

# Modulare Anwendungsentwicklung

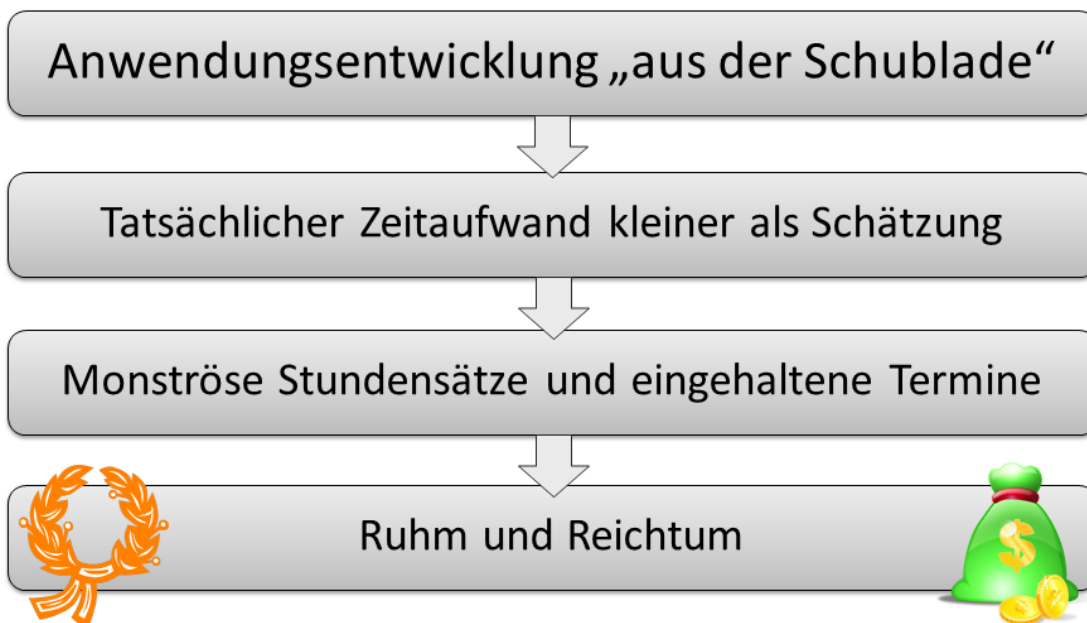
Referent: Michael Zimmermann

Zimmermann@SZWeb.de

## Übersicht

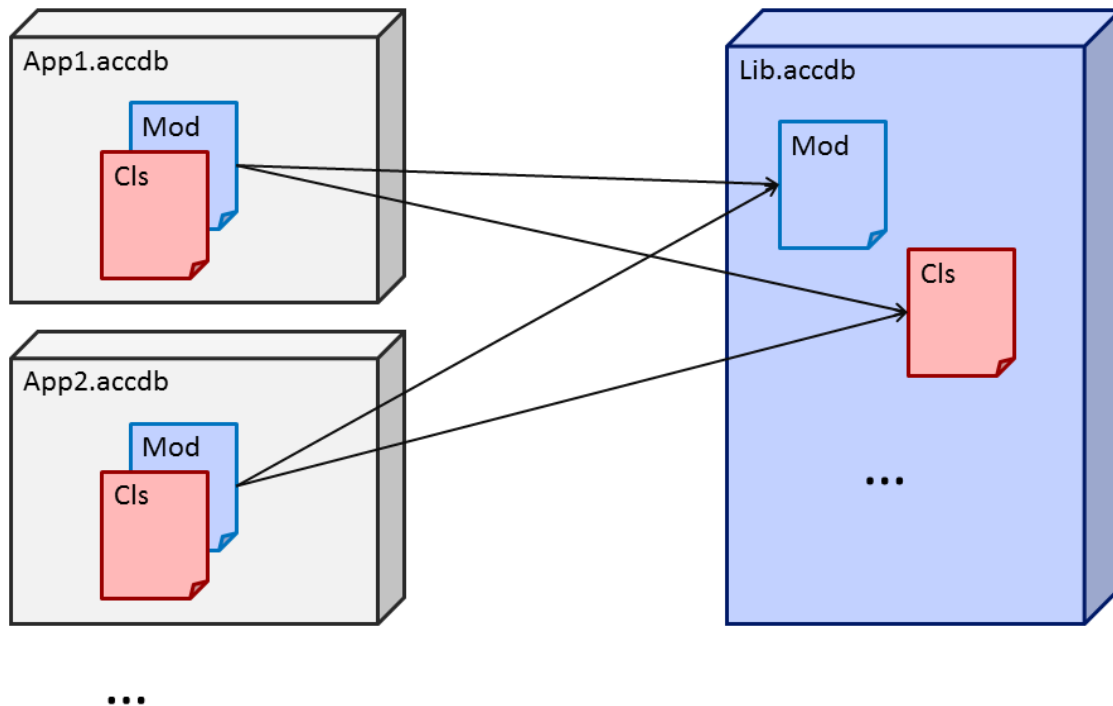
- Motivation
- Grundlagen
- Bibliotheksarten
- Code-Zugriff
- Formular-Handling
- Modulare Architektur
- Probleme/Einschränkungen

## Motivation



# Grundlagen

- Spezialfall von Zugriff über Dateigrenzen
- Zugriff auf **Quellcode** von „oben nach unten“
- Technische Realisierung: durch Verweis
- Es darf mehrere „Oben“ geben
- Kein direkter Zugriff von „unten nach oben“
  - Ausnahme: DoCmd.Openxxx (Bug?)
  - Aber: Organisatorisch gar nicht sinnvoll



## Anmerkungen:

DoCmd.OpenForm durchsucht die Verweise von oben nach unten, ob es ein dem Namen entsprechendes Objekt findet. So von der Library in der Application zugreifen zu wollen, hat den Nachteil, daß man beim Entwickeln der Bibliothek nicht weiß, in welchen Anwendungen sie eingebunden werden.

# Bibliotheksarten

- Technisch
  - Funktionsbibliothek
  - Klassenbibliothek
  - Formular-/Reportbibliothek (*mit eigener Oberfläche*)
  - Anwendungsbibliothek
- Organisatorisch
  - Standard
  - Optional

## Funktionsbibliothek

- Empfängt Parameter (Werte oder Objekte)
- Führt Aktionen durch
- Liefert Ergebnisse (Werte oder Objekte)
  - Objekttypen müssen in Anwendung und Bibliothek bekannt sein!
- Stellt **keine** instanzierbaren Klassen bereit
  - Objekte werden nur innerhalb der Bibliothek deklariert und erzeugt

## Klassenbibliothek

- Wie Funktionsbibliothek, aber zusätzlich instanzierbare Klassen
  - Objekte werden in der App deklariert **und erzeugt**
- Warum Klassen?
  - Anschaulichkeit, Intellisense
  - Mehrere Instanzen: Datenkapselung
  - Ereignisse (Auslösen und empfangen)
  - Übersichtliche Parametrisierung durch Eigenschaften
  - Modifikator Friend

## Formular-/Reportbibliothek

- Stellt (interaktive) Formulare und Berichte bereit
- Bibliotheksformulare können **nicht** als Unterformulare eingesetzt werden!

