teile der Anwendung können in eigenen Dateien implementiert werden.
    - vom Endkunden getrennt lizensiert?
- Großteils MDE/ACCDE-basiert
  - Code-intensive Komponenten zum Schutz des Codes kompiliert als MDE/ACCDE-Dateien
- MDB/ACCDB-Module möglich
  - Berichtseditor von Access zur Anpassung der Berichte für Endanwender

Vorschlag für eine modulare Anwendungsarchitektur
Die Ausgangsbasis

- Problematik
  - Verweis-Thematik
    - vorheriger Ort der Komponente funktioniert nicht mehr
    - Auflösungsalgorithmus findet falsche/alte Version
  - Prinzipiell nicht möglich:
    - MDE mit Bibliotheksverweis auf MDB

- Überlegungen
  - Warum eigentlich Verweis?
    - Um auf Code der Komponente zugreifen zu können
  - Klassisches Late-Binding mit Access-Komponente?
    - `CreateObject()` oder `GetObject()` funktionieren nur bei registrierten Komponenten ☹

Vorschlag für eine modulare Anwendungsarchitektur
Grundlegender Gedanke

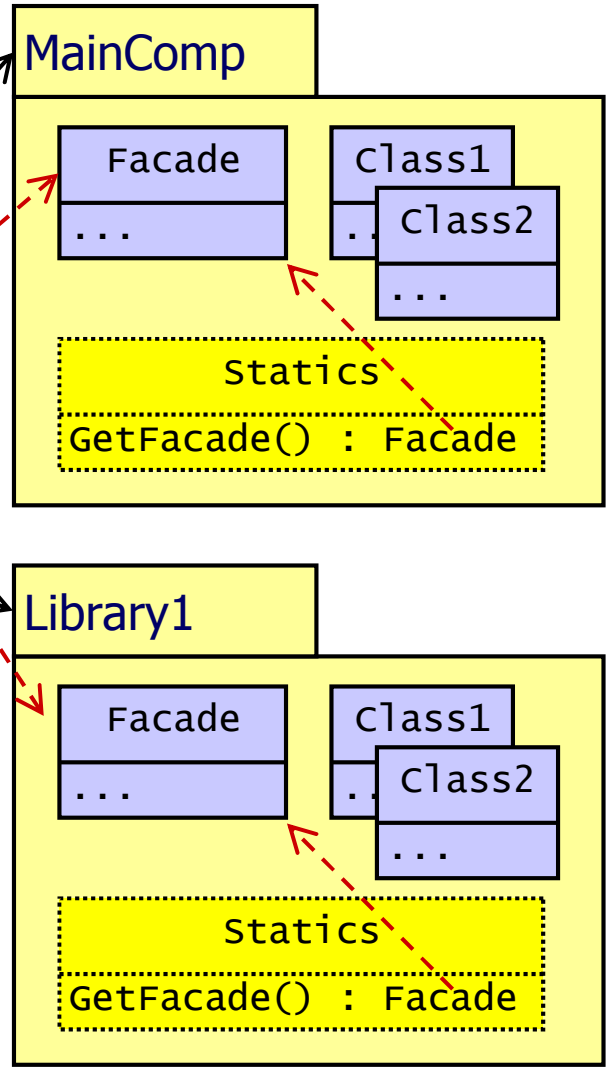
- Idee
  - Die Module erhalten Referenzen auf Objekte anderer Module
  - und stoßen über Methoden dieser Objekte Funktionen an
- Realisierung: Starter-MDB/ACCDB
  - Stellt Bibliotheksverweise auf alle Komponenten sicher (Registry, Konfiguration, Dateiauswahl, ...)
  - Besorgt sich für jede Komponente eine Instanz einer Fassade (Singleton)
  - Reicht Referenzen der Fassaden an andere Komponenten weiter
  - Startet die Applikation über Methode der Hauptfassade
- $\Rightarrow$ „Late Binding für Access-Komponenten“

Vorschlag für eine modulare Anwendungsarchitektur
Grundlegender Gedanke grafisch dargestellt

