ANSYS ACT Migration for Legacy Workbench Customization Products

Copyright and Trademark Information

© 2017 ANSYS, Inc. Unauthorized use, distribution or duplication is prohibited.

ANSYS, ANSYS Workbench, AUTODYN, CFX, FLUENT and any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries located in the United States or other countries. ICEM CFD is a trademark used by ANSYS, Inc. under license. CFX is a trademark of Sony Corporation in Japan. All other brand, product, service and feature names or trademarks are the property of their respective owners. FLEXlm and FLEXnet are trademarks of Flexera Software LLC.

Disclaimer Notice

THIS ANSYS SOFTWARE PRODUCT AND PROGRAM DOCUMENTATION INCLUDE TRADE SECRETS AND ARE CONFIDENTIAL AND PROPRIETARY PRODUCTS OF ANSYS, INC., ITS SUBSIDIARIES, OR LICENSORS. The software products and documentation are furnished by ANSYS, Inc., its subsidiaries, or affiliates under a software license agreement that contains provisions concerning non-disclosure, copying, length and nature of use, compliance with exporting laws, warranties, disclaimers, limitations of liability, and remedies, and other provisions. The software products and documentation may be used, disclosed, transferred, or copied only in accordance with the terms and conditions of that software license agreement.

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2008 companies.

U.S. Government Rights

For U.S. Government users, except as specifically granted by the ANSYS, Inc. software license agreement, the use, duplication, or disclosure by the United States Government is subject to restrictions stated in the ANSYS, Inc. software license agreement and FAR 12.212 (for non-DOD licenses).

Third-Party Software

See the legal information in the product help files for the complete Legal Notice for ANSYS proprietary software and third-party software. If you are unable to access the Legal Notice, contact ANSYS, Inc.

Published in the U.S.A.

Contents

Advisory

As of release 19.0, ANSYS has ended support for the Workbench customization products listed below. Our drive toward a simplified and consistent platform customization vision now delivers the preferred ANSYS ACT approach for your Workbench customization projects. This document provides transition guidance.

ANSYS understands the importance of software customization. We debuted ANSYS Workbench 12.0 as an innovative platform for cohesive simulation workflow management. We placed high priority on robust Workbench journaling, scripting, and extensibility. Accordingly, early in the Workbench timeline, we offered a pair of customization tools:

- External Connection Add-in (Released at 12.1)

  The External Connection Add-in supported lightweight integration of external applications into the Workbench **Project Schematic**. You could leverage additional customization opportunities by defining new user interface elements. Solutions contained plain text XML files and IronPython scripts. Development and deployment required neither additional licenses nor specialized tools.

- ANSYS Workbench Software Development Kit (SDK) (Released at 13.0)

  The ANSYS Workbench Software Development Kit (SDK) exposed C#-based APIs used by new feature development to plug into Workbench's modular add-in architecture. You could enrich framework functionality and data-integrate external applications by creating add-ins of your own. Add-in development required both an additional "ANSYS Customization Suite" license and Microsoft .NET developer tools.

Please discontinue using customization solutions built with the External Connection Add-in and SDK. We will neither support new External Connection Add-in development nor ship SDK packages at 19.0.

Existing Workbench projects (WBPJ or WBPZ files) that use External Connection Add-in references will continue to operate at 19.0. However, we cannot guarantee compatibility beyond 19.0.

The SDK's compiled nature requires immediate action by content providers. Developers should re-implement their solutions with ACT extensions as soon as possible to ensure compatibility for 19.0 and beyond. Users of SDK-based solutions should contact the solution owners or distributors to request new ACT-based versions.

ANSYS ACT

Application customization can seem complex and time-consuming. That's why we've developed ANSYS ACT. ACT removes the coding complexity to let you focus on driving your designs. Deliver in days what would traditionally take weeks or months.

ACT provides a consistent approach for custom simulation experiences across the ANSYS product line. Easy-to-use app creation tools and a global deployment strategy offer you the most robust and straightforward ANSYS ecosystem opportunity to date.

Not an expert programmer? Don't worry. ACT uses simple, plain-text file formats. XML describes your customization. IronPython brings it to life. Additional ACT tools—App Builder, Debugger, and Console—guide you through the app creation process.

ACT for Workbench

ANSYS Workbench manages simulation workflow with clarity and order. Why not leverage the straightforward ACT extension approach within your Workbench projects?

Make our Workbench environment your own with ANSYS ACT. Inject your simulation requirements in a platform where your own engineering tools coexist with the industry's broadest suite of advanced engineering simulation technology.

Accomplish the opportunities offered by the legacy External Connection Add-in and SDK with improved ease and rapid delivery. Uncover new customization possibilities with expanded access beyond legacy capabilities.