## Anwendungsbibliothek

- Faßt die bisherigen Möglichkeiten zusammen
- Eigene Daten
- Selbständig lauffähig
- Idealfall: Nach Verweiserstellung ist die Komponente ohne weiteren Aufwand verwendbar

### Anmerkungen:

Die Unterteilung ist rein begrifflich, da je nach Art der verwendeten Objekte andere (größere) Schwierigkeiten auftauchen.

Es gibt also keine technische Eigenschaft, die die Art der Bibliothek beschrieb.

**Funktionsbibliothek** meint, daß alle Zugriffe von außen und Rückgaben nach außen ausschließlich über Prozeduren in Standardmodulen erfolgen. Intern können dabei ruhig Klassen und Formulare verwendet werden. Solche Bibliotheken sind einfach und problemlos verwendbar.

**Klassenbibliothek** meint, daß die Bibliothek Klassenmodule enthält, die im Quellcode der verwendenden Anwendung deklariert und instanziiert<sup>1</sup>\* werden können. Das erfordert einige kleine Trickereien. Dazu mehr im nächsten Kapitel „**Code-Zugriff Bibliothek**“.

**Formularbibliothek** meint, daß von außen Formulare (Reports) aufgerufen werden können. Das bereitet je nach Vorgehen zusätzliche Probleme zu einfachen Klassenmodulen. Details im Kapitel „**Formularhandling**“.

**Anwendungsbibliothek** meint, daß alle obengenannten Möglichkeiten realisiert werden und zusätzlich noch Rückgriffe von der Bibliothek in die Anwendung möglich sein sollen, obwohl die Bibliothek gar nicht weiß, wie die Anwendung überhaupt beschaffen ist, und die Anwendungen sogar beliebig verschieden sein können. Dabei sollen Anwendungen und Bibliotheken sich auch beim Entfernen von Verweisen noch sinnvoll verhalten. Obendrein sollen sie ggf. eigene Datenquellen haben können. Das ist naturgemäß am schwierigsten zu verwirklichen. Hinweise dazu in den folgenden Kapiteln verstreut und konzentriert in den beiden **Modellen „Zugriff nach oben/unten“**.

**Standardbibliothek:** Damit ist eine Bibliothek gemeint, deren Funktionalitäten sinnvollerweise in jedem Projekt gebraucht werden: Datensicherung, Komprimierung, Dateneinbindung oder feingranulare Dinge wie Maximum/Minimum von Arrays, Kapselung häufig benötigter APIs etc. Bei der Verwendung des Codes solcher Bibliotheken in Anwendungen sind keine Maßnahmen erforderlich, die Sorge tragen für den Fall, daß die Bibliothek nicht verfügbar wäre.

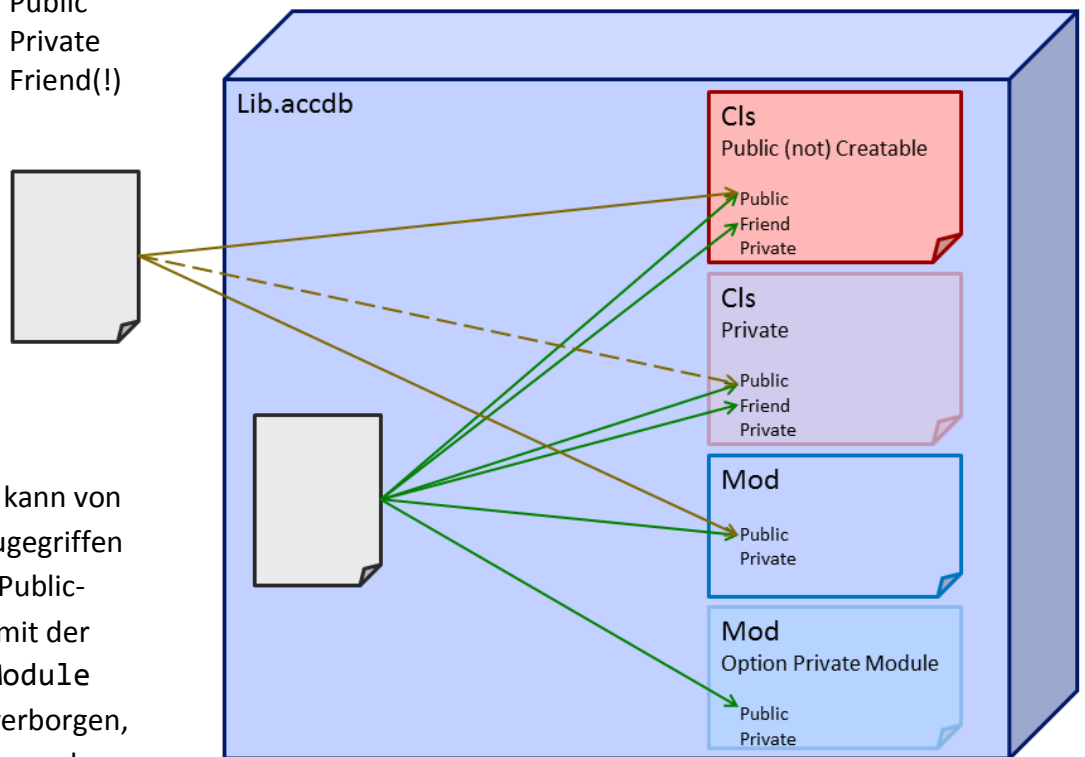
**Optionale Bibliothek:** Damit ist –ibs. Im Hinblick auf Partialanwendungen – gemeint, daß eine Bibliothek möglicherweise sogar in der gleichen Anwendung bei verschiedenen Installationen vorhanden oder nicht vorhanden sein kann. Dann sind im Code besondere Maßnahmen zu treffen, damit es nicht zu Kompilier- oder Laufzeitfehlern kommt, wenn z. B. ein Objekttyp oder eine Prozedur plötzlich nicht mehr vorhanden ist.

---

<sup>1</sup> Für alle Freunde von „instanziiert“: bilanziert, differenziert, finanziert, denunziert, distanziert, potenziert

# Code-Zugriff Bibliothek

- Offenlegen des Codes der Bibliothek
- Zugriffsmodifikatoren:
  - Klassen:
    - PublicCreatable
    - PublicNotCreatable
    - Private
  - Module:
    - Public (Default)
    - Option Private Module
  - Prozeduren:
    - Public
    - Private
    - Friend(!)



## Anmerkungen:

Auf **Standardmodule** kann von außen ganz normal zugegriffen werden, also auf alle Public-Prozeduren. Module mit der Option Private Module werden nach außen verborgen, sind aber von innen normal verwendbar.

**Klassenmodule** haben bei Erstellung standardmäßig eine Eigenschaft Instancing: Private (1). Damit sind sie nur innerhalb des Projekts verwendbar, aber nicht von außen. Damit eine Anwendung Klassen in einer Bibliothek verwenden kann, muß das Instancing auf PublicNotCreatable (2) oder besser auf PublicCreatable (5) gestellt sein.

