Application Note

# Using COM in AutomationDesk

Release 2021-A – May 2021

## How to Contact dSPACE

| | |
|---|---|
| Mail: | dSPACE GmbH<br>Rathenaustraße 26<br>33102 Paderborn<br>Germany |
| Tel.: | +49 5251 1638-0 |
| Fax: | +49 5251 16198-0 |
| E-mail: | info@dspace.de |
| Web: | http://www.dspace.com |

## How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: http://www.dspace.com/go/locations
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.
  Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: http://www.dspace.com/go/supportrequest. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

## Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/go/patches for software updates and patches.

## Important Notice

# Contents

# About This Document

**Content**

This document introduces features that let you access other applications in AutomationDesk sequences via Microsoft's Component Object Model (COM) .

**Required knowledge**

Working with AutomationDesk requires:

- Basic knowledge in handling the PC and the Microsoft Windows operating system.
- Basic knowledge in developing applications or tests.
- Basic knowledge in handling the external device, which you control remotely via AutomationDesk.

dSPACE provides trainings for AutomationDesk. For more information, refer to https://www.dspace.com/go/trainings.

**Symbols**

dSPACE user documentation uses the following symbols:

| Symbol | Description |
|---|---|
| ⚠ DANGER | Indicates a hazardous situation that, if not avoided, will result in death or serious injury. |
| ⚠ WARNING | Indicates a hazardous situation that, if not avoided, could result in death or serious injury. |
| ⚠ CAUTION | Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury. |
| NOTICE | Indicates a hazard that, if not avoided, could result in property damage. |
| Note | Indicates important information that you should take into account to avoid malfunctions. |
| Tip | Indicates tips that can make your work easier. |
| ⍰ | Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise. |

| Symbol | Description |
|---|---|
| 📖 | Precedes the document title in a link that refers to another document. |

**Naming conventions**

dSPACE user documentation uses the following naming conventions:

**%name%** Names enclosed in percent signs refer to environment variables for file and path names.

**< >** Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

**Special folders**

Some software products use the following special folders:

**Common Program Data folder** A standard folder for application-specific configuration data that is used by all users.

`%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>`

or

`%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>`

**Documents folder** A standard folder for user-specific documents.

`%USERPROFILE%\Documents\dSPACE\<ProductName>\<VersionNumber>`

**Local Program Data folder** A standard folder for application-specific configuration data that is used by the current, non-roaming user.

`%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\<ProductName>`

**Accessing dSPACE Help and PDF Files**

After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as PDF files.

**dSPACE Help (local)** You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

**dSPACE Help (Web)** You can access the Web version of dSPACE Help at www.dspace.com/go/help.

To access the Web version, you must have a *mydSPACE* account.

**PDF files** You can access PDF files via the 📄 icon in dSPACE Help. The PDF opens on the first page.

# Introduction

**Important notes**

**Note**

- It is recommended to use the VirtualCOM object handling provided by AutomationDesk. This facilitates proper COM handling.
  For more information, refer to UsingVirtualCOMInAutomationDeskApplicationNote.pdf.
- Some instructions and examples are based on the **Test Framework** library. This library is discontinued with AutomationDesk 5.3 (dSPACE Release 2016-B). You can use its successor, the **Test Builder** library, in a similar way.
  For the migration of Test Framework projects to Test Builder projects, you can use a tool that is provided by dSPACE Support. For further information, refer to http://www.dspace.com/go/TestBuilderMigration.

# Technical Background

**Introduction**

This chapter explains the basic technical background of COM and AutomationDesk, and introduces recommended implementation structures.

**Where to go from here**

Information in this section

## Component Object Model (COM)

**General information**

*Wikipedia:*

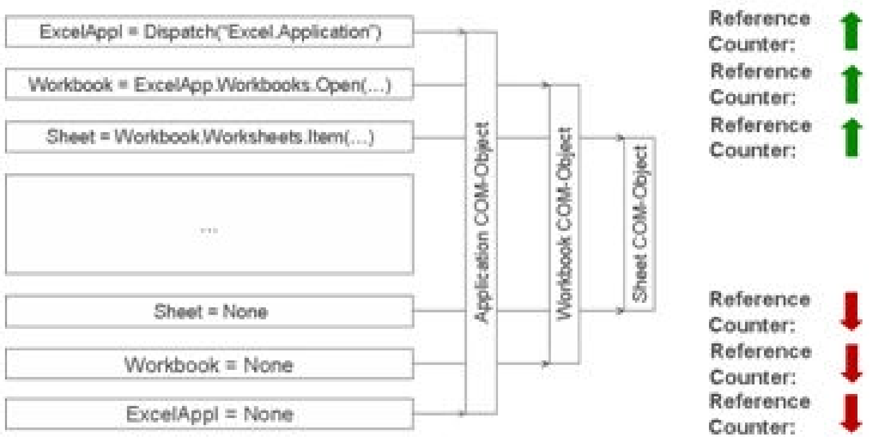
*Component Object Model (COM) is an interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in any programming language that supports the technology. The term COM is often used in the software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies. […].*

