Sehr solide Grundidee 🤝 – ZIP entpacken, XML parsen, `sharedStrings` nutzen, Zellen nach `r`-Referenz (A1, B2 …) einsortieren. Für viele einfache XLSX klappt das. Für „robust in freier Wildbahn" fehlen aber ein paar kritische Details.

Kurzfazit

- ✅ Good: keine Excel-Abhängigkeit, klare Struktur, funktioniert für einfache Zahlen/SharedStrings.
- ⚠️ Fixen: **Sheet-Datei-Mapping**, **Inline-Strings**, **Datumsformate (Styles)**, **Formeln/Errors**, **Rich-Text**, **Temp-Ordner/Parallelität**.
- 🧠 Optional: Streaming/SAX für große Dateien, Iterator-API statt alles in RAM.

Wichtigste Korrekturen

1. **Sheet-Datei richtig auflösen (r:id statt sheetId)**
   `workbook.xml` listet `<sheet name="..." r:id="rIdX">`. Die **Datei** steht in `xl/_rels/workbook.xml.rels` unter dem passenden `Id` → `Target="worksheets/sheetN.xml"`.
   Dein Code nutzt `sheetId` → das ist **nicht** zuverlässig die Dateinummer.

2. **Inline-Strings unterstützen**
   Zellen können `t="inlineStr"` haben oder `<is><t>…</t></is>` enthalten. Aktuell liest du nur `<v>` + `t="s"` (SharedStrings).

3. **Datums-/Zahlformat per styles.xml**
   Excel speichert **Daten als Zahl** (Seriennummer). Ob es ein Datum ist, steht am **Style** (`s`-Attribut → `cellXfs/xf/@numFmtId`).

- Built-ins: 14, 15, 16, 17, 22 (u.a.) sind Datum/Uhrzeit.
- Custom: `numFmts/numFmt/@formatCode` mit `d`, `m`, `y`, `h`, `s`.
  Ohne styles-Parsing werden Daten als Floats ausgegeben.

4. **Rich-Text in SharedStrings**
   `<si>` kann mehrere `<r>`-Runs enthalten; du holst aktuell nur `si/t`. Für viele Dateien reicht das, bei formatiertem Text fehlt Inhalt.

5. **Formeln & Fehler**
   `<f>` (Formel) + `<v>` (Resultat) bzw. `t="e"` (Error). Aktuell ignoriert.

6. **Temp-Ordner & Parallelität**
   Fester Ordner `tmpxls\` neben der EXE → Kollisionen bei Mehrfachaufruf/Parallelbetrieb. Besser **pro Lauf** ein eindeutiges Unterverzeichnis (z. B. `hb_MilliSeconds()` oder GUID). Danach **sicher** löschen.

7. **Kleinigkeiten/Bugs**

- `aGetFiles`-Check: `IF Len(aDocs) == 0` statt `If Empty(Len(aDocs))`.
- Typos in `Get()`: `AlLTrim` → `AllTrim`, `End` → `EndIf`.
- `CleanupTemp()`: sicherstellen, dass nur **unterhalb** deines Temp-Basispfades gelöscht wird (Pfad-Traversal vermeiden).
- `ProcessCellValue` sollte zusätzlich den **Style-Index** kennen (siehe unten).

Schnelle Fixes (Diff-Stil, minimal)

1) Temp-Verzeichnis pro Lauf

```
-METHOD CreateTempDirs() CLASS TExcelXlsxReader
-    LOCAL aDirs := { "tmpxls", ... }
+METHOD CreateTempDirs() CLASS TExcelXlsxReader
+    LOCAL cBase := cFilePath(GetModuleFileName(GetInstance())) + "tmpxls\"
+    LOCAL cRun  := hb_NumToHex( hb_RandomInt(0, 0x7FFFFFFF) )
+    ::cTempDir  := cBase + cRun + "\"
+    hb_DirBuild( ::cTempDir )
+    hb_DirBuild( ::cTempDir + "_rels\" )
+    hb_DirBuild( ::cTempDir + "docProps\" )
+    hb_DirBuild( ::cTempDir + "xl\_rels\" )
+    hb_DirBuild( ::cTempDir + "xl\worksheets\" )
     ...
```