Mit PublicNotCreatable kann eine äußere Anwendung immerhin ein `Dim k As Lib.Klasse` ausführen. Die Anweisung `Set k = New Lib.Klasse` scheitert aber. Hierzu müßte die Bibliothek eine Public Function in einem Standardmodul folgenden Inhalts bereitstellen:

```
Public Function NewKlasse() As Klasse
Dim tmp As Klasse
```

```
Set tmp = New Klasse  
Set NewKlasse = tmp  
End Function
```

Die Anwendung kann dann wie folgt vorgehen:

```
Dim k As Lib.Klasse  
Set k = Lib.NewKlasse
```

Die eigentlich wünschenswerte Einstellung PublicCreatable (5) läßt sich über die Oberfläche nicht einstellen. Wenn man sie erzwingt (ja, gleich), kann sie bei einem Import aller Module in eine neue leere Db oder bei Decompile zu (1) kaputtgerichtet werden.

Um sicherzugehen, daß die Klassen immer betriebsbereit sind, kann man beim Start eine Anweisung laufenlassen, die sie auf (5) setzt oder im produktiven Betrieb nicht mehr decompilen.

So jetzt aber endlich die Zauberformel:

```
Application.VBE.ActiveVBProject.VBComponents(„cls...“).Properties("Instancing").Value = 5
```

Statt ActiveProject kann auch VBProjects("Projektname") verwendet werden, um auf ein bestimmtes Projekt, also eine Bibliothek, zuzugreifen.

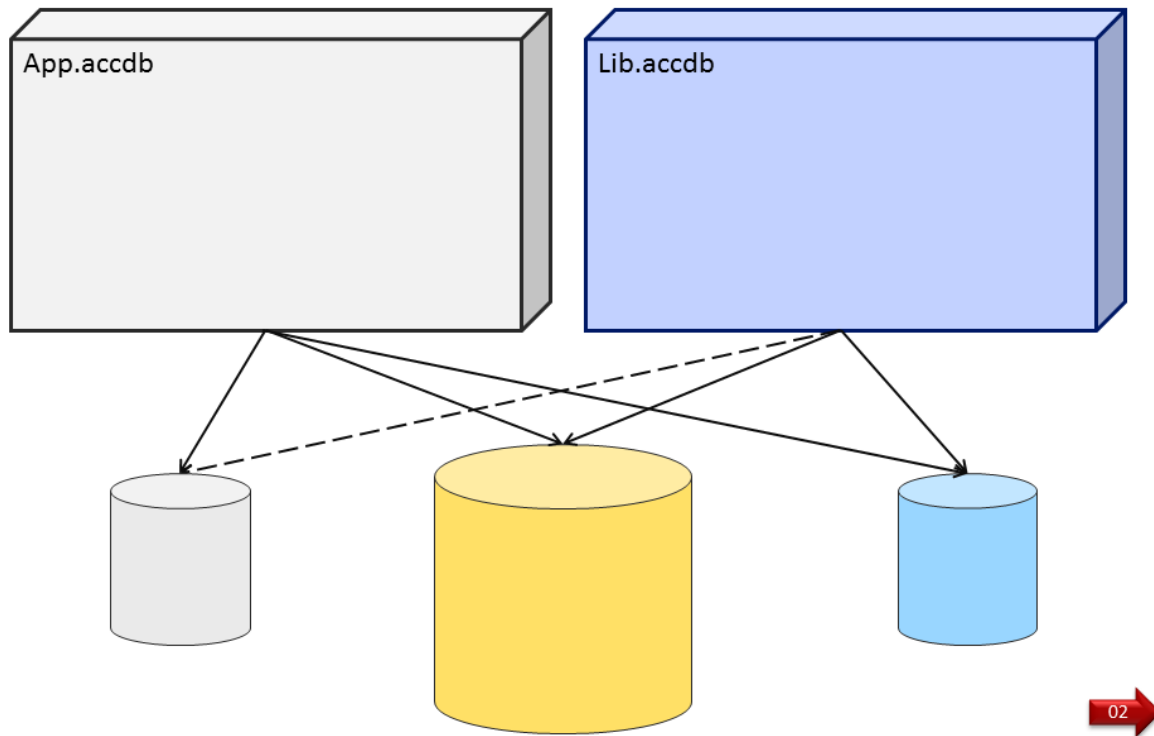
Wenn diese Eigenschaft gesetzt ist, stehen die betreffenden Klassen in der Bibliothek der Anwendung ganz normal zur Verfügung, können also deklariert und auch ohne Hilfsfunktion instanziiert werden.

Die Zugriffsmodifikatoren für **Prozeduren** Public und Private sind hinlänglich bekannt und verhalten sich in Bibliotheken nach innen und außen genauso, wie man es gewohnt ist.

Für den Fall, daß man in einer Klasse eine Prozedur (Sub-, Function-Methode, Property) in der Bibliothek öffentlich, also Public, haben möchte, nach außen für die Anwendung aber Private, also verborgen, kann man dafür den Modifikator Friend benutzen. Er ist nur in Klassen verwendbar.

# Datenzugriff

- Hat nichts mit Bibliotheken und Verweisen zu tun
- Grundsätzlich unproblematisch
  - Tabellenverknüpfungen
  - IN <external database>-Klausel
    - in Abfragen
    - in innerhalb VBA eingebettetem SQL
  - In VBA mittels OpenDatabase() statt CurrentDb



## Anmerkungen:

Dateiübergreifender Datenzugriff ist erfreulich problemlos.

Zum einen kann man in jedem Frontend (eine Bibliothek ist aus Datensicht nichts anderes) aus beliebigen unterschiedlichen Quellen Tabellen verknüpfen.

Mit der SQL-Klausel IN <External Datasource> hat man ein weiteres Instrument zur Verfügung. Man kann damit nicht nur auf Tabellen (also Backends) sondern auch auf Abfragen (also Frontends, d. i. andere Bibliotheken/Anwendungen) zugreifen. Um zum Beispiel eine Abfrage qryKunde unter einem beliebigem Namen – z. B. ebenfalls qryKunde oder abstrakter qryParentObject – in eine Bibliothek zu transportieren, erstellt man sie dort einfach mit folgendem SQL:

```
SELECT * FROM qryKunde IN „C:\Pfad\Anwendung.accdb“
```

Eine solche bibliothekslokale Abfrage kann dann als Datenquelle für Formulare, Kombi- und Listenfelder in der Bibliothek dienen. Man kann das SQL der Abfrage per VBA von der Anwendung aus manipulieren, um den Namen der Quellabfrage anzupassen, oder einfach mit sich selbst vereinbaren, daß jede Anwendung, die eine bestimmte Bibliothek nutzt, eine Abfrage bestimmten Namens, z. B. qryExportLibBeispielcboParent o. ä. verfügbar macht. Dann erübrigt sich das Anpassen bis auf den Pfad.

Umgekehrt kann man aus der Anwendung heraus auf Abfragen in Bibliotheken Bezug nehmen:

```
SELECT * FROM qryLibObjekt IN „C:\Pfad\Bibliothek.accdb“
```

Das Ganze funktioniert natürlich auch mit VBA-embedded SQL, z. B. um ein Recordset zu erstellen.