*The essence of COM is a language-neutral way of implementing objects which can be used in environments different from the one they were created in, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation because it forces component implementers to provide well-defined interfaces that are separate from the implementation. The different allocation semantics of languages are accommodated by making objects responsible for their own creation and* *destruction through reference-counting. Casting between different interfaces of an object is achieved through the QueryInterface() function. The preferred method of inheritance within COM is the creation of sub-objects to which method calls are delegated.*

**Life time of COM objects**

The communication between the client(s) and the server is realized via the Window Message Queue. Every COM object has to be connected to the message queue. The COM objects have to be cleaned up before the concerning Message Queue shuts down, otherwise it is possible to produce memory leaks after every usage.

The server counts the objects which are associated with it. Theoretically the COM server should know the order of cleaning up the COM objects. But in fact, it is much more secure to manually clean up the COM objects in the inverse order of the creation.



Picture 2-1: Lifetime and Reference Counter Model (COM)

## AutomationDesk

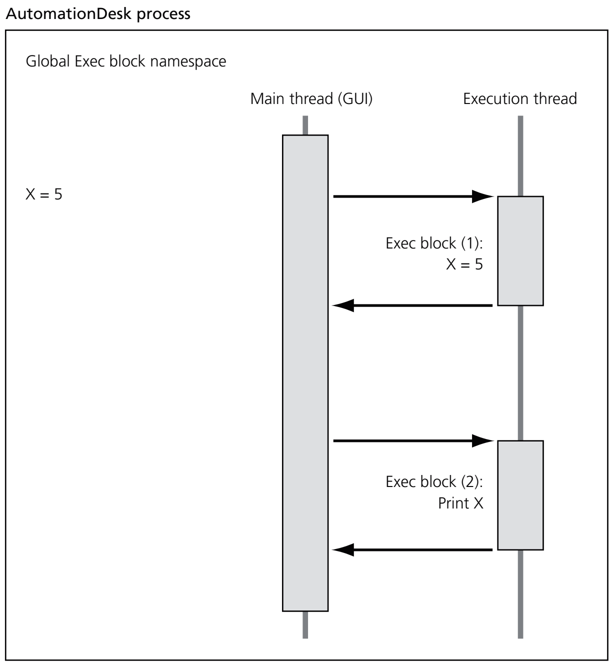
**Threads in AutomationDesk**

The AutomationDesk process contains several threads. The main thread contains the GUI. Each execution of AutomationDesk elements (Project, Folder, Sequence, Serial and Blocks) runs in a separated thread. In case of the execution of a parallel subsystem each subtree is started in an own thread.

**Python namespace in AutomationDesk**

AutomationDesk has one global namespace for the use of Python variables in Exec blocks. The namespace is not cleaned up until the end of the AutomationDesk session. After one execution, all assignments made in Exec blocks will remain in the namespace and can be accessed in the next executions.

Picture 2-2 shows the global Exec block namespace. In the first execution the Exec block (1) initializes a Python object, an integer variable. In the second execution the Exec block (2) uses the same namespace and can access the integer variable.

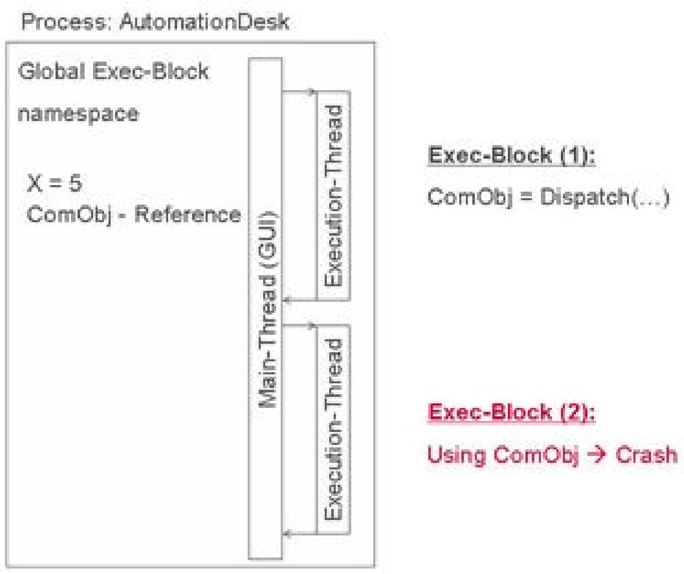


Picture 2-2: Global Exec block namespace

## COM objects in AutomationDesk threads

In AutomationDesk each thread has its own message loop. In conclusion of this and the fact that a COM object has to be released before its thread is terminated, all used COM objects have to be released manually by the user's code at the end of each execution (thread).