2) Workbook-Rels auflösen (r:id → Target)

```
-METHOD ProcessWorkbook(hXml) CLASS TExcelXlsxReader
-    ... // nutzt sheetId
+METHOD ProcessWorkbook(hXml) CLASS TExcelXlsxReader
+    LOCAL hRels := XMLToHash( ::cTempDir + "xl\_rels\workbook.xml.rels" )
+    LOCAL hMap  := {=>}  // rId -> Target
+    LOCAL aRel, i
+    IF hRels != NIL .AND. hb_HHasKey(hRels, "Relationships") .AND. hb_HHasKey(hRels["Relationships"], "Relationship")
+       aRel := hRels["Relationships"]["Relationship"]
+       IF !HB_ISARRAY(aRel); aRel := { aRel }; ENDIF
+       FOR i := 1 TO Len(aRel)
+          IF hb_HHasKey(aRel[i], "attributes")
```

```
+             hMap[ aRel[i]["attributes"]["Id"] ] := aRel[i]["attributes"]["Target"]  // e.g. "worksheets/sheet1.xml"
+          ENDIF
+       NEXT
+    ENDIF
+    ::aSheetNames := {}
+    ::hWorkbook := hXml
+    IF hb_HHasKey(hXml, "workbook") .AND. hb_HHasKey(hXml["workbook"], "sheets")
+       LOCAL aSheet := hXml["workbook"]["sheets"]["sheet"]
+       IF !HB_ISARRAY(aSheet); aSheet := { aSheet }; ENDIF
+       FOR i := 1 TO Len(aSheet)
+          LOCAL attrs := aSheet[i]["attributes"]
+          LOCAL cName := attrs["name"]
+          LOCAL cRid  := attrs["r:id"]
+          LOCAL cTarget := hMap[ cRid ]
+          IF !Empty(cTarget)
+             AAdd(::aSheetNames, IF(Empty(cName), "Sheet" + LTrim(Str(i)), cName))
+             // Merke den konkreten Dateipfad je Index:
+             IF ::aSheets == NIL; ::aSheets := {}; ENDIF
+             AAdd(::aSheets, { => "path" => ::cTempDir + "xl\" + cTarget } )
+          ENDIF
+       NEXT
+    ENDIF
 RETURN NIL
```

3) Inline-String & Style mitverarbeiten

```
-METHOD ProcessSheetData(hXml, nSheet) CLASS TExcelXlsxReader
+METHOD ProcessSheetData(hXml, nSheet) CLASS TExcelXlsxReader
     ...
-                        IF hb_HHasKey(oCell, "v")
+                        // Style-Index (für Datumsprüfung)
+                        LOCAL nStyle := NIL
+                        IF hb_HHasKey(oCell, "attributes") .AND. hb_HHasKey(oCell["attributes"], "s")
+                           nStyle := Val(oCell["attributes"]["s"])
+                        ENDIF
+
+                        IF hb_HHasKey(oCell, "is")  // inlineStr
+                           LOCAL oIs := oCell["is"]
+                           LOCAL cText := ""
+                           IF hb_HHasKey(oIs, "t")
+                              cText := HB_ISHASH(oIs["t"]) .AND. hb_HHasKey(oIs["t"], "value") ;
+                                       ? oIs["t"]["value"] : oIs["t"]
+                           ENDIF
+                           xValue := cText
+                        ELSEIF hb_HHasKey(oCell, "v")
                             ...
-                            xValue := ::ProcessCellValue(cValue, cType)
+                            xValue := ::ProcessCellValue(cValue, cType, nStyle)
                         ENDIF
```

4) styles.xml lesen + Datum erkennen

```
-    PROTECTED:
+    PROTECTED:
         ...
+        METHOD LoadStyles()
+        DATA hStyles AS HASH INIT NIL
```

Aufruf in `New()`:

```
-::LoadSharedStrings()
-::LoadWorkbook()
+::LoadSharedStrings()
+::LoadStyles()
+::LoadWorkbook()
```

Implementierung (Kurzform):

```
METHOD LoadStyles() CLASS TExcelXlsxReader
   LOCAL cFile := ::cTempDir + "xl\styles.xml", hXml
   IF File(cFile)
      hXml := XMLToHash(cFile)
      ::hStyles := hXml
   ENDIF
RETURN Self
```

Erweiterte Datumserkennung:

```
STATIC FUNCTION IsDateStyle( self, nStyle )
   LOCAL h := self:hStyles, nFmtId := NIL, aXf, fmt, i
   IF h == NIL; RETURN .F.; ENDIF
   // cellXfs/xf[nStyle+1]/@numFmtId
   IF hb_HHasKey(h,"styleSheet") .AND. hb_HHasKey(h["styleSheet"],"cellXfs")
      aXf := h["styleSheet"]["cellXfs"]["xf"]
      IF !HB_ISARRAY(aXf); aXf := { aXf }; ENDIF
      IF nStyle+1 >= 1 .AND. nStyle+1 <= Len(aXf) .AND. hb_HHasKey(aXf[nStyle+1],"attributes")
         nFmtId := Val( aXf[nStyle+1]["attributes"]["numFmtId"] )
      ENDIF
   ENDIF
   IF nFmtId == NIL; RETURN .F.; ENDIF
   IF nFmtId $ {14,15,16,17,22, 45,46,47}; RETURN .T.; ENDIF
```

```
   // custom numFmts check for d/m/y
   IF hb_HHasKey(h["styleSheet"],"numFmts") .AND. hb_HHasKey(h["styleSheet"]["numFmts"],"numFmt")
      fmt := h["styleSheet"]["numFmts"]["numFmt"]
      IF !HB_ISARRAY(fmt); fmt := { fmt }; ENDIF
      FOR i := 1 TO Len(fmt)
         IF Val(fmt[i]["attributes"]["numFmtId"]) == nFmtId
            LOCAL code := Upper(fmt[i]["attributes"]["formatCode"])
            RETURN ( "D" $ code .OR. "M" $ code .OR. "Y" $ code )
         ENDIF
      NEXT
   ENDIF
RETURN .F.
```