Migration

The Workbench External Connection Add-in and SDK end-of-life requires you to migrate your customization solutions to ANSYS ACT.

Before you review our migration guidance, please familiarize yourself with ACT philosophy, terms, and usage—especially if you are new to ACT customization. We provide the following educational guides:

- *ANSYS ACT Developer's Guide*—This guide offers an introduction to ACT, describing its capabilities and how to use them. While primarily intended for developers of ACT apps, it also provides information for end users who manage and execute apps.
- *ANSYS ACT Customization Guide for Workbench*—This guide offers information for using ACT to customize Workbench.

The nature of customization and providing general application hooks offers almost unlimited opportunity and creative solutions. The guidance we provide in this document covers common use cases for each of our legacy customization products. We do not exhaustively cover every possible scenario. Please use the following information as a foundation for migration and general guidance for mapping between legacy code and new ACT APIs.

External Connection Add-in

The External Connection Add-in supported three customization paths:

- Customizing the Workbench User Interface
- Integrating an External Application
- Creating Custom Systems and Components

We will cover the migration process for each path. Along the way, we will illustrate migration by using the examples listed in the legacy *Workbench External Connection Add-In User Guide*.

Customizing the Workbench User Interface

To customize the Workbench user interface with the External Connection Add-in, you defined a `CustomizationToolBarConfiguration.xml` file inside a `Customization` folder and placed the folder either in the ANSYS `Addins` directory or your user application data directory:

ANSYS `Addins` directory:

```
ANSYS_INSTALL/Addins/ExternalConnection
```

User application data directory for Windows:

```
%APPDATA%\Ansys\version\ExternalConnection
```

User application data directory for Linux:

```
$HOME/.config/Ansys/version/ExternalConnection
```

If your custom user interface entries referenced scripts and images, you also created `Images` and `Scripts` folders under the `Customization` folder.

The `CustomToolBarConfiguration.xml` file defined <GuiOperation> entries that Workbench converted into toolbar buttons, context menu entries, or menu entries.

With ACT, you define an XML extension definition file, extension folder, scripts, and an images sub-folder. You must place the XML file and folder in one of three locations:

1. **ACT extensions directory:**

```
ANSYS_INSTALL/Addins/ACT/extensions
```

2. **User application data directory**

   For Windows:

```
%APPDATA%\Ansys\version\ACT\extensions
```

   For Linux:

```
$HOME/.config/Ansys/version/ACT/extensions
```

3. **User-specified directory**
   In Workbench, you can set a user-specific extension location in **Tools > Options > Extensions**.

Inside the ACT extension XML file, you define an `<interface>` tag to describe new user Interface elements.

Toolbar Buttons

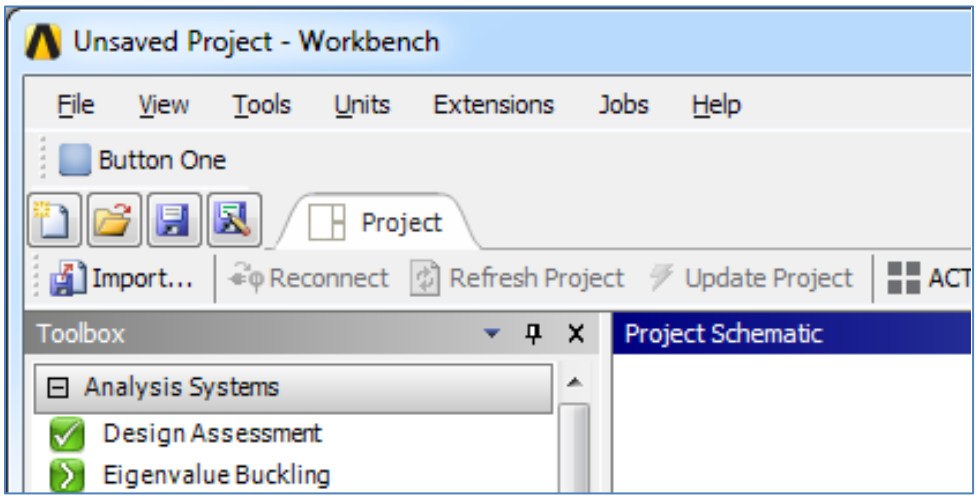
You can use specific ACT XML elements to define toolbars and toolbar entries at the interface level:

```
<extension version="1" name="ToolBar" icon="images\toolbar.png">
 <guid shortid="ToolBar">69d1235b-e138-4841-a13a-de12238c83f2</guid>
 <script src="main.py" />
 <interface context="Project">
  <images>images</images>
  <toolbar name="MyToolbar" caption="My Toolbar">
   <entry name="Button1" caption="Button One" icon="mybutton1">
    <callbacks>
     <onclick>button1Click</onclick>
    </callbacks>
   </entry>
  </toolbar>
 </interface>
</extension>
```