Dafür gibt es auch noch die Möglichkeit statt der gewohnten CurrentDb einfach ein eigenes DB-Objekt mit OpenDatabase zu öffnen.

```
Dim db as DAO.Database
Set db = DBEngine.OpenDatabase(„Pfad\Name.accdb“)
Set rcs = db.OpenRecordset(SQL)
...
rcs.close
db.close
Set rcs = Nothing
Set db = Nothing
```

SQL oder Tabellenname oder Abfragename bezieht sich dann auf so benannte Objekte in Pfad\Name.accdb und nicht auf möglicherweise gleichnamige Objekte in der CurrentDb.

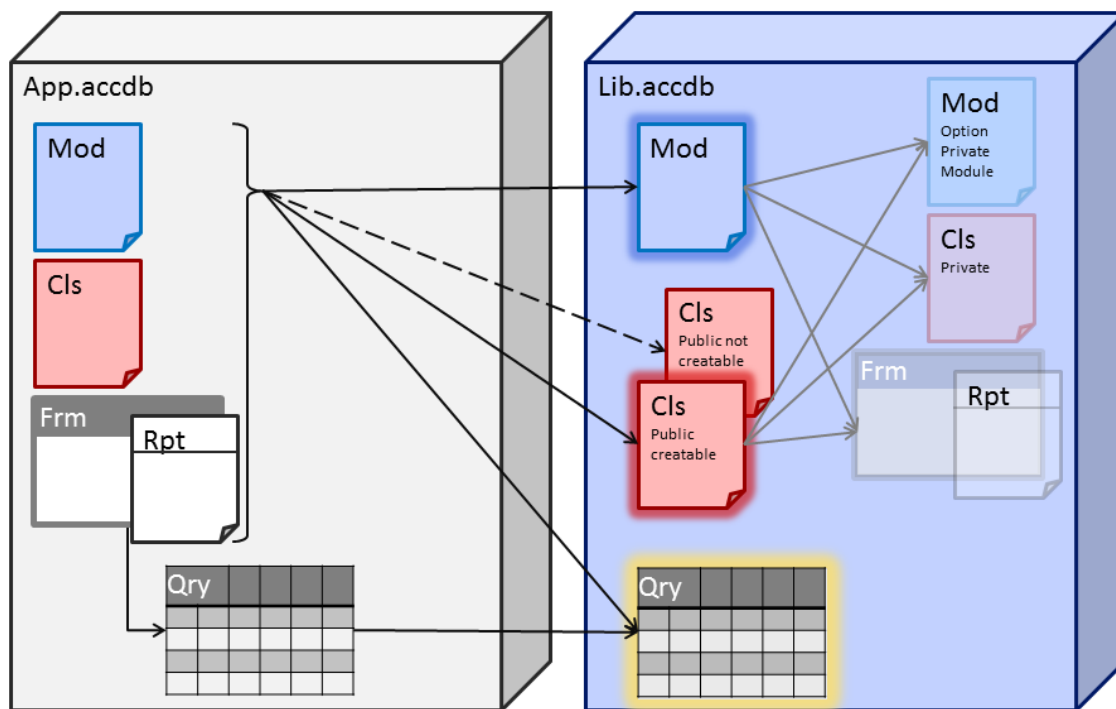
Als weitere Variante bietet sich ein gemeinsamer Datencontainer, was statt Jet auch ein SQL-Server sein kann, für alles an. Tabellen zu nicht eingebundenen Bibliotheken sind dann zwar da, bleiben aber einfach leer.

Die Problemlosigkeit von Datenzugriffen bezieht sich dabei sowohl auf die Richtung Anwendung > Bibliothek als auch umgekehrt. Auch an den Methoden ändert sich nichts.



# Zusammenfassung

- Alle App-Objekte können zugreifen
  - Uneingeschränkt auf Öffentliche Module
  - Eingeschränkt auf Öffentliche Klassen
    - Abhilfe:
      - Hilfsfunktion in Bibliothek
      - `Application.VBE.ActiveVBProject.VBComponents(„cls...“).Properties("Instantiating").Value = 5`
  - Uneingeschränkt auf Abfragen
  - Gar nicht auf Formulare/Reports
    - Abhilfe:
      - Hilfsfunktion, -klasse

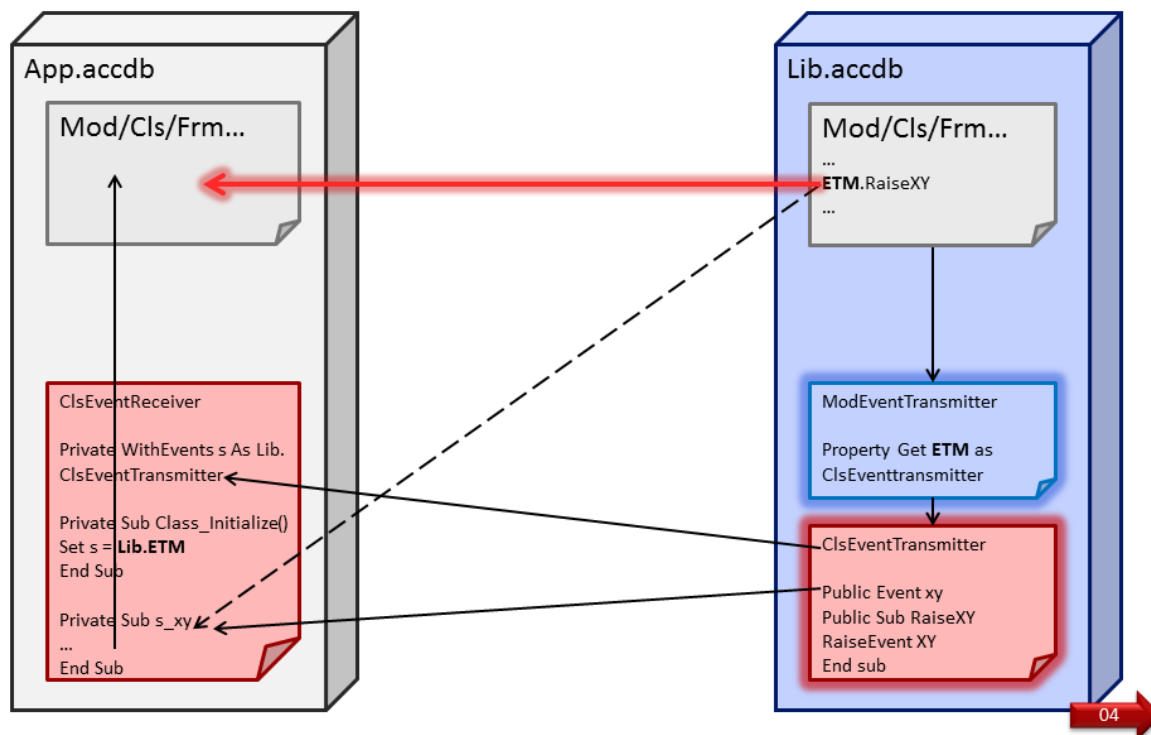


## Anmerkungen:

Diese Zusammenfassung bezieht sich nur auf die Richtung Anwendung greift auf Bibliothek zu. Der umgekehrte Weg wird nachfolgend beschrieben.