`ProcessCellValue` anpassen:

```
-METHOD ProcessCellValue(cValue, cType) CLASS TExcelXlsxReader
+METHOD ProcessCellValue(cValue, cType, nStyle) CLASS TExcelXlsxReader
    LOCAL xResult := ""
    DO CASE
       CASE cType == "s"
          xResult := ::GetSharedString(Val(cValue))
-      CASE cType == "n" .OR. Empty(cType)
-         xResult := Val(cValue)
+      CASE cType == "n" .OR. Empty(cType)
+         IF nStyle != NIL .AND. IsDateStyle( Self, nStyle )
+            xResult := ExcelSerialToHarbourDate( Val(cValue) )
+         ELSE
+            xResult := Val(cValue)
+         ENDIF
       CASE cType == "b"
          xResult := (cValue == "1")
       OTHERWISE
          xResult := cValue
    ENDCASE
RETURN xResult
```

Excel-Seriennummer → Date:

```
FUNCTION ExcelSerialToHarbourDate( n )
   // Excel 1900-System; Achtung: fiktiver 29.02.1900 Bug – pragmatisch ignorieren
   LOCAL d := CToD("1899-12-30")  // Basis für Harbour
RETURN d + Int(n)
```

5) Kleinere Korrekturen

```
- If Empty(Len(aDocs))
+ IF Len(aDocs) == 0

- xValue := AlLTrim(Str(xValue))
+ xValue := AllTrim(Str(xValue))

- End
+ EndIf
```

Edge-Cases, die du (später) abdecken kannst

- **Merged Cells** (`mergeCells`) → leere Zellen übernehmen den Wert der Merge-Ankerzelle.
- **Spaltenbreiten/Hidden** → i. d. R. ignorierbar für reinen Datenimport.
- **Error-Zellen** (`t="e"`) → sinnvoll als `NIL` oder eigener Marker.
- **Formeln** (`<f>`) → optional Flag `:ReturnFormula()`.

Könnte man das ohne eine Klasse funktional nicht einfacher und für die User, die mitentwickeln möchten, transparenter machen

---

Absolutely. Here's a **transparent, procedural ("no class")** version that keeps everything in plain functions and simple data structures (hashes & arrays). It unzips the XLSX, resolves sheet paths via `_rels`, loads `sharedStrings.xml` and `styles.xml`, then lets you iterate sheet cells with a callback. Easy to read, hack, and extend.