Starter.mdb

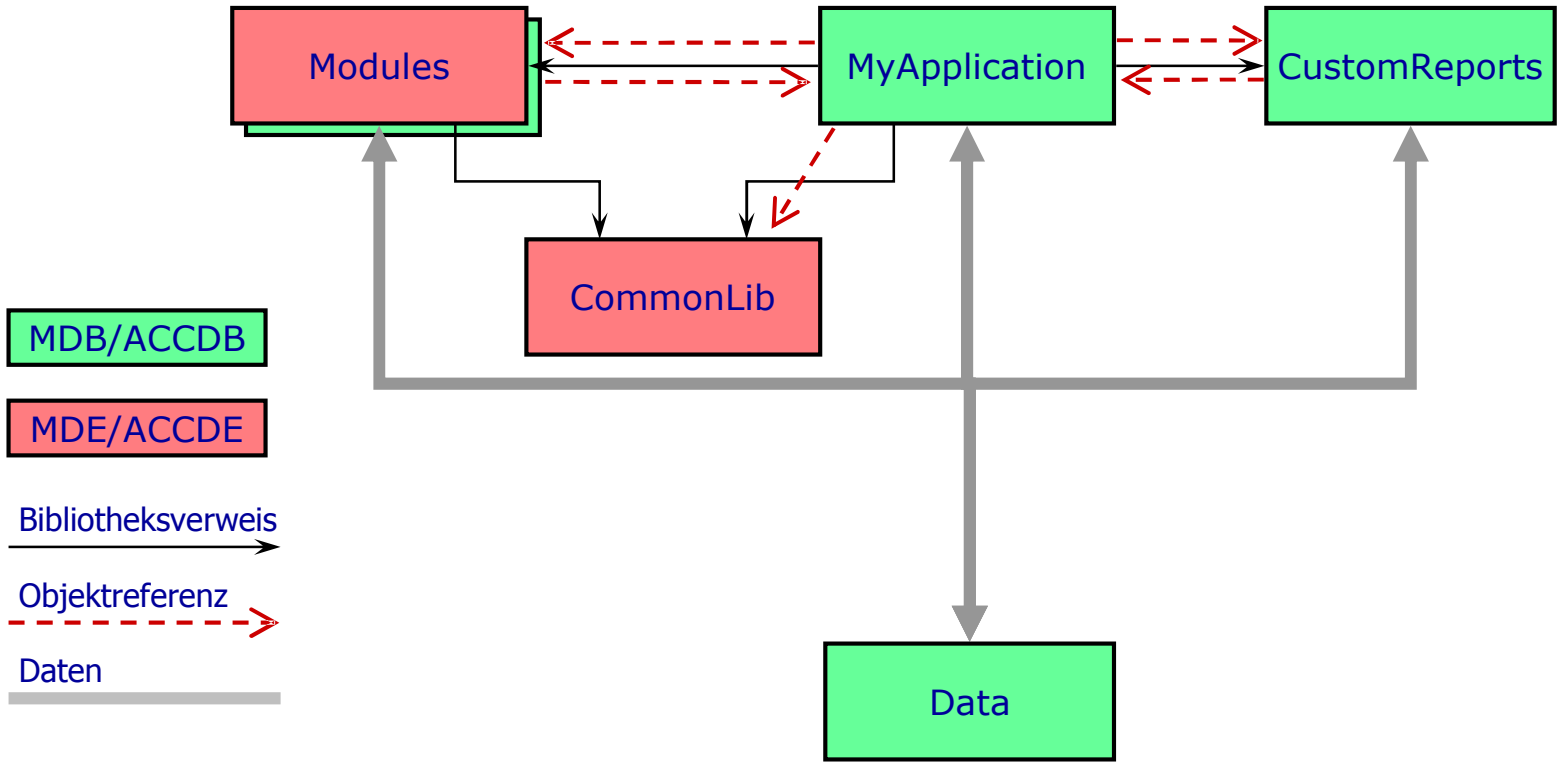
Fassadenklassen:
`Instancing` ist `Public`*`NotCreatable`*!

```
Dim MainFac As MainComp.Facade
Dim Lib1Fac As Library1.Facade

' Fassaden-Instanzen besorgen
Set MainFac = MainComp.Statics.GetFacade()
Set Lib1Fac = Library1.Statics.GetFacade()

' Komponenten mit einander bekanntmachen
Set MainFac.Library = Lib1Fac

' Die eigentliche Anwendung starten
MainFac.StartUp()

' Aufräumen
Set Lib1Fac = Nothing
Set MainFac = Nothing
```