# Code-Zugriff Anwendung

- Zugriff „nach oben“ über Events oder Interfaces
  - Nur Code-Anforderung wird übertragen
  - Eigentlicher Code steht im Client
  - Kann daher Client-spezifisch sein
- Ereignissender (EventTransmitter) in Lib
- Ereignisempfänger (EventReceiver) in App
  - Auszuführender Code wird erst in App festgelegt → Gut so!
  - Hinweis: Events können Parameter haben



## Anmerkungen:

Hier geht es jetzt darum, daß der Code in einer Bibliothek Code in der verwendenden Anwendungen aufrufen können soll. Zunächst geht das nicht, d. h. die Klassen und Prozeduren der Anwendung sind der Bibliothek unbekannt. Man umgeht diese Beschränkung mit Klassenereignissen. Jede Klasse kann Ereignisse auslösen und jede andere Klasse kann Ereignisse von Klasseninstanzen empfangen, die in ihr WithEvents deklariert sind. Das funktioniert auch projektübergreifend.

Im Hinblick darauf, daß Bibliotheken in der gleichen Anwendung mal vorkommen sollen und mal nicht – weil nämlich ein Kunde die Faktura mit DTA und ein anderer sie ohne DTA bestellt hat – sollte man davon, daß beliebige Klassen Ereignisse erzeugen und beliebige Klassen diese empfangen können, keinen Gebrauch machen sondern die Schnittstelle zwischen Anwendung und Bibliothek möglichst schmal halten.

Konkret: Es gibt in der Bibliothek eine einzige Sender-Klasse und in der Anwendung eine einzige Empfänger-Klasse, die die Code-Anforderung einerseits einsammelt und andererseits verteilt.

Der EventTransmitter in der Bibliothek ist sehr simpel gestrickt:

```
clsEventTransmitter
```

```
Public Event Erstens
Public Event Zweitens(Text As String)
Public Event Drittens(Zahl1 As Long, Zahl2 As Double, Text As String)

Public Sub RaiseErstens: RaiseEvent Erstens: End Sub
Public Sub RaiseZweitens(Text As String): RaiseEvent Zweitens(Text): End Sub
Etc ...
```

Weiterer Code ist nicht erforderlich, wiewohl durchaus möglich. Man könnte das Erzeugen des Ereignisses ggf. noch von Bedingungen abhängig machen.

In einem Standardmodul mod EventTransmitter wird diese Klasse instanziiert, so daß alle Bibliotheksprozeduren, aber auch Anwendungsprozeduren darauf zugreifen können:

```
modEventTransmitter
```

```
Private m_ETM As clsEventTransmitter

Public Function ETM() As clsEventTransmitter
If m_ETM Is Nothing Then Set m_ETM = New clsEventTransmitter
Set ETM = m_ETM
End Function

Public Sub ETMClear()
Set m_ETM = Nothing
End Sub
```

Wenn jetzt irgendwo im Code der Bibliothek ein solches Ereignis ausgelöst werden soll, egal ob beim Klick auf eine Schaltfläche in einem Formular, bei einem bestimmten Ergebnis einer Function in einem Modul oder wo auch immer, wird einfach ein Aufruf angebracht wie z. B.:

```
ETM.RaiseZweitens „Hallo“
```

Dasselbe Ereignis kann natürlich an vielen Stellen erzeugt werden, aber z. B. unterschiedlich parametrisiert.

Die Bibliothek hat damit ihre Schuldigkeit getan, das Ereignis ist ausgelöst und kann – nicht muß – empfangen werden. Wenn es keiner empfängt passiert also nichts, es ist keine Anpassung nötig, wenn eine Anwendung ein bestimmtes Ereignis einfach nicht braucht. Dann konsumiert sie es eben einfach nicht.

Nun zum Empfänger. Jede Klasse, also auch Formulare und Reports, kann Ereignisse empfangen.

Wenn der Ereignissender eine obligatorische Standardbibliothek ist, spricht auch nichts dagegen. Wenn die Bibliothek optional ist, sollte man das aber nicht tun, sondern einen einzigen Empfänger erstellen.

Ein solcher Empfänger kann eine Klasse oder ein (unsichtbares) Formular sein.

```
clsEventReceiver
```

```
Private WithEvents ETMLib1 As Lib1.clsEventTransmitter
```

```
Private Sub Class_Initialize
```

```
Set ETMLib1 = Lib1.ETM 'Das ist die Function aus der Bibliothek!
```

```
End Sub
```

```
Private Sub ETMLib1_Erstens
```

```
...
```

```
End Sub
```

```
Private Sub ETMLib1_Zweitens(Text As String)
```

```
...
```

```
End Sub
```

```
Private Sub ETMLib1_Drittens(Zahl1 As Long, Zahl2 As Double, Text As string)
```

```
...
```

```
End Sub
```

Die Zeilen mit fettem Lib1 müssen für jede Bibliothek, die Ereignisse senden soll, eingefügt werden.

(Natürlich entsprechend mit ETMLib2 und dem passenden Lib-Namen.) In den Ereignisprozeduren könnte Nutzcode stehen; organisatorisch ist es übersichtlicher, wenn dieser in Modulen und Klassen der Anwendung steht, und die Prozeduren hier nur aufgerufen werden.

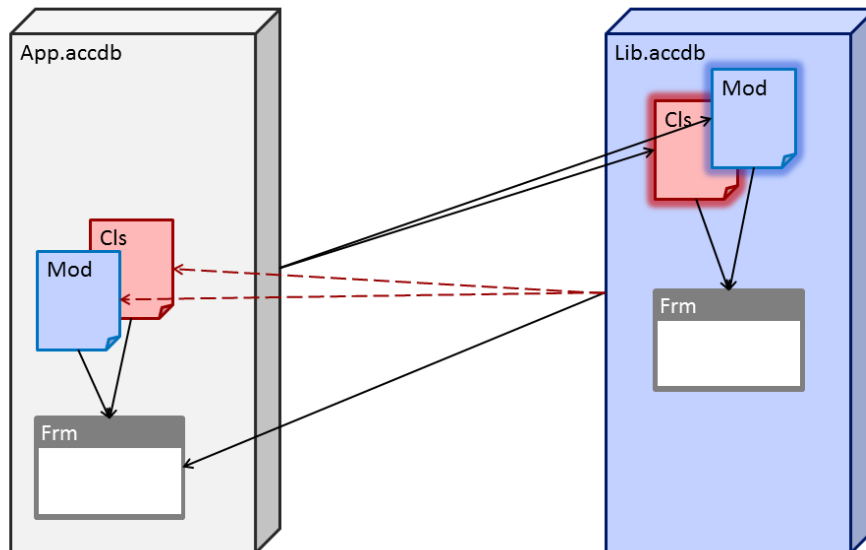
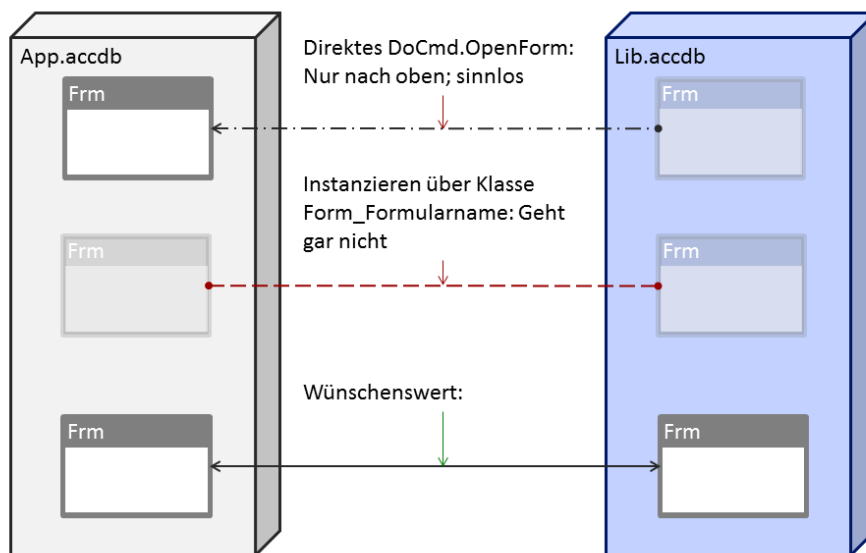
Beim Entfernen einer optionalen Klasse müssen die entsprechenden Zeilen auskommentiert werden, da es sonst Kompilierfehler gibt. Immerhin sind es nur zwei Zeilen.