Picture 2-3 shows the global Exec block namespace. In the first execution the Exec block (1) initializes a COM object. In the second execution the Exec block (2) is expected to use it. The integer object is indeed available. Also the COM object is available, because of the global namespace. But its area of validity was terminated with the previous execution (thread) and the usage of the same COM object will raise an error.



Picture 2-3: Namespace and the life time of COM objects in AutomationDesk

**Execution Stop button in AutomationDesk**

The **Stop** button terminates the execution before starting the next block. In this case, no error handling or cleanup block will be executed. It can be therefore critical to use the **Stop** button in sequences using COM.

An exception is given by the high-level TestFramework library. When pressing a **Stop** button, in this case a popup window asks the user whether the cleanup parts of the current sequences shall be executed before the execution stops.

## Use of COM Objects in AutomationDesk

**Introduction**

The implementation of COM objects in AutomationDesk has to follow several rules given in the following chapters. The best COM handling however does not help if the server has bugs or problems.

**Assignment of COM objects**

COM objects must not be assigned to the usual AutomationDesk data objects (_AD_). Instead simple Python variables should be used:

- Incorrect (do not use):

```
_AD_.ExcelAppl = win32com.client.Dispatch("Excel.Application")
```

- Correct:

```
ExcelAppl = win32com.client.Dispatch("Excel.Application")
```

**Tip**

AutomationDesk provides a modified dispatch method that ensures proper COM handling. For example, this dispatch method is used in the ControlDesk Access demo scripts.

```
# Create and return COM object (Application)
return virtualcomobject.VirtualCOMObject.VC_Dispatch(
                "ControlDeskNG.Application")
```

## Cleanup of COM objects

By reason of the global Exec block namespace, it is essential to release all COM objects when the execution thread is terminated. This can be done by an assignment to `None` or using the `del` statement. For the realization, the `try-finally` error handling should be used. The following code structure shows a secure way for the implementation of COM handling (see Code 2-1). The Code can be divided into three parts: initialization usage and cleanup of the COM objects. The initialization is necessary, because an error can occur before a COM object is created. In this case the `del` statement itself would raise an error and the following code lines of the cleanup part would not be executed.

```
# Initialize the COM Objects
ComObj_1 = None
ComObj_2 = None
...
try:
   # Creation of the first COM object, e.g. Excel - Application
   # (see Code 3-2)
   ComObj_1 = win32com.client.Dispatch(<...>)
   ...
   # Creation of an additional COM object via the first COM object
   # (ComObj_1)
   ComObj_2 = ComObj_1.<...>(<...>)
   ...
finally:
   # Releasing the COM objects
   del ComObj_2
   del ComObj_1
```

Code 2-1: Implementation Structure with COM objects

The COM server generally provides a lot of objects, but normally only few are directly visible by the client. For example, Microsoft Excel provides an application object. In the further hierarchy the object for a single Excel sheet is dependent on the workbook object and the workbook object on the application object. Theoretically the server knows the cleanup order of all objects and shuts down after releasing the last COM object, independent of the hierarchy level of the COM object. The server documentation should also explain the rules for the shutdown.

If there are problems with the shutdown, one can try to cleanup the COM objects in the inverse order to their hierarchy level or their creation order:

```
# Creation order
ExcelAppl = win32com.client.Dispatch('Excel.Application')
Workbook  = ExcelAppl.Workbook.Item(<...>)
Sheet     = Workbook.Sheet.Item(<...>)
...
# Clean-up order (inverse to the creation order)
del Sheet
del Workbook
del ExcelAppl
```

Code 2-2: Cleanup order of COM objects

## Implementation Structure

**Introduction**

Concerning the COM handling in AutomationDesk, different implementations are possible. The main differences regard the placement of the error handling. The error handling can be realized in the Exec blocks as Python code or as a block solution in the AutomationDesk sequence.

**Error handling in Python (Exec blocks)**

The single-Exec-Block solution is interesting for tasks which are rarely used (in relation to the test duration). Using COM, this is the case if the communication between the server and the client is rare or if the client is able to bring the server to a state where the client is able to disconnect from the server but the server will stay open.

But this implementation is secure, because the error handling realizes a clean up of the COM objects and the execution Stop button cannot interrupt the block execution.