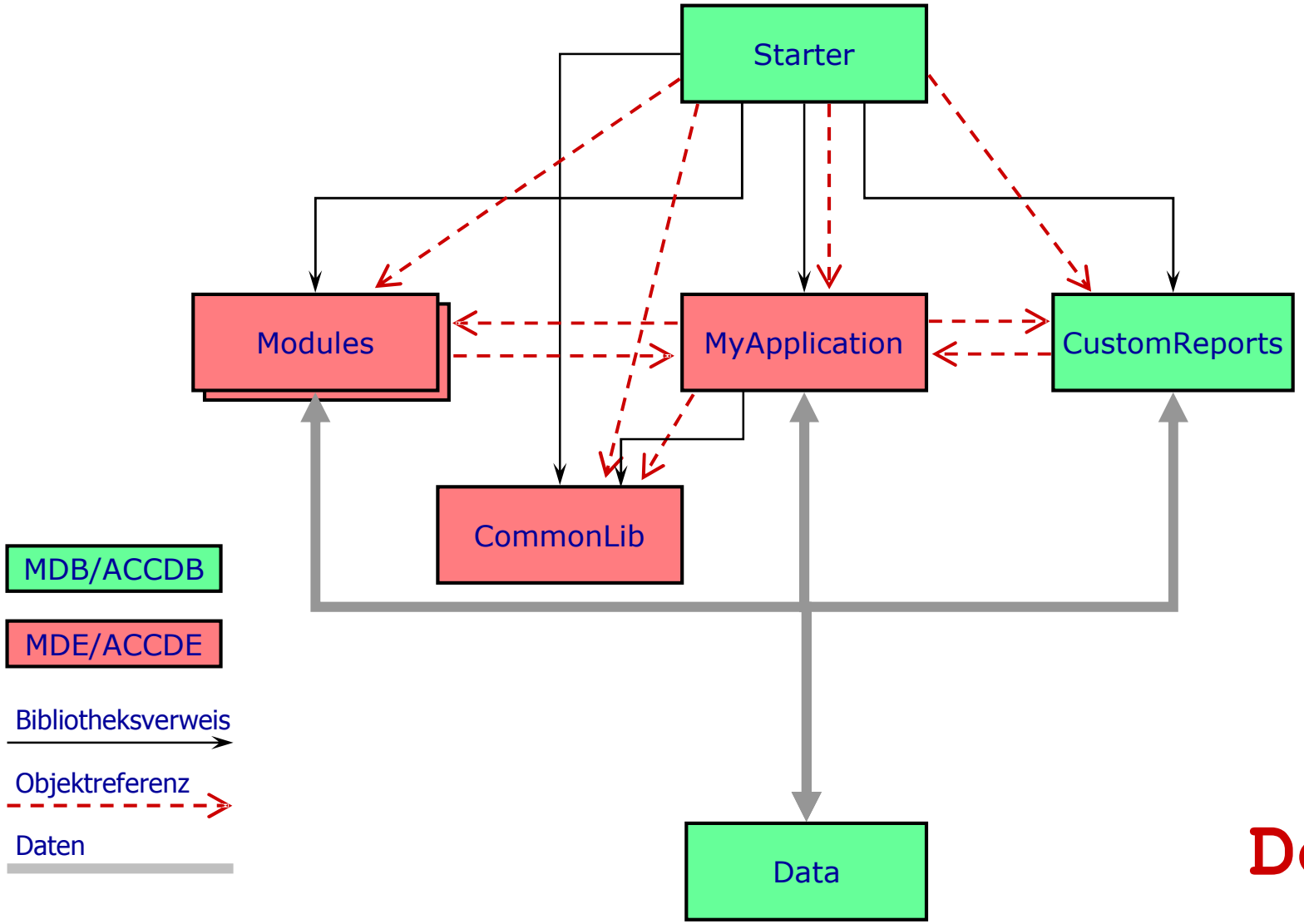


Vorschlag für eine modulare Anwendungsarchitektur
Struktur bei der Entwicklung

- Starter-MDB ist nicht involviert



Vorschlag für eine modulare Anwendungsarchitektur
Struktur in Produktion



Demo?

Vorschlag für eine modulare Anwendungsarchitektur
Struktur in Produktion – Vorsicht!

- Der Starter ist der Host für alle anderen Komponenten
  - Auch für die Haupt-MDE!