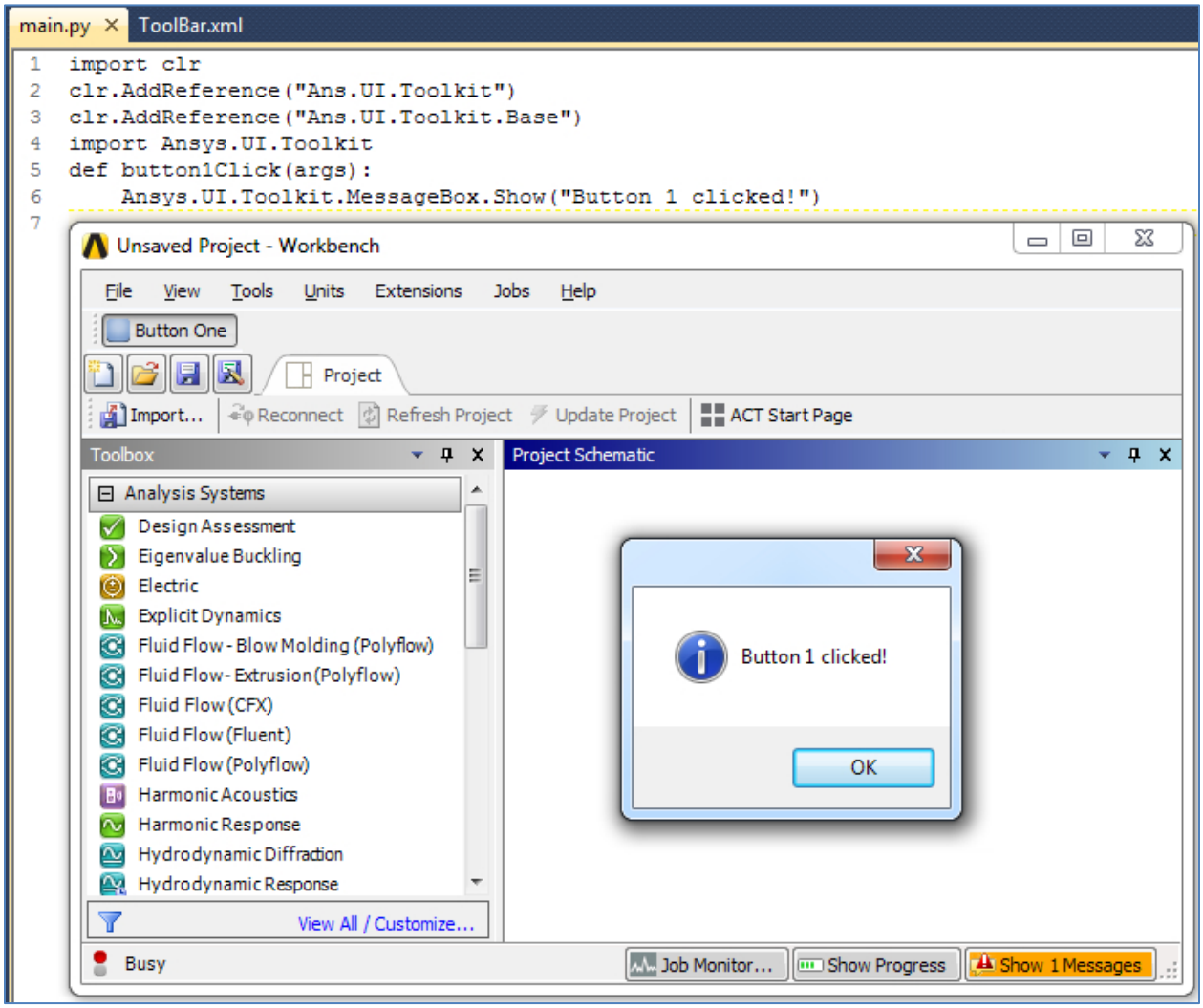
For comparison, an External Connection Add-in implementation follows:

```
<Configuration ShowEditConfiguration="True">

 <GuiOperations>

  <GuiOperation Name="Button One" Priority="2" ScriptFile="main.py"
Image="mybutton1" Type="ToolbarButton" SourceType="Python" Entry="My
Toolbar"/>

 </GuiOperations>

</Configuration>
```

This figure shows the result within Workbench:



The callback `button1Click` maps to the method `button1Click` defined in the extension's script `main.py`. Clicking the button displays a message box:



Context Menu Entries

The External Connection Add-in allowed you to specify limited context menu entries, which you accessed by clicking the right-mouse button. ACT provides two ways to define these same actions:

- Custom task context menu entries
- Application-wide context menu entries

Context menu entries defined for custom tasks are covered in Creating Custom Systems and Components. For application-wide context menu entries, you leverage the same interface block as toolbar buttons. Instead of distinct XML elements, you enroll two callbacks for user interface initialization and termination:

```
<extension version="1" name="DynamicContextMenus"
icon="images\contextMenu.png">
```

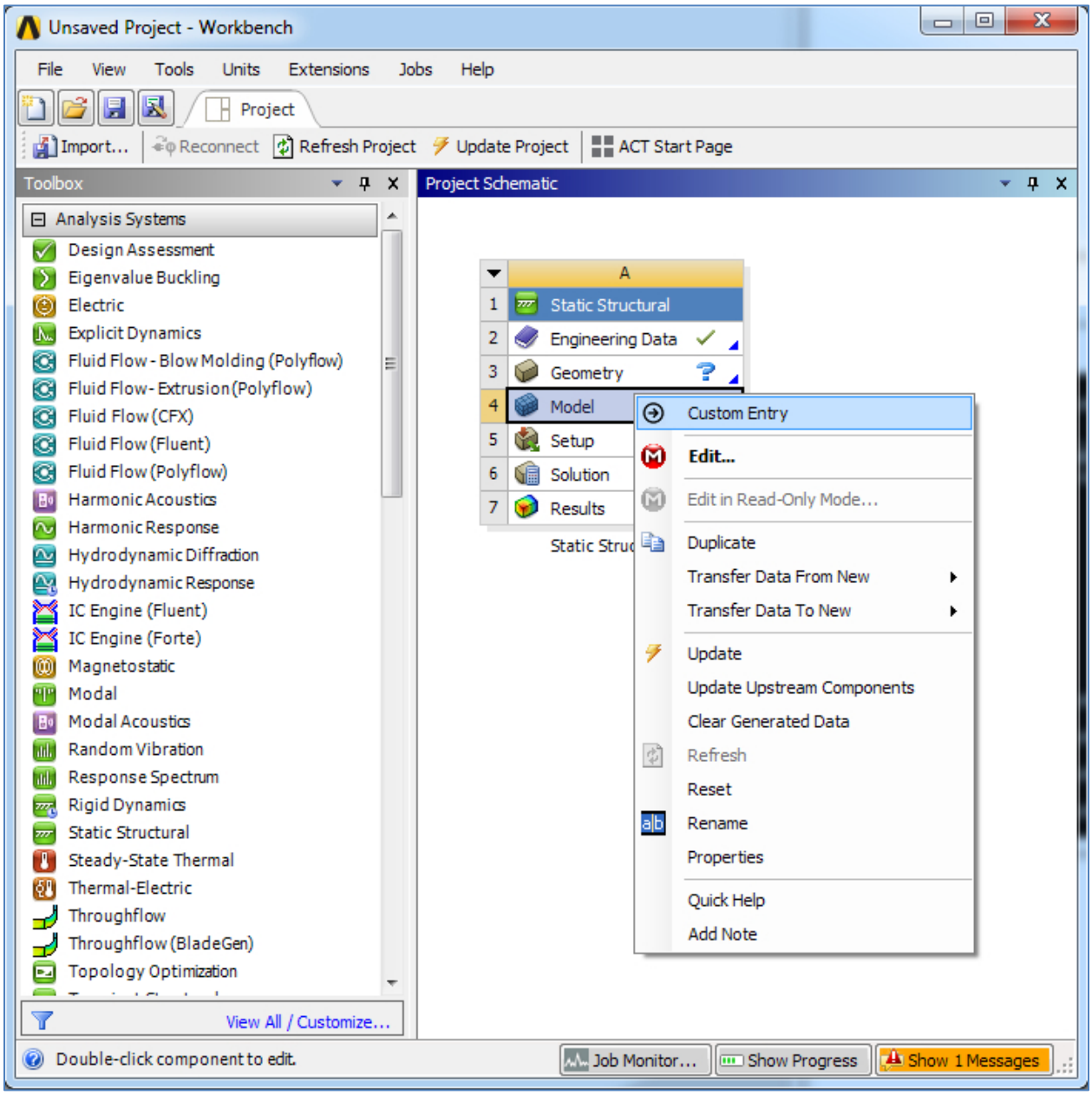
```
  <guid shortid="DynamicContextMenus">9e701503-e8bc-4b9c-a12e-
6c285be5edc9</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>createContextMenu</oninit>
      <onterminate>removeContextMenu</onterminate>
    </callbacks>
  </interface>
</extension>
```

The two callbacks provide empty *hooks* into