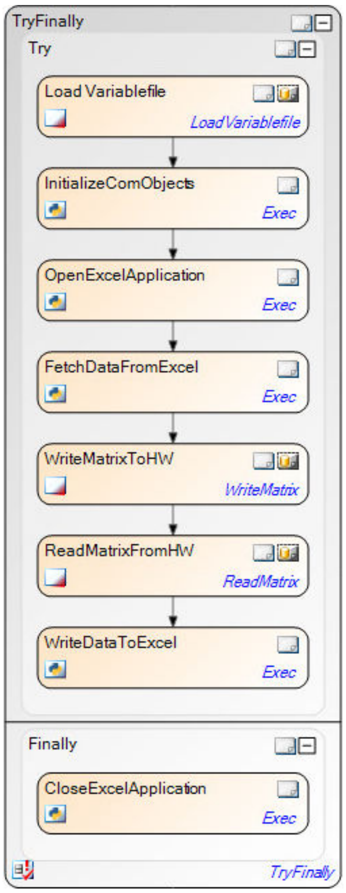
**Error handling in AutomationDesk (Sequences)**

The error handling can be realized with the TryFinally AutomationDesk block (Picture 2-4) and the TestFramework elements TestSequence and Test (Picture 2-5).

On the one hand, with this implementations the execution Stop button should not be used and on the other hand the whole 'TryFinally' or rather the whole 'Test' can be executed.

**Note**

The following implementation is based on the Platform Access library. This library is discontinued with AutomationDesk 5.3 (dSPACE Release 2016-B). You can use its successor, the XIL API Convenience library, in a similar way.

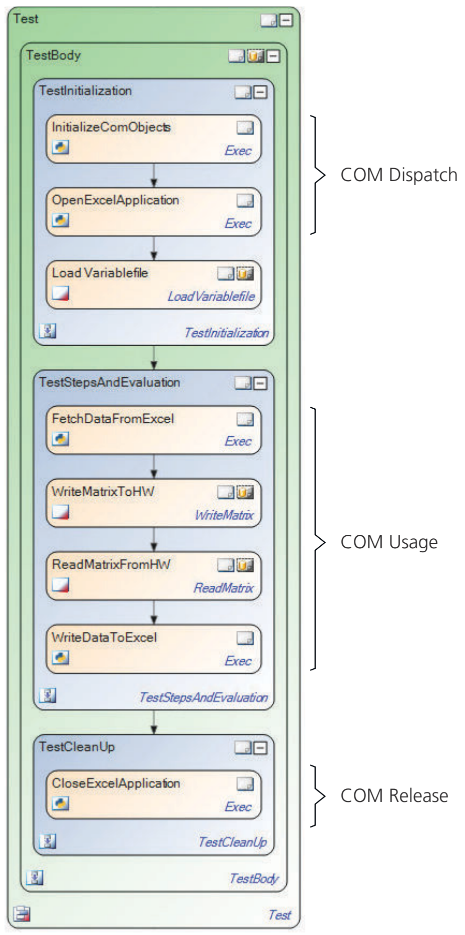


Picture 2-4: Implementation structure with the AutomationDesk Try-Finally element

**Note**

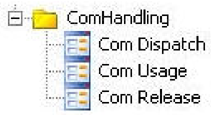
The following implementations are based on the **Test Framework** library. This library is discontinued with AutomationDesk 5.3 (dSPACE Release 2016-B). You can use its successor, the **Test Builder** library, in a similar way.

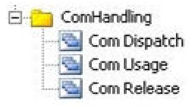
The **TestInitialization** phase of the Test block is used for dispatch call, the **TestStepsAndEvaluation** phase is used for the COM usage and clean up of the COM objects is performed in the **TestCleanUp** phase (Picture 2-5).



Picture 2-5: Implementation structure with the TestFramework Test block

In the following two implementations the COM usage is distributed over several Sequences or TestSequences. The first Sequence/TestSequence realizes the dispatch call, the following Sequences/TestSequences use the COM object(s) and the last Sequence/TestSequence is responsible for the cleanup.





Picture 2-6: COM handling over several Sequences (top) / TestSequences (bottom)

Both solutions have restrictions on the development and on the usage. It is not allowed to execute one of the Sequences/TestSequences separately. Only the folder which contains all Sequences/TestSequences with COM handling (“Com Dispatch”, “Com Usage” and “Com Release”) may be executed. The Stop button must not be pressed during the execution, because the “Com Release”-Sequence/TestSequence might not be executed. The difference between the Sequence and the TestSequence solution is the error handling: The TestSequence contains an error handling as described above.

# Examples

**Introduction**

This chapter shows the implementation structure for COM handling on the different implementation levels in AutomationDesk. The first example shows a remote control of Microsoft Outlook®. The implementation is on the Exec block level. The whole COM handling is implemented in a single Exec block.

Four different implementations are presented for the second example, a remote control of Microsoft Excel®. The COM handling is distributed over several Exec blocks and over several TestSequences.