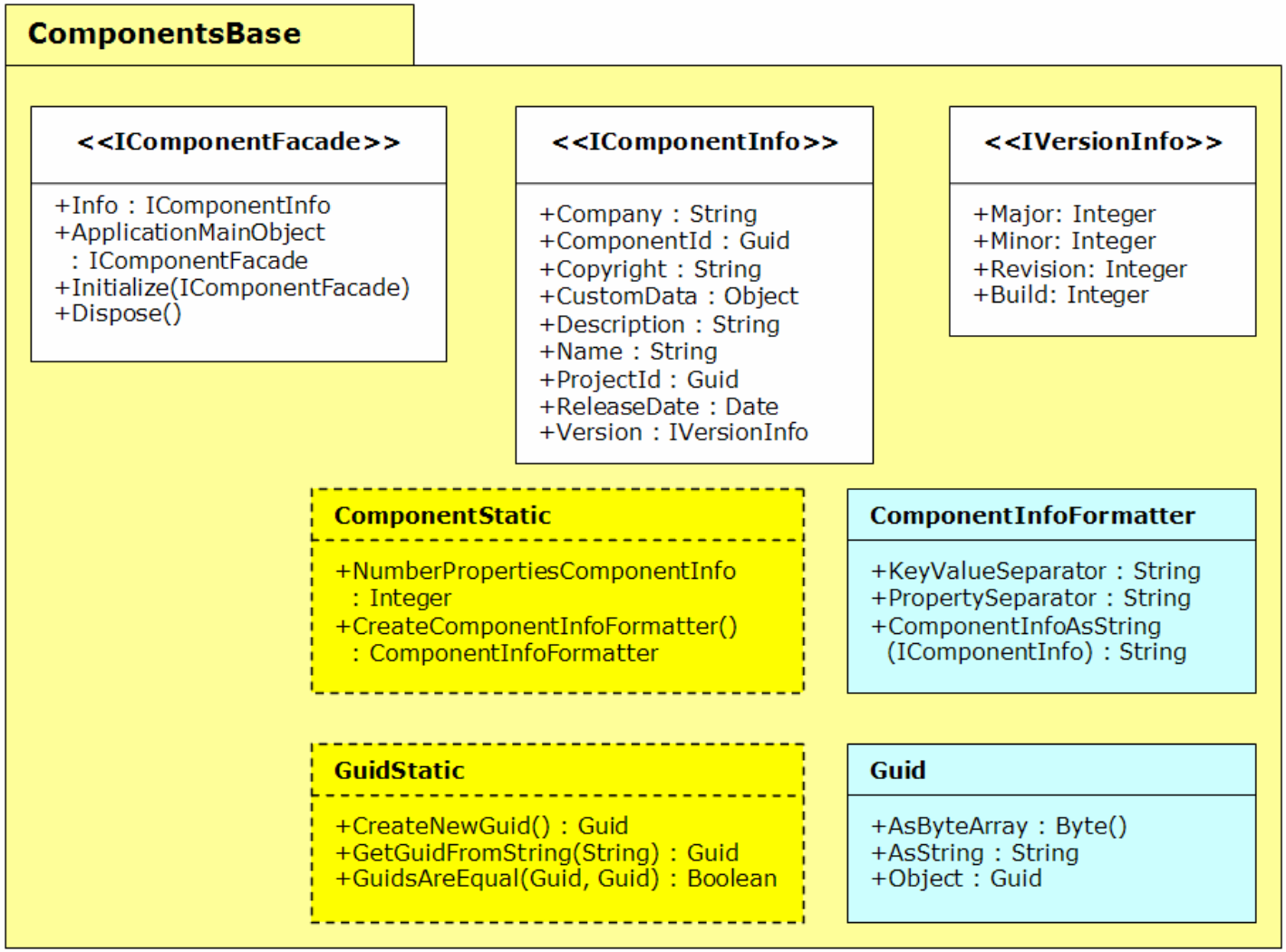
Auswirkungen:

- `CodeDb()` statt `CurrentDb()`
  - Ersatz für `DoCmd.Xxx()`, ... verwenden
- Symbolleisten (und Menüs):
  - Änderungen in Haupt-MDE und Starter.mdb durchführen!
  - Kein direktes Öffnen von Formularen der Haupt-MDE
  - Aufruf von Methoden der Haupt-MDE aus Starter.mdb
    - Bei Aktion: `MyApplication.ShowKundenDetails`

Vorschlag für eine modulare Anwendungsarchitektur
Aufbau einer Komponente – Allgemeines