Drop-in: `xlsx_reader.prg` (procedural, minimal)

```
/*  xlsx_reader.prg  – functional XLSX reader (no classes)
    Goals:
      - no Excel installed
      - few moving parts, easy to hack
      - supports: sharedStrings, inlineStr, basic date detection via styles
    Notes:
      - Uses HB_UNZIPFILE and TXmlDocument (FiveWin)
      - Good for "data import" cases; not a full OOXML engine
*/

#include "FiveWin.ch"
```

```
// ---------- Public API ----------

// XLSX_Load()  -> returns a hash { "tmp", "shared", "styles", "sheets" }
FUNCTION XLSX_Load( cXlsxFile )
   LOCAL h := {=>}, cTmp, hRels, hWb, aMap, i, aSheets, cRid, cTarget, cName

   XLSX_Check( File( cXlsxFile ), "File not found: " + cXlsxFile )

   cTmp := XLSX_MakeTemp()
   XLSX_Check( HB_UNZIPFILE( cXlsxFile, {|| .T.}, .T., NIL, cTmp, NIL ), "Unzip failed" )

   // core files
   XLSX_Check( File( cTmp + "xl\workbook.xml" ), "Missing xl\\workbook.xml" )
   // optional files (sharedStrings, styles) — load if present
   h["tmp"]    := cTmp
   h["shared"] := IF( File( cTmp + "xl\sharedStrings.xml" ),  XLSX_LoadShared( cTmp + "xl\sharedStrings.xml" ), {} )
   h["styles"] := IF( File( cTmp + "xl\styles.xml" ),         XLSX_LoadStyles( cTmp + "xl\styles.xml" ),        NIL )

   // resolve sheet paths via workbook.rels (r:id -> Target)
   hRels := XLSX_XMLToHash( cTmp + "xl\_rels\workbook.xml.rels" )
   aMap  := XLSX_RelsMap( hRels ) // { "rId1" => "worksheets/sheet1.xml", ... }

   hWb   := XLSX_XMLToHash( cTmp + "xl\workbook.xml" )
   aSheets := {}
   IF HB_ISHASH( hWb ) .AND. hb_HHasKey( hWb, "workbook" ) .AND. hb_HHasKey( hWb["workbook"], "sheets" )
      LOCAL a := hWb["workbook"]["sheets"]["sheet"]
      IF ! HB_ISARRAY( a ); a := { a }; ENDIF
      FOR i := 1 TO Len( a )
         cName := a[i]["attributes"]["name"]
         cRid  := a[i]["attributes"]["r:id"]
         cTarget := aMap[ cRid ]
         IF !Empty( cTarget )
            AAdd( aSheets, { ;
              "name" => cName, ;
              "path" => cTmp + "xl\" + cTarget ;
            } )
         ENDIF
      NEXT
   ENDIF
   h["sheets"] := aSheets

RETURN h

// Iterate all cells of a sheet and call a block {|nRow,nCol,xValue| ... }
// Returns number of visited cells
FUNCTION XLSX_ForEachCell( hBook, nSheet, bEach )
   LOCAL cSheet, hXml, nVisited := 0

   XLSX_Check( nSheet >= 1 .AND. nSheet <= Len( hBook["sheets"] ), "Sheet index out of range" )
   XLSX_Check( HB_ISEVALITEM( bEach ), "Callback block required" )

   cSheet := hBook["sheets"][ nSheet ]["path"]
   XLSX_Check( File( cSheet ), "Sheet file missing: " + cSheet )

   hXml := XLSX_XMLToHash( cSheet )
   IF HB_ISHASH( hXml ) .AND. hb_HHasKey( hXml, "worksheet" ) .AND. hb_HHasKey( hXml["worksheet"], "sheetData" )
      LOCAL sd := hXml["worksheet"]["sheetData"]
      LOCAL aRows := sd["row"]
      IF ! HB_ISARRAY( aRows ); aRows := { aRows }; ENDIF

      LOCAL i, oRow, nRow, aCells, j, oCell, cRef, cType, nCol, xVal, nStyle, cVal
      FOR i := 1 TO Len( aRows )
         oRow := aRows[i]
         nRow := IF( HB_ISHASH(oRow) .AND. hb_HHasKey(oRow,"attributes") .AND. hb_HHasKey(oRow["attributes"],"r"), ;
                     Val( oRow["attributes"]["r"] ), i )

         aCells := oRow["c"]
         IF Empty( aCells ); LOOP; ENDIF
         IF ! HB_ISARRAY( aCells ); aCells := { aCells }; ENDIF

         FOR j := 1 TO Len( aCells )
            oCell := aCells[j]
            IF ! HB_ISHASH( oCell ); LOOP; ENDIF

            cRef  := IF( hb_HHasKey(oCell,"attributes") .AND. hb_HHasKey(oCell["attributes"],"r"), oCell["attributes"]["r"], "" )
            cType := IF( hb_HHasKey(oCell,"attributes") .AND. hb_HHasKey(oCell["attributes"],"t"), oCell["attributes"]["t"], "" )
            nCol  := XLSX_ColFromRef( cRef )