**Where to go from here**

Information in this section

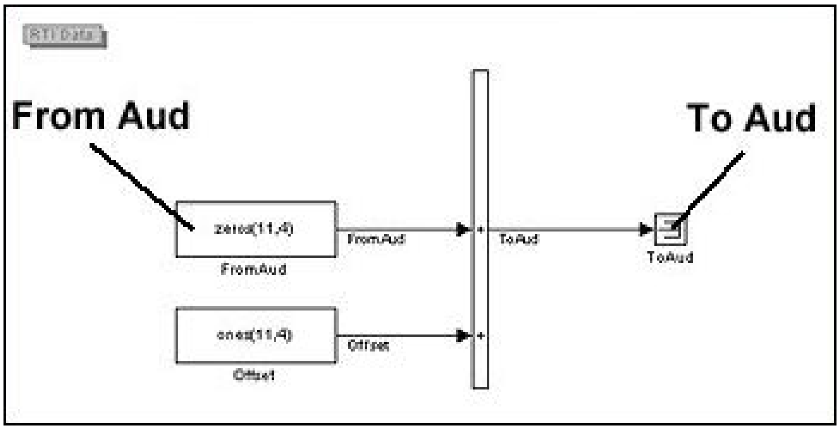
## Example Use Cases

**Example 1: Sending an email**

The AutomationDesk block ‘SendEMail’ realizes a remote control of Microsoft Outlook. The task of the block is to send an email.

**Example 2: Transfer Data between Excel and AutomationDesk**

The second example shows the remote control of Excel. An Excel sheet contains the parameterization of the real-time model and should get the results too.



Picture 3-1: Model for the Excel-AutomationDesk-Real-Time hardware data transfer

## Implementation

**Introduction**

The first example can be implemented in a single Exec block. The only interaction with the AutomationDesk project consists in the block parameters (data objects) which can be used to set necessary information for sending an email, for example, the recipient and the subject. In the second example more interaction between AutomationDesk and Excel is necessary. Excel will be needed twice during the project execution; at first at the start of the execution, for reading the model parameterization, and at the end of the execution to store the result.

**Implementation Example 1**

**Behavior of Microsoft Outlook during Remote Control** The behavior of Outlook using the remote control is as follows: If Outlook is closed when creating the COM object, it will be opened, and the final COM object cleanup correspondingly causes the termination of Outlook. Otherwise, the related COM object connects to the open Outlook and a the clean up only closes the connection between the COM object and Outlook.

**Block Interface and Code** Block-Interface:

| **Name (type)** | **Description** |
|---|---|
| To (String object) | Addresses of the email |
| Subject (String object) | Subject of the email |
| Body (String object) | Email text |
| Attachment (String object) | Path of the attachment file |

The remote control of Outlook contains two COM objects, the application object (Python variable name 'Outlook', line 7) and the object of the email (Python variable name ‘EMail’, line 8). The email object depends on the application object and should be released before the application object is released.

```
1  import win32com.client
2  
3  Outlook = None
4  EMail   = None
5  
6  try:
7      Outlook = win32com.client.Dispatch("Outlook.Application")
8      EMail = Outlook.CreateItem(0)
9      EMail.To      = _AD_.To
10    EMail.Subject = _AD_.Subject
11    EMail.Body    = _AD_.Body
12    if _AD_.Attachment != '':
13        EMail.Attachments.Add(_AD_.Attachment, 1, 1)
14    EMail.Send()
15 finally:
16    del EMail
17    del Outlook
```

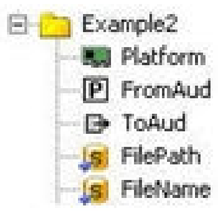
Code 3-1: SendEMail

## Implementation Example 2

**Behaviour of Microsoft Excel during Remote Controlling** The Excel application will be used “very often” during the project execution. For this case it is better to open Excel at the beginning of the execution and not to close it until the end of the execution. The read and write access is realized in several separate Exec blocks.

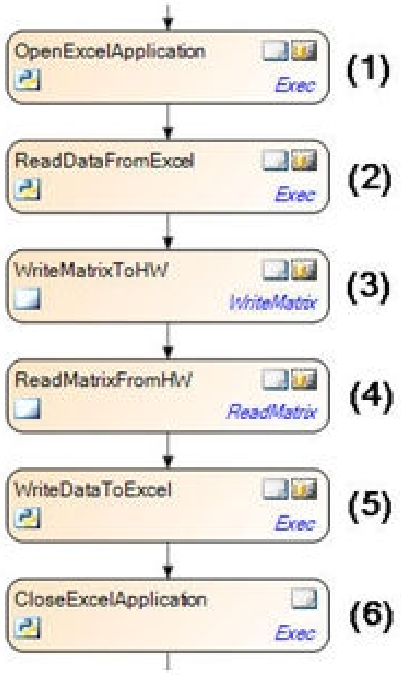
**Block Interface and Code** The following parameters have to be set to the current location of the example:

| Name (type) | Description |
|---|---|
| Platform (Platform object) | Specify the Real-Time hardware, SDF file for the example: `<...>\work\Matlab\DummyMdl.sdf` |
| FilePath (String object) | Path of the Excel file: `<...>\work\File\` |
| FileName (String object) | Name of the Excel file: `Test.xls` |



Picture 3-2: AutomationDesk-Project Structure of the Example 2

**Implementation Version A** The Picture 3-3 shows the first implementation of the Excel example.



Picture 3-3: Example 2 – Implementation Version A, Sequence View

In this implementation every COM handling block contains a clean up routine. After each runtime error the COM instance is cleaned up. The problem in this case is the state of the remote-controlled software after the abort. For example, if the execution is aborted after block 3 the Excel file is still open and has to be closed manually before restarting the project again.

On the other side, the COM Handling is safe. It is not possible to produce COM memory leaks.

The internal implementation of the following blocks is described:

- OpenExcelApplication (1)
- ReadDataFromExcel (2)
- WriteDataToExcel (5)
- CloseExcelApplication (6)

The blocks (3) and (4) are standard AutomationDesk blocks.

Block (1) realizes the opening of Excel and makes the Excel instance visible. After this action the COM object will be destroyed. The Excel instance is still running. If more actions with Excel are planned, a new COM object has to be created by invoking the dispatch call again. The call is implemented in blocks (2), (5), and (6).

```
import win32com.client

# Initialize the COM Objects
ExcelAppl = None

```

```
try:
   # Creation of the COM object
   #  Open an Excel instance
   ExcelAppl = win32com.client.Dispatch("Excel.Application")
  #  make the Excel instance visible (not necessary)
  ExcelAppl.Visible = 1
finally:
   # Release the COM object
   del ExcelAppl
```

Code 3-2: OpenExcelApplication - Block (1)

Block (2) opens the workbook and transfers the matrix to AutomationDesk. Again, the COM objects are destroyed after using them. In this case be careful. At first the COM object for Excel is generated (Dispatch – line 12). The next COM objects are generated in line 16 (for the Excel workbook/file) and 18 (for the Excel sheet). In the finally part the COM objects have to be destroyed in the inverse order of the creation:

Creation order of the COM objects:

- Excel object
- Workbook
- Sheet

Destruction order of the COM objects:

- Sheet
- Workbook
- Excel Object

```
 1  import win32com.client
 2  import os.path
 3
 4  # Initialize the COM objects
 5  Sheet     = None
 6  Workbook  = None
 7  ExcelAppl = None
 8
 9  try:
10     # Creation of the COM object
11     #  Open an Excel instance
12     ExcelAppl = win32com.client.Dispatch("Excel.Application")
13     # Open the Excel file
14     ExcelAppl.Workbooks.Open(os.path.join(_AD_.FilePath,_AD_.FileName))
15     # Fetch COM interface of the Excel file
16     Workbook = ExcelAppl.Workbooks.Item(_AD_.FileName)
17     # Fetch COM interface of the Excel sheet
18     Sheet = Workbook.Worksheets.Item(_AD_.SheetName)
19     # Fetch the table from the Excel file
20     # (e.g. From cell 'A1', 'D11' -> Matrix size: 4x11)
21     # Return value is a tuple
22     Data = Sheet.Cells.Range(_AD_.StartCell,   _AD_.EndCell).Value
23     # the data type of 'Data': tuple of tuples
24     # conversion to List of Lists
25     _AD_.Data = map(list, Data)
26  finally:
27     # Release the COM objects
28     del Sheet
29     del Workbook
30     del ExcelAppl
```

Code 3-3: ReadDataFromExcel - Block (2)

Block (5) transfers the matrix back to Excel and saves the workbook.

```python
import win32com.client

# Initialize the COM Objects
Sheet     = None
Workbook  = None
ExcelAppl = None

try:
     # Creation of the COM object
    # Open an Excel instance
    ExcelAppl = win32com.client.Dispatch("Excel.Application")
    # Fetch COM interface of the Excel file
    Workbook = ExcelAppl.Workbooks.Item(_AD_.FileName)
    # Fetch COM interface of the Excel sheet
    Sheet = Workbook.Worksheets.Item(_AD_.SheetName)
    # write the matrix back to the Excel sheet
    Sheet.Cells.Range(_AD_.StartCell,_AD_.EndCell).Value =_AD_.Data
    # save the Excel file
    Workbook.Save()
finally:
    # Release the COM objects
    del Sheet
    del Workbook
    del ExcelAppl
```

Code 3-4: WriteDataToExcel – Block (5)

Block (6) closes Excel:

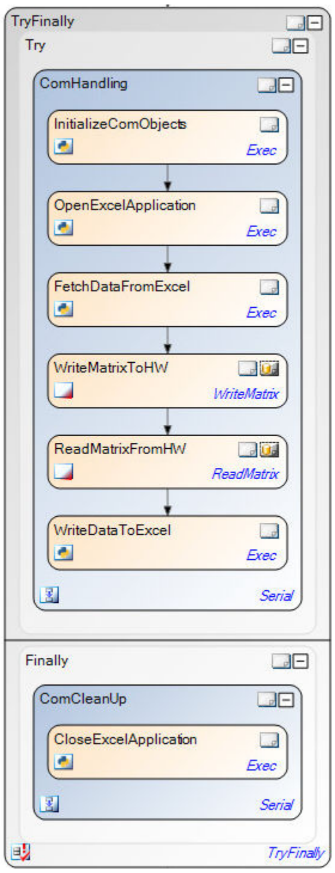
```python
import win32com.client

# Initialize the COM objects
ExcelAppl = None

try:
     # Creation of the COM object
     # Open an Excel instance
     ExcelAppl = win32com.client.Dispatch("Excel.Application")
    # Hide the Excel application
    ExcelAppl.Visible = 0
    # force Excel shutdown
    ExcelAppl.Quit()
finally:
    # Release the COM object
    del ExcelAppl
```

Code 3-5: CloseExcelApplication – Block (6)

**Implementation Version B** The next implementation version (Picture 3-4) differs from the previous implementation structure. The error handling is realized with AutomationDesk's TryFinally block. The dispatch call and the realizing of each COM object are invoked only one time.

**Note**

- Be careful in the usage of this implementation version. Do not press the execution **Stop** button, because otherwise the Finally part of the TryFinally will not be executed. In this case the COM object is not released and it is possible to block the whole system or "only" a part of the system. In this case it is not possible to close Excel.
- The following implementation is based on the **Platform Access** library. This library is discontinued with AutomationDesk 5.3 (dSPACE Release 2016-B). You can use its successor, the **XIL API Convenience** library, in a similar way.



Picture 3-4: Remote Controlling of Excel with the TryFinally block

The following code lines are not very different from the code lines of the previous example. The Exec blocks do not contain the error handling and only the second block has to call the `Dispatch` command. The first block contains the initialization of all used variables for COM objects:

Block "InitializeComObjects":

```
# Initialize the COM Objects
Sheet     = None
Workbook = None
ExcelAppl  = None
```

Block: “OpenExcelApplication”

```
# import COM Module
import win32com.client
# Creation of the COM Object
#  Open an Excel-Instance
ExcelAppl = win32com.client.Dispatch("Excel.Application")
#  make the Excel-Instance visible (not necessary)
ExcelAppl.Visible = 1
```

Code 3-6: OpenExcelApplication

Block: “ReadDataFromExcel”

```
import os.path
# Open the Excel file
ExcelAppl.Workbooks.Open(os.path.join(_AD_.FilePath, _AD_.FileName))
# Fetch COM interface of the Excel file
Workbook = ExcelAppl.Workbooks.Item(_AD_.FileName)
# Fetch COM interface of the Excel sheet
Sheet = Workbook.Worksheets.Item(_AD_.SheetName)
# Fetch the table from the Excel file
# (e.g. from cell 'A1', 'D11' -> Matrix 4,11)
# Return value is a tuple
Data = Sheet.Cells.Range(_AD_.StartCell, _AD_.EndCell).Value
# the data type of 'Data': tuple of tuples
# conversion to List of Lists
_AD_.Data = map(list, Data)
```

Code 3-7: ReadDataFromExcel

Block: “WriteDataToExcel”

```
# write the matrix back to the Excel sheet
Sheet.Cells.Range(_AD_.StartCell, _AD_.EndCell).Value = _AD_.Data
# save the Excel file
Workbook.Save()
```

Code 3-8: WriteDataToExcel

Block: “CloseExcelApplication”

```
# Release the COM objects
del Sheet
del Workbook
# Hide the Excel application
ExcelAppl.Visible = 0
# force a Excel shutdown
ExcelAppl.Quit()
# Release the COM objects
del ExcelAppl
```

Code 3-9: CloseExcelApplication

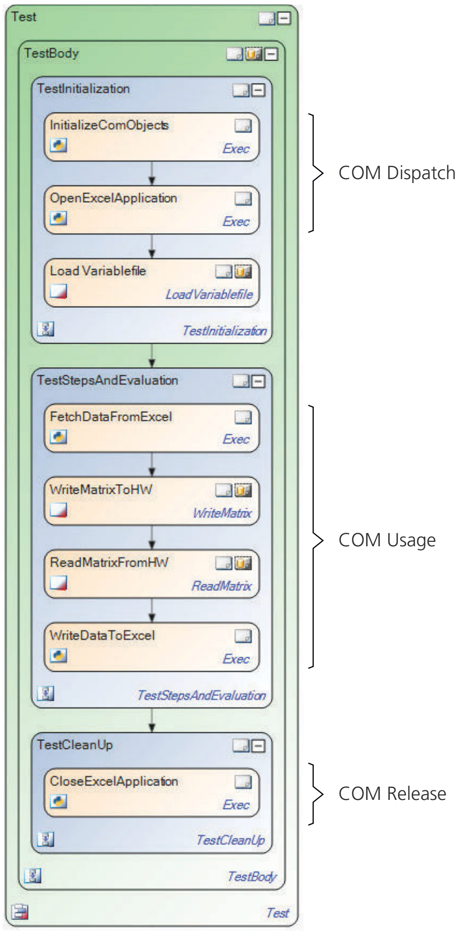
### Implementation Version C

**Note**

The following implementations are based on the Test Framework library. This library is discontinued with AutomationDesk 5.3 (dSPACE Release 2016-B). You can use its successor, the Test Builder library, in a similar way.

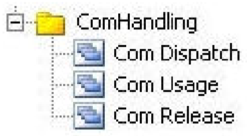
The last implementation (Picture 3-4) can be also implemented with the TestSequence element and the Test element of the TestFramework library (Picture 3-5).

The blocks are the same as in the TryFinally solution, but the Stop button problem is not so critical. After pressing the Stop button the user is asked, whether AutomationDesk should execute all clean up parts. In the case of COM handling it is always necessary to run the clean up parts. The blocks are the same like the previous implementations (Implementation Version B).

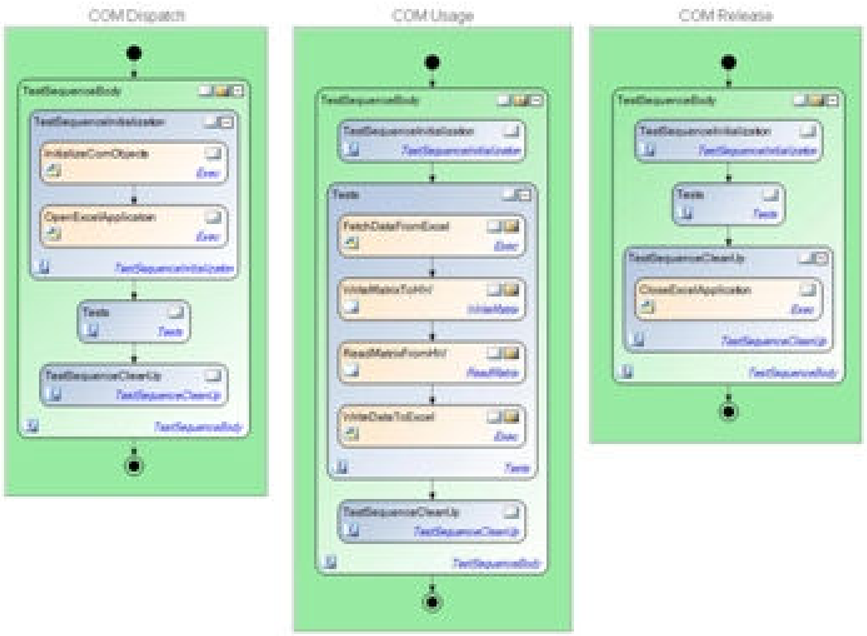


Picture 3-5: Remote Controlling of Excel with TestFramework element Test

**Implementation Version D** The implementation Version D distributes the COM handling over several TestSequences. The blocks are the same as in the previous implementations (Implementation Version B). The execution of the structure is only allowed on the enclosing folder. The pressing of the Stop button is not allowed during the whole execution, because the execution will stop and the last sequence will not be executed. As consequence all COM objects will not be released.



Picture 3-6: Sequence-Structure of the Implementation Version D



Picture 3-7: Remote Controlling of Excel with TestFramework element TestSequence

# Conclusion

## Comparison of the Implementations

**Implementation-dependent behavior**

The following table shows the conclusion of COM handling with the different implementation solutions described in this document.

| COM Handling Implementation ... | Usage of AutomationDesk Blocks Between two COM Accesses | Behavior Concerning Run-Time Errors | Behavior Concerning the Stop Button |
|---|---|---|---|
| ... in Exec block (Implementation Example 1) | Not possible | Secure | Secure |
| ... in TryFinally block (Implementation Example 2 / Version B) | Possible | Secure | Critical |
| ... in Test/TestSequence (Implementation Example 2 / Version C) | Possible | Secure | Secure |
| ... in Sequence | Possible | Critical | Critical |
| ... over several Tests/Testsequences (Implementation Example 2 / Version D) | Possible | Secure | Critical |
| ... over several sequences | Possible | Critical | Critical |