- Nutzfunktionalität hauptsächlich in Klassen
  - Voraussetzung für „Late-Binding für Access-Komponenten“
- Bereitstellen von Factory-Methoden für Klassen
  - Eigenschaft `Instancing` von Klassen aus Access-Komponenten:
    - `Private` – die Klasse ist nur innerhalb der Komponente sichtbar
    - `PublicNotCreatable` – die Klasse
      * ist auch außerhalb der Komponente sichtbar
      * kann aber von außen nicht instanziiert werden
  - Daher innerhalb der Komponente instanziieren (Factory)
- Verbergen aller Interna
  - Details des Codes in der Komponente von außen nicht sichtbar!
  - Zugriffsmodifizierer `Friend`! – nur innerhalb der Komponente
- Identifikation der Komponenten
  - Grundlegende Informationen (→ `IComponentInfo`)

Vorschlag für eine modulare Anwendungsarchitektur
Infrastruktur in ComponentsBase



Vorschlag für eine modulare Anwendungsarchitektur
ComponentsBase – das Interface IComponentFacade

- IComponentFacade definiert die erforderlichen Members einer Fassade:
  - Info As IComponentInfo
    - Liefert Referenz auf Objekt mit Informationen über die Komponente
  - Initialize(IComponentFacade)
    - Initialisiert die Komponente
    - Paramter ist Fassade der Hauptkomponente
  - ApplicationMainObject As Object
    - Liefert Referenz auf Fassade der Hauptkomponente
  - Dispose()
    - Freigabe von Ressourcen
    - Auflösen von Zirkelreferenzen!

Vorschlag für eine modulare Anwendungsarchitektur
ComponentsBase – das Interface IComponentInfo

- IComponentInfo definiert die erforderlichen Members einer Klasse für Informationen über die Komponente:
  - Name As String
    - Name der Komponente
  - ComponentId As Guid
    - Eindeutige Kennung für diese Komponente
  - Version As IVersionInfo
    - Version der Komponente
  - ProjectId As Guid
    - Eindeutige Kennung für das Projekt zu dem die Komponente gehört
  - Description As String
    - Beschreibung
  - CustomData As Object
    - Zusätzliche Spezial-Informationen (z.B. History, ...) der Komponente

Vorschlag für eine modulare Anwendungsarchitektur
ComponentsBase – das Interface IVersionInfo

- IVersionInfo definiert die erforderlichen Members einer Klasse für die Version der Komponente:
  - Major As Integer
  - Minor As Integer
  - Revision As Integer
  - Build As Integer
- Ermöglicht vierteilige Versionsangabe 3.1.4.0