Damit die Klasse Ereignisse empfängt, muß sie instanziiert sein. Mehr dazu bei den Modellen.

Theoretisch könnte man mit einem einzigen Ereignis je Bibliothek auskommen, wenn man einen festen Parameter (z. B. den ersten) dazu verwendet, festzulegen, was das Ereignis auslösen soll. Im Empfänger könnte das dann eine Select-Case-Struktur entscheiden. Mehrere Ereignisse sind aber m. E. übersichtlicher.

# Formularhandling

- Analog auch für Reports
- Formulare öffnen und schließen
  - DoCmd.OpenForm + DoCmd.Close
  - Set f = New Form\_FormName + Set f = Nothing
- Formular als Objekt per Code angreifen
  - Eigene Methoden und Eigenschaften aufrufen
    - Allgemeine Formularenderweiterungen, z. B. Navigation
    - Spezielle Erweiterungen, ibs.
      - Kapseln von Zugriff auf Steuerelemente
      - Ändern von Datenquellen



## Anmerkungen:

Von Formularen (entsprechend Reports) war bisher noch nicht die Rede. Es gibt zwei Möglichkeiten, Formulare aus Code heraus zu verwenden.

Man kann einerseits mit DoCmd.OpenForm "frmName" und DoCmd.Close arbeiten. Außer allergischen Reaktionen auf DoCmd spricht da wenig dagegen. Den Makel, daß man keinen Objektverweis auf das geöffnete Form bekommt, kann man leicht umgehen.

Vorteile: Das Formular ist in der Forms-Auflistung und kann so für Abfrageparameter verwendet werden. Das funktioniert auch, wenn das Form kein Modul enthält.

Nachteile: Man bekommt einen Objektverweis auf ein allgemeines Form-Objekt und nicht auf die Klasse Form\_frmName. Man hat also keinen Intellisense-Zugriff auf Steuerelemente und eigene Methoden der Form-Klasse. Wenn man's auswendig kann, geht's natürlich trotzdem.

Man kann andererseits auch mit Private f as Form\_frmName: Set f = New Form\_frmName und Set f = Nothing arbeiten.

Nachteile: Das geht nur mit Forms, die ein Modul enthalten. Das Form gelangt **nicht** in die Forms-Auflistung.

Vorteil: Man bekommt einen Objektverweis speziell auf die Klasse Form\_frmName. Man hat also auf deren spezielle Eigenschaft und Methoden Intellisense-Zugriff. Man kann mehrere Instanzen desselben Forms erzeugen.

Ein direkter Zugriff **von der Anwendung auf Formulare in einer Bibliothek** ist so jedoch nicht möglich.

DoCmd.OpenForm läßt nur Zugriffe auf lokale Formulare zu. Eine Angabe der Db ist nicht möglich.

Mit New Form\_frmName kommt man auch nicht weiter: Die Formularklassen verhalten sich standardmäßig wie Klassen mit Instancing: Private, sind also von außen nicht verwendbar. Im Gegensatz zu Klassenmodulen lassen sie sich aber auch durch Trickereien nicht auf ein anderes Verhalten manipulieren.

Um von der Anwendung in einer Bibliothek Formulare steuern zu können, muß also die Bibliothek selbst in Klassen oder Modulen eine Zugriffsmöglichkeit bereitstellen.

Im einfachsten Fall genügt eine öffentliche Prozedur OpenForm in einem Modul:

```
Public Function OpenForm (frmName As String) As Access.Form
DoCmd.OpenForm frmName
Set OpenForm = Forms(frmName)
End Function
```

Meistens kommt man damit aus, wenngleich die oben genannten Nachteile bestehen. Immerhin gelangt ein so geöffnetes Formular in die Forms-Auflistung der Hauptanwendung und man erhält, wenn nötig,

einen Verweis auf das geöffnete Form-Objekt. Man könnte natürlich auch die Methode `New Form_frmName` so kapseln, aber da nur eine Rückgabe als `Access.Form` möglich ist – `Form_frmName` in der Bibliothek ist `Private`, also in der Anwendung nicht bekannt! –, hat man dadurch keinen Vorteil.

Wenn man auch auf spezifische Eigenschaften und Methoden von Formularen per Intellisense zugreifen will, wird's kompliziert. Folgendes ist eine Möglichkeit:

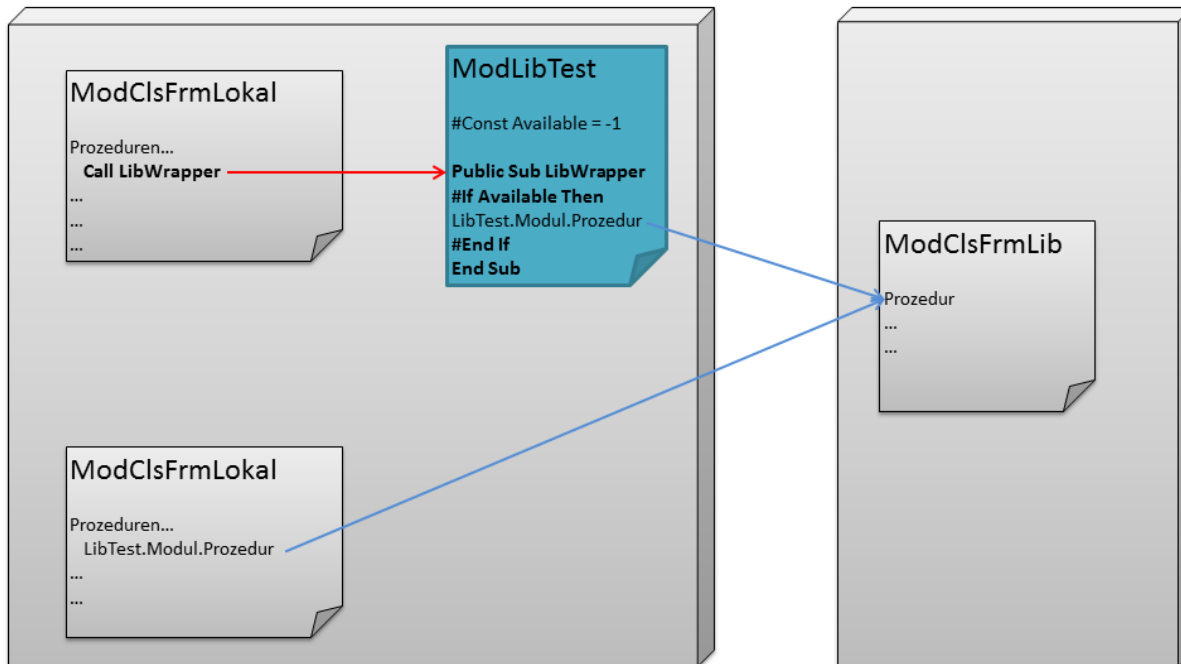
Man lagert alle Prozeduren, die man standardmäßig in jedem Formular zusätzlich haben möchte – z. B. eine ID-Property und Code zur Navigation zu bestimmten Datensätzen, Statusinformationen, allgemeine Datensatzoperationen – in eine eigene Klasse aus. Diese Klasse bekommt eine Name-Eigenschaft und eine `Open`-Methode spendiert, die das Öffnen der Formulare nach dem Muster von `OpenForm` weiter oben übernimmt. Im Gegensatz zum Form-Modul kann dieses Klassenmodul in einer Anwendung verwendet werden.

Wer es ganz arg treiben will, kann zusätzlich zu jedem Formular, das über spezielle Eigenschaften verfügen soll, je eine Klasse anlegen, die dies kapselt. Im Form-Modul wäre dann eine Property vom Rückgabetyt dieser Erweiterungsklasse anzulegen. Letzteres ist mit einigem Aufwand verbunden, der den Rahmen dieser kleinen Einführung sprengen würde, weshalb darauf nicht weiter eingegangen werden soll.

Der umgekehrte Weg, **von der Bibliothek in die Anwendung**, wäre mit `DoCmd.OpenForm` durchaus möglich. Wesentlich besser ist aber, auch das mit den schon beschriebenen Ereignissen zu handhaben: Wenn ein Klick in der Bibliothek ein Formular öffnen soll, steht das eben im passenden Ereignis-Handler in der Anwendung – eine andere Anwendung macht an dieser Stelle vielleicht ein ganz anderes Formular auf.

# Modell: Zugriff nach unten

App führt Code in Lib aus



## Anmerkungen:

Der einfachste Weg, Code in einer Bibliothek auszuführen, ist, die entsprechenden Prozeduren einfach im Anwendungscode aufzurufen: Lib.Modul.Prozedur, wobei Lib und Modul optional, aber empfehlenswert sind. Entsprechend Dim x As Lib.Klasse: Set x = New Lib.Klasse (im Bild der untere Ast).

Bei Standardbibliotheken spricht nichts dagegen. Bei optionalen Bibliotheken können beim Entfernen Kompilier- oder Laufzeitfehler quer durch die ganze Anwendung entstehen. Dem kann man entgegenwirken, indem man die Aufrufe konzentriert und kapselt (im Bild der obere Ast).

Für jede optionale Bibliothek legt man ein Modul modLibXXX an, das Aufrufe für alle gewünschten Prozeduren und Klassen der Bibliothek enthält. Im normalen Anwendungscode steht dann statt z. B. Libxxx.Modul.TestSub der Aufruf WrapLibxxxModulTestSub, der wie folgt definiert ist:

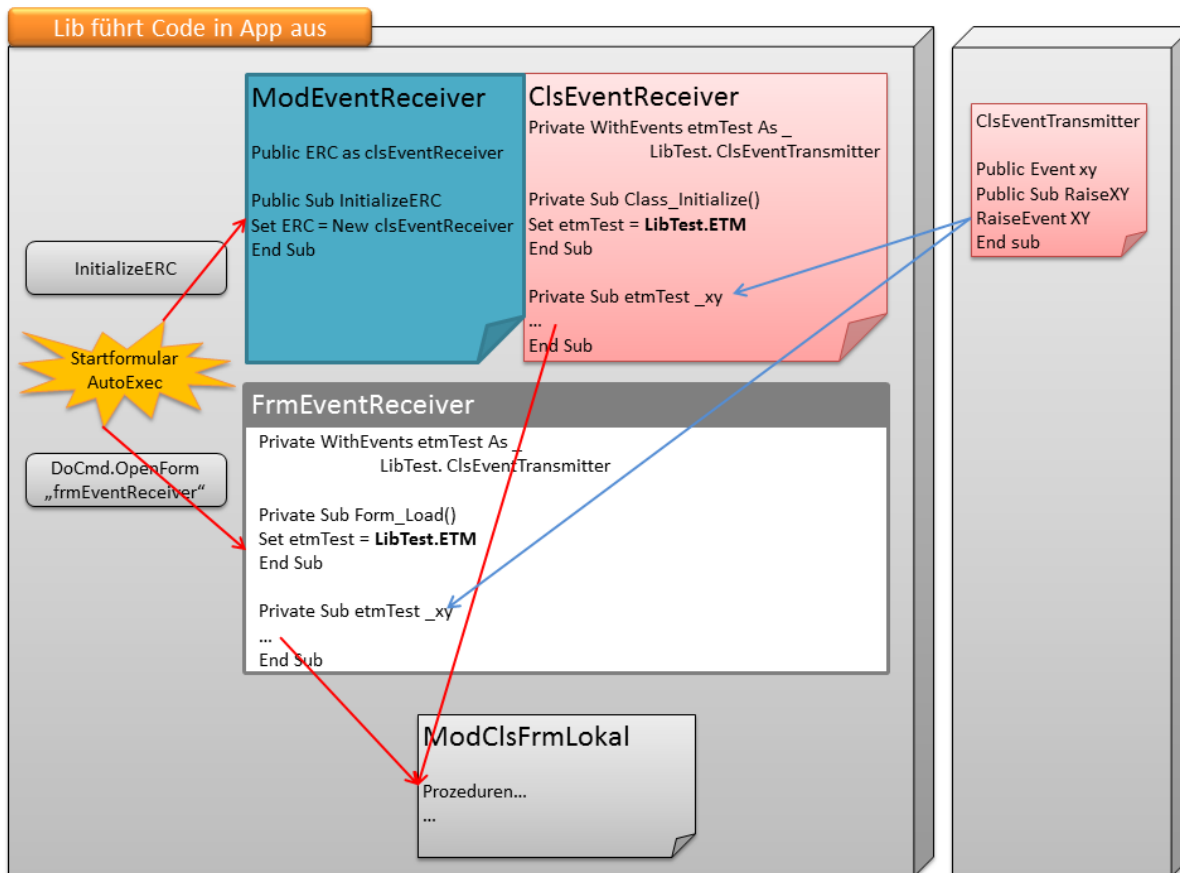
```
#Const Available=-1 '0
```

```
Public Sub WrapLibxxxModulTestSub()  
#If Available Then  
Libxxx.Modul.TestSub  
#End If  
End Sub
```



Wenn man LibXXX aus dem Projekt entfernt, muß man nur die Compiler-Konstante Available auf 0 setzen, und der Code läßt sich weiterhin fehlerfrei betreiben. Beim Aufruf von WrapLibxxxModulTestSub passiert dann einfach gar nichts. Alternativ kann man in einem Zweig #Else noch Ersatzcode unterbringen, z. B. einen Hinweis, daß die Funktion in dieser Version nicht verfügbar ist ö. ä.

# Modell: Zugriff nach oben



## Anmerkungen:

Um den vorhin beschriebenen Event-Receiver zu erzeugen, gibt es zwei Wege. Wenn der Receiver eine Klasse ist, braucht man eine Instanzvariable von dieser Klasse und Code, der die Instanzvariable füllt. Dazu kann man in einem Standardmodul folgendes anlegen:

```
Public ERC as clsEventReceiver

Public Sub InitializeERC()
Set ERC = New clsEventReceiver
End Sub
```

Die Prozedur muß dann beim Anwendungsstart im Form\_Load des Startformulars oder in einem Auto\_Exec-Makro aufgerufen werden.

Alternativ kann man auf die Klasse verzichten und statt dessen ein Formular verwenden. Der Code in diesem frmEventReceiver ist mit dem in der entsprechenden Klasse präzise identisch bis auf die Prozedur Form\_Load anstelle von Class\_Initialize. Ein Hilfsmodul wie eben braucht man dann nicht, beim Anwendungsstart wird statt InitializeERC einfach DoCmd.OpenForm " frmEventReceiver" aufgerufen.

Man kann das Formular unsichtbar machen oder einfach auf Höhe und Breite 0 stellen und an die Position (-1, -1) schieben. Welche der beiden Varianten man wählt, ist im Prinzip wurscht.

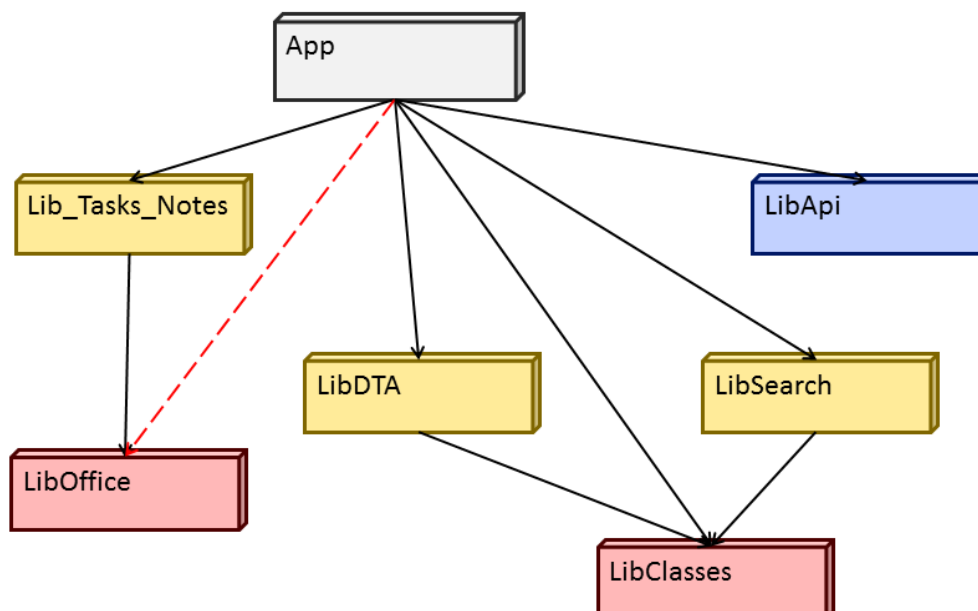
# Modulare Architektur

- Hauptanwendung
  - Spezifische Daten
  - Bibliotheksverwaltung
- Teilanwendungen
  - Allgemeine Daten und Funktionen
  - Sub-Bibliotheksverwaltung
- Datenbankaufbau
  - Verteilt oder konzentriert
  - Dehnfähige Normalisierungsmoral erforderlich

## Ideen für Bibliotheken

- Modul allgemeine Hilfsfunktionen
- Modul Office-Handling
- Modul Mathematische Erweiterungen
- Modul Verbesserte Objekttypen
- Modul Performance-Manager
- Modul API-Sammlung
- Modul Protokollierung
- Teilanwendung Ähnlichkeitssuche
- Teilanwendung Aufgabenverwaltung
- Teilanwendung DTA/Online-Banking
- Teilanwendung Handheld/Scanner-Anbindung
- Modul Geo-Daten/Karten

## Szenariobeispiel



# Probleme

- Abfrageparameter Forms(„Formularname“)...
  - ... funktionieren nicht bei New Form\_xxx
  - ... aber bei DoCmd.OpenForm auch in Bibliothek
- Zirkelreferenzen
- Verweise (Verlieren und Hängenbleiben)
- Kaputtreparieren von Klassenmodulen
- Current[Db|Project] vs. Code[Db|Project]
- Ufo-Beschränkungen
- Namenskonflikte – Präfixe existieren!

Zirkelreferenzen zwischen Objekten in Anwendung und Bibliothek oder innerhalb einer Bibliothek sind unbedingt zu vermeiden. Es kann sonst passieren, daß Access sich nicht mehr beenden läßt.

Bei häufigen Problemen kann man in Betracht ziehen, Verweise per Code beim Anwendungsstart zu setzen und vor dem Ende zu entfernen. Wenn man ordentlich programmiert sollte das aber nicht nötig sein.

Wie eingangs erwähnt können beim Decompile und beim Import in eine neue DB Klassenmodule mit Instancing 5 kaputtrepariert werden.

CurrentDb ist immer die Anwendungsdatei. Wenn man sich auf die Datei beziehen will, in der gerade der Code läuft, muß man statt dessen CodeDb nehmen. Man soll in einer Bibliothek grundsätzlich immer CodeDb statt CurrentDb verwenden, außer man will ausdrücklich auf die Anwendung zugreifen.

Analoges gilt für die wenigen ADOisten bezüglich CurrentProject und CodeProject.

Formulare einer Bibliothek können nicht als Unterformulare in Formularen der Anwendung verwendet werden. Man kann allenfalls mit viel Positionierungscode und rahmenlosen Forms drum herum programmieren.

Um Namenskonflikte zu vermeiden, sollte man intensiv – nein, besser exzessiv – von Präfixen Gebrauch machen. Außerdem verhindert man so, daß man bei Verwendung vieler Bibliotheken irgendwann aus dem Fenster springt. Insbesondere muß man jedem VBA-Projekt einen sinnvollen Namen geben, da diese standardmäßig alle "Database" heißen.

# Appendix

## AEK 10

- Thomas Möller  
Verweise
- Paul Rohorzka:  
Modulare Anwendungsentwicklung

## Fazit

Wenn man sich auf Funktionsbibliotheken im ursprünglichen Sinn beschränkt, also die Schnittstelle nur aus Aufrufe über öffentliche Prozeduren in Standardmodulen erfolgt (die intern durchaus Klassen und Formulare nutzen können), gibt es eigentlich keine nennenswerten Probleme. Diese Verwendung von Bibliotheken ist uneingeschränkt empfehlenswert.

Bei allem, was darüber hinausgeht, sollte man ein wenig Lust am Abenteuer zumindest während der Entwicklung mitbringen. Wenn's dann aber mal läuft, hängt die Symbolleiste der IDE voller Geigen und der Code quietscht vor Vergnügen beim Kompilieren.