            nStyle:= IF( hb_HHasKey(oCell,"attributes") .AND. hb_HHasKey(oCell["attributes"],"s"), Val(oCell["attributes"]["s"]), NIL )

            // value source: inlineStr vs <v>
            IF hb_HHasKey( oCell, "is" )
               xVal := XLSX_ReadInlineStr( oCell["is"] )
            ELSE
               cVal := XLSX_ReadV( oCell )
               xVal := XLSX_ProcessValue( cVal, cType, nStyle, hBook["styles"], hBook["shared"] )
            ENDIF

            nVisited++
            Eval( bEach, nRow, nCol, xVal )
         NEXT
      NEXT
   ENDIF
```

```
RETURN nVisited

// Convenience: read cell value by A1
FUNCTION XLSX_GetA1( hBook, nSheet, cA1 )
   LOCAL r := XLSX_RowFromRef( cA1 )
   LOCAL c := XLSX_ColFromRef( cA1 )
   LOCAL xFound := NIL
   XLSX_ForEachCell( hBook, nSheet, {|nr,nc,x| IF( nr==r .AND. nc==c, ( xFound := x ), NIL ) } )
RETURN xFound

// Cleanup temp dir
FUNCTION XLSX_Close( hBook )
   IF HB_ISHASH( hBook ) .AND. hb_HHasKey( hBook, "tmp" )
      IF ! Empty( hBook["tmp"] ) .AND. lIsDir( hBook["tmp"] )
         hb_DirRemoveAll( hBook["tmp"] )
      ENDIF
   ENDIF
RETURN NIL

// ---------- Helpers (parsing) ----------

STATIC FUNCTION XLSX_MakeTemp()
   LOCAL cBase := cFilePath( GetModuleFileName( GetInstance() ) ) + "tmpxls\"
   LOCAL cRun  := hb_NumToHex( hb_RandomInt(0,0x7FFFFFFF) )
   LOCAL cDir  := cBase + cRun + "\"
   hb_DirBuild( cDir )
   hb_DirBuild( cDir + "xl\_rels\" )
   hb_DirBuild( cDir + "xl\worksheets\" )
RETURN cDir

STATIC FUNCTION XLSX_RelsMap( hRels )
   LOCAL h := {=>}
   IF HB_ISHASH( hRels ) .AND. hb_HHasKey( hRels, "Relationships" ) .AND. hb_HHasKey( hRels["Relationships"], "Relationship" )
      LOCAL a := hRels["Relationships"]["Relationship"], i
      IF ! HB_ISARRAY( a ); a := { a }; ENDIF
      FOR i := 1 TO Len( a )
         IF hb_HHasKey( a[i], "attributes" )
            h[ a[i]["attributes"]["Id"] ] := a[i]["attributes"]["Target"]
         ENDIF
      NEXT
   ENDIF
RETURN h

STATIC FUNCTION XLSX_LoadShared( cFile )
   LOCAL h := XLSX_XMLToHash( cFile ), a := {}
   IF HB_ISHASH(h) .AND. hb_HHasKey(h,"sst") .AND. hb_HHasKey(h["sst"],"si")
      LOCAL si := h["sst"]["si"], i
      IF ! HB_ISARRAY( si ); si := { si }; ENDIF
      FOR i := 1 TO Len( si )
         AAdd( a, XLSX_SI_ToText( si[i] ) )
      NEXT
   ENDIF
RETURN a

STATIC FUNCTION XLSX_SI_ToText( oSi )
   // handle <si><t>... or <si><r>...</r><r>...</r>
   IF hb_HHasKey( oSi, "t" )
      RETURN XLSX_ExtractT( oSi["t"] )
   ENDIF
   IF hb_HHasKey( oSi, "r" )
      LOCAL a := oSi["r"], i, c:=""
      IF ! HB_ISARRAY( a ); a := { a }; ENDIF
      FOR i := 1 TO Len( a )
         IF hb_HHasKey( a[i], "t" ); c += XLSX_ExtractT( a[i]["t"] ); ENDIF
      NEXT
      RETURN c
   ENDIF
RETURN ""

STATIC FUNCTION XLSX_ExtractT( oT )
   RETURN IF( HB_ISHASH(oT) .AND. hb_HHasKey(oT,"value"), oT["value"], ;
           IF( HB_ISSTRING(oT), oT, "" ) )

STATIC FUNCTION XLSX_LoadStyles( cFile )
   RETURN XLSX_XMLToHash( cFile ) // we'll query it when needed

STATIC FUNCTION XLSX_ReadInlineStr( oIs )
   IF hb_HHasKey( oIs, "t" )
      RETURN XLSX_ExtractT( oIs["t"] )
   ENDIF
RETURN ""

STATIC FUNCTION XLSX_ReadV( oCell )
   IF hb_HHasKey( oCell, "v" )
      IF HB_ISHASH( oCell["v"] ) .AND. hb_HHasKey( oCell["v"], "value" )
         RETURN oCell["v"]["value"]
      ELSEIF HB_ISSTRING( oCell["v"] )
         RETURN oCell["v"]
      ENDIF
   ENDIF
RETURN ""

// Convert raw cVal using type, style and dictionaries
STATIC FUNCTION XLSX_ProcessValue( cVal, cType, nStyle, hStyles, aShared )
```

```
   LOCAL x

   DO CASE
   CASE cType == "s"  // shared string
      x := XLSX_GetShared( aShared, Val( cVal ) )

   CASE cType == "b"  // boolean
      x := ( cVal == "1" )

   CASE Empty( cType ) .OR. cType == "n" // number/date
      x := Val( cVal )
      IF nStyle != NIL .AND. XLSX_IsDateStyle( hStyles, nStyle )
         x := XLSX_ExcelSerialToDate( x )
      ENDIF

   OTHERWISE // inline or unknown
      x := cVal
   ENDCASE

RETURN x

STATIC FUNCTION XLSX_GetShared( aShared, nIdx )
   nIdx++ // 0-based in XLSX
   IF nIdx >= 1 .AND. nIdx <= Len( aShared )
      RETURN aShared[ nIdx ]
   ENDIF
RETURN ""

// Style helpers
STATIC FUNCTION XLSX_IsDateStyle( hStyles, nStyle )
   LOCAL nFmtId := NIL

   IF hStyles == NIL
      RETURN .F.
   ENDIF

   // cellXfs/xf[nStyle+1]/@numFmtId
   IF hb_HHasKey(hStyles,"styleSheet") .AND. hb_HHasKey(hStyles["styleSheet"],"cellXfs")
      LOCAL aXf := hStyles["styleSheet"]["cellXfs"]["xf"]
      IF ! HB_ISARRAY( aXf ); aXf := { aXf }; ENDIF
      IF nStyle + 1 >= 1 .AND. nStyle + 1 <= Len( aXf ) .AND. hb_HHasKey( aXf[nStyle+1], "attributes" )
         nFmtId := Val( aXf[nStyle+1]["attributes"]["numFmtId"] )
      ENDIF
   ENDIF

   IF nFmtId == NIL
      RETURN .F.
   ENDIF

   // common built-ins (14,15,16,17,22,45-47)
   IF nFmtId $ {14,15,16,17,22,45,46,47}
      RETURN .T.
   ENDIF

   // custom numFmts: look for d/m/y in formatCode
   IF hb_HHasKey(hStyles["styleSheet"],"numFmts") .AND. hb_HHasKey(hStyles["styleSheet"]["numFmts"],"numFmt")
      LOCAL aFmt := hStyles["styleSheet"]["numFmts"]["numFmt"], i, c
      IF ! HB_ISARRAY( aFmt ); aFmt := { aFmt }; ENDIF
      FOR i := 1 TO Len( aFmt )
         IF Val( aFmt[i]["attributes"]["numFmtId"] ) == nFmtId
            c := Upper( aFmt[i]["attributes"]["formatCode"] )
            RETURN ( "D" $ c .OR. "M" $ c .OR. "Y" $ c )
         ENDIF
      NEXT
   ENDIF

RETURN .F.

STATIC FUNCTION XLSX_ExcelSerialToDate( n )
   // Excel 1900 date system (pragmatic, ignores 1900-02-29 bug)
   LOCAL dBase := CToD( "1899-12-30" )
RETURN dBase + Int( n )

// ---------- Small utilities ----------

STATIC FUNCTION XLSX_ColFromRef( cRef )
   LOCAL cCol := "", i, n := 0
   FOR i := 1 TO Len( cRef )
      IF IsAlpha( SubStr( cRef, i, 1 ) )
         cCol += SubStr( cRef, i, 1 )
      ELSE
         EXIT
      ENDIF
   NEXT
   FOR i := 1 TO Len( cCol )
      n := n * 26 + ( Asc( Upper( SubStr( cCol, i, 1 ) ) ) - Asc( "A" ) + 1 )
   NEXT
RETURN n

STATIC FUNCTION XLSX_RowFromRef( cRef )
   LOCAL i, c := ""
   FOR i := 1 TO Len( cRef )
      IF IsDigit( SubStr( cRef, i, 1 ) )
         c += SubStr( cRef, i, 1 )
```

```
      ENDIF
   NEXT
RETURN Val( c )

STATIC FUNCTION XLSX_XMLToHash( cXMLFile )
   LOCAL oDoc, h := hash()
   IF ! File( cXMLFile ); RETURN h; ENDIF
   TRY
      oDoc := TXmlDocument():New( cXMLFile )
   CATCH
      RETURN h
   END
   IF oDoc:oRoot == NIL
      RETURN h
   ENDIF
   XLSX_ProcessNode( oDoc:oRoot:oChild, @h )
RETURN h

STATIC FUNCTION XLSX_ProcessNode( oNode, hOut )
   LOCAL hNode, xVal
   DO WHILE oNode != NIL
      xVal := IF( !Empty(oNode:cData), oNode:cData, "" )
      IF !Empty( oNode:aAttributes ) .OR. oNode:oChild != NIL
         hNode := hash()
         hb_HSet( hNode, "value", xVal )
         IF !Empty( oNode:aAttributes )
            hb_HSet( hNode, "attributes", hash() )
            HEval( oNode:aAttributes, {|k,v| hb_HSet( hNode["attributes"], k, v ) } )
         ENDIF
         IF oNode:oChild != NIL
            XLSX_ProcessNode( oNode:oChild, @hNode )
         ENDIF
         xVal := hNode
      ENDIF
      IF hb_HHasKey( hOut, oNode:cName )
         IF ! HB_ISARRAY( hOut[ oNode:cName ] )
            hOut[ oNode:cName ] := { hOut[ oNode:cName ] }
         ENDIF
         AAdd( hOut[ oNode:cName ], xVal )
      ELSE
         hb_HSet( hOut, oNode:cName, xVal )
      ENDIF
      oNode := oNode:oNext
   ENDDO
RETURN NIL

STATIC PROCEDURE XLSX_Check( lCond, cMsg )
   IF ! lCond
      BREAK cMsg
   ENDIF
RETURN
```

How to use (example)

```
FUNCTION Main()
   LOCAL cXlsx := "C:\temp\data.xlsx"
   LOCAL hBook, n

   hBook := XLSX_Load( cXlsx )

   // List sheets
   AEval( hBook["sheets"], {|s,idx| ? idx, ":", s["name"] } )

   // Iterate first sheet and print row;col;value
   n := XLSX_ForEachCell( hBook, 1, {|r,c,v| QOut( LTrim(Str(r)) + ";" + LTrim(Str(c)) + ";" + hb_ValToExp(v) ) } )
   ? "Cells visited:", n

   // Read a single A1 value
   ? "A1:", XLSX_GetA1( hBook, 1, "A1" )

   XLSX_Close( hBook )
RETURN NIL
```

Why this is “simpler & transparent”

- **No class state**: the “workbook” is just a hash you can inspect (`? hb_ValToExp( hBook )`).
- **Tiny, composable functions**: unzip → map rels → load shared → load styles → iterate cells.
- **Callback iteration**: contributors can plug import logic right in the `bEach` block (e.g., buffer rows and bulk insert).
- **Safe temp handling**: per-run folder, wiped in `XLSX_Close()`.

Known limits (kept on purpose for simplicity)

- Merged cells aren’t expanded (value only on the anchor cell).

- Error cells (`t="e"`) and formulas (`<f>`) are surfaced as their stored result (if present); evaluation isn't implemented.
- Full rich text is concatenated (formatting ignored).
- For very large sheets, TXmlDocument loads the whole XML; if you need huge files, we can switch the sheet reader to a simple SAX/stream approach later.

If you want, I can add a **CSV writer** (`XLSX_SaveAsCSV(hBook, nSheet, cPath)`) or a **PDO/ADO importer** that uses the `bEach` callback to build rows and bulk-insert—still purely functional.

Versteh ich total. Für viele Mitentwickler ist **funktional** (ein Satz klarer Funktionen + einfache Datenstrukturen) viel zugänglicher als Klassen/OOP-Magie.

Hier ist ein pragmatischer Fahrplan, der in Teams super funktioniert:

Warum Funktionen hier besser sind

- **Lesbarkeit:** Jede Funktion macht genau eine Sache, hat Ein-/Ausgabe – fertig.
- **Onboarding:** Neue Leute sehen „Daten rein → Daten raus" statt Vererbung/Overrides.
- **Testbarkeit:** Unit-Tests pro Funktion sind trivial.
- **Erweiterbarkeit:** Über **Callbacks** oder **Pipelines** statt Subklassen.

Minimal-Architektur (Daten-first)

- Datencontainer als **Hash/Array** (z. B. `hBook`).
- Ein schmales, dokumentiertes **Public API**:
  - `XLSX_Load( cFile ) -> hBook`
  - `XLSX_ForEachCell( hBook, nSheet, bEach )`
  - `XLSX_GetA1( hBook, nSheet, cA1 )`
  - `XLSX_Close( hBook )`
- Intern: kleine Helfer (`XLSX_LoadShared`, `XLSX_LoadStyles`, …). Kein globaler Zustand.

Erweiterungen ohne OOP

1) Pipeline-Callbacks

So binden andere ihre Logik ein, ohne Code zu „vererben":

```
// Validierung + Transform + Insert als Funktionen
FUNCTION ValidateCell( nRow, nCol, xVal ) ; RETURN .T.
FUNCTION TransformCell( nRow, nCol, xVal ) ; RETURN xVal
FUNCTION InsertRow( aRow ) ; RETURN NIL

// Sammeln pro Zeile:
LOCAL aRow := {}
XLSX_ForEachCell( hBook, 1, {|r,c,v|
   IF ValidateCell(r,c,v)
      v := TransformCell(r,c,v)
      aRow[ c ] := v
      IF c == nLastCol // Ende der Zeile bekannt/ausgelesen
         InsertRow( aRow )
         aRow := {}
      ENDIF
   ENDIF
})
```

2) Konfiguration statt Vererbung

```
LOCAL hCfg := {=>}
hCfg["sheet"] := 1
hCfg["columns"] := { "NAME" => "A", "PLZ" => "B", "ORT" => "C" }
hCfg["onRow"] := {|hRow| MyInsert(hRow) }  // callback

XLSX_ImportWithMap( hBook, hCfg )
```

Jeder versteht „Mapping + Callback", niemand muss eine Klasse überschreiben.

Team-Regeln (kurz & wirksam)

- **Datei-Schnitt:** `xlsx_reader.prg` = nur Lesen; `xlsx_import.prg` = Importregeln; kein Cross-Import.
- **Namensschema:** `XLSX_*` für Public, `XLSX_*_*` für intern.
- **Keine implizite Magie:** alles über Parameter; keine versteckten Globals.
- **Ergebnis-Typ:** immer `{ "ok"=>.T./.F., "err"=>cMsg, "data"=>... }` oder klar definierter Return.
- **Logging:** `XLOG( level, msg )` als einfacher Hook (austauschbar).
- **Tests:** Mini-Datensätze/Beispiel-.xlsx im `tests/` Ordner; Scripts zeigen Ende-zu-Ende (Load → Iterate → CSV/SQL).

Drop-in-Beispiele (kurz)

A) „Row-Mapping“-Import ohne OOP

```
FUNCTION XLSX_ImportWithMap( hBook, hCfg )
   LOCAL nSheet := hCfg["sheet"], hMap := hCfg["columns"], hRow := {=>}
   LOCAL aCols := hb_HKeys( hMap ), i, nColMax := Len( aCols )
   LOCAL aColNums := Array( nColMax )
   FOR i := 1 TO nColMax
      aColNums[i] := XLSX_ColFromRef( hMap[ aCols[i] ] + "1" ) // „A“→1 usw.
   NEXT

   LOCAL nCurRow := 0
   XLSX_ForEachCell( hBook, nSheet, {|r,c,v|
      IF r != nCurRow .AND. nCurRow > 0
         Eval( hCfg["onRow"], hRow )  // Übergabe an User-Callback
         hRow := {=>}
      ENDIF
      nCurRow := r
      FOR i := 1 TO nColMax
         IF c == aColNums[i]
            hRow[ aCols[i] ] := v
            EXIT
         ENDIF
      NEXT
   })
   IF Len( hRow ) > 0
      Eval( hCfg["onRow"], hRow )
   ENDIF
RETURN { "ok"=>.T. }
```

B) CSV-Export als einfache Brücke

```
FUNCTION XLSX_SaveAsCSV( hBook, nSheet, cOut )
   LOCAL h := FCreate( cOut ), r := 0, cLine := "", nMaxCol := 0
   IF h < 0 ; RETURN { "ok"=>.F., "err"=>"cannot create " + cOut } ; ENDIF
   // optional: nMaxCol bestimmen
   XLSX_ForEachCell( hBook, nSheet, {|nr,nc,x| nMaxCol := Max(nMaxCol, nc) } )
   // zeilenweise schreiben
   LOCAL aRow := Array( nMaxCol )
   AEval( aRow, {|_,i| aRow[i] := "" } )
   XLSX_ForEachCell( hBook, nSheet, {|nr,nc,x|
      IF nr != r .AND. r > 0
         cLine := XLSX_JoinCSV( aRow )
         FWrite( h, cLine + hb_eol() )
         AEval( aRow, {|_,i| aRow[i] := "" } )
      ENDIF
      r := nr
      aRow[ nc ] := XLSX_CsvEscape( x )
   })
   IF r > 0
      cLine := XLSX_JoinCSV( aRow )
      FWrite( h, cLine + hb_eol() )
   ENDIF
   FClose( h )
RETURN { "ok"=>.T., "data"=>cOut }

STATIC FUNCTION XLSX_JoinCSV( aRow )
   LOCAL i, c := ""
   FOR i := 1 TO Len(aRow)
      IF i > 1 ; c += "," ; ENDIF
      c += aRow[i]
   NEXT
RETURN c

STATIC FUNCTION XLSX_CsvEscape( x )
   LOCAL c := hb_ValToStr( x )
   IF '"' $ c .OR. "," $ c .OR. hb_eol() $ c
      c := '"' + StrTran( c, '"', '""' ) + '"'
   ENDIF
RETURN c
```

Fazit

Ja – **funktional** ist hier absolut der bessere Default: verständlicher, leichter zu warten, und kollisionsarm in der Zusammenarbeit. Die oben skizzierten Bausteine reichen für 90 % der Praxisfälle (XLSX lesen, validieren, mappen, CSV/SQL). Wer später „tiefer“ will (Merged Cells, riesige Dateien, Styles), erweitert gezielt einzelne **Funktionen**, nicht eine Klasse mit verstecktem Zustand.

Wenn du magst, passe ich dir das **procedural XLSX-Reader-Snippet** aus unserem letzten Turn sofort auf deine Ordnerstruktur an (Dateinamen, Logging, Insert-Callback).