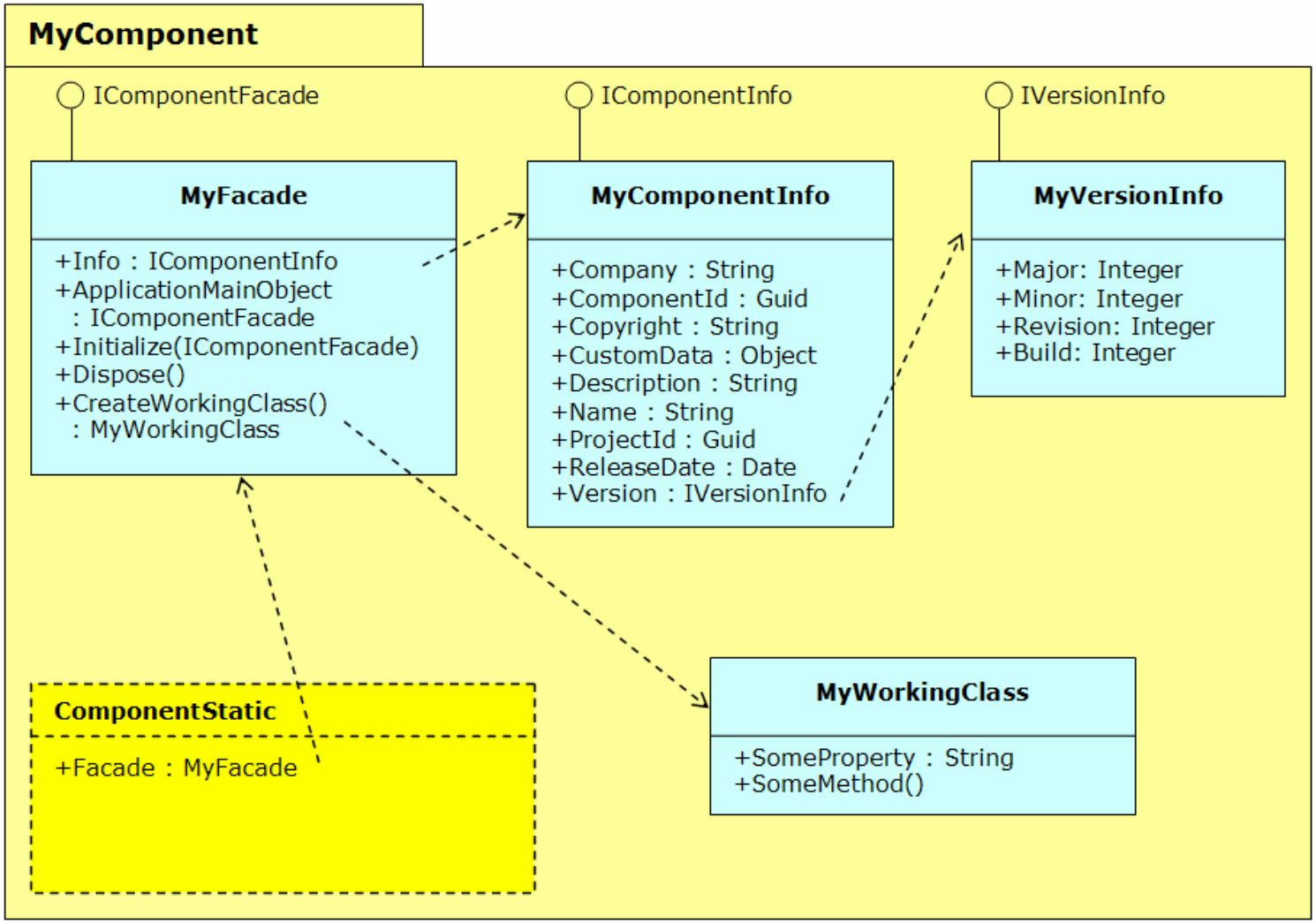
Vorschlag für eine modulare Anwendungsarchitektur
ComponentsBase – weitere Module

- ComponentStatic (Standardmodul)
  - Factory-Methode für Instanz des ComponentInfoFormatter
- Guid (Klassenmodul)
  - Repräsentiert Globally Unique Identifier
  - z.B. 419F7D87-A424-43B4-84AF-37706C7A583A
  - für ComponentId und ProjectId
- GuidStatic (Standardmodul)
  - Factory und Hilfsmethoden für Klasse Guid
- ComponentInfoFormatter (Klassenmodul)
  - Generiert String-Repräsentation des ComponentInfos

Vorschlag für eine modulare Anwendungsarchitektur
Entwickeln einer Komponente

- Vorgangsweise:
  - Verweis auf ComponentsBase setzen
  - Die drei Interfaces implementieren
    - `IComponentsFacade`
    - `IComponentsInfo`
    - `IVersionInfo`
  - In einem Standardmodul `ComponentStatic`
    - Eigenschaft `Facade As IComponentInfo` implementieren
    - liefert Referenz auf Singleton-Instanz der Fassade

Vorschlag für eine modulare Anwendungsarchitektur
Aufbau einer Komponente



Vorschlag für eine modulare Anwendungsarchitektur
Verwendung der Komponente in der Hauptanwendung

- Starter übergibt eine Referenz auf Fassade der Komponente an Fassade der Hauptanwendung
- Kontrollieren von
  - `IComponentInfo.ProjectId`
  - ev. `IComponentInfo.ComponentId`
  - ev. `IComponentInfo.Version`
- Speichern der Referenz in Feld
- Ev. Übergabe der Referenz der Hauptfassade an Eigenschaft der Komponenten-Fassade
  - Zirkuläre Referenz – Aufzulösen in `Dispose()`!

Uff – Fragen, Bemerkungen?

Danke für die Aufmerksamkeit!

Tools

- MZTools
  - www.mztools.com
  - „Must have“
  - Freeware für VB6/VBA
- Diverse AddIns von Thomas Möller
  - www.team-moeller.de
  - TM VBA-CodeInspector
  - TM VBA-ToDoFinder
  - TM-CodeDokumentation
  - TM-DatenKlassenGenerator
- mossSoft Proc Browser
  - http://www.access-im-unternehmen.de/374.0.html
- AddIns von Paul Rohorzka
  - Ev. im Laufe 2007/2008 unter www.softconcept.at zu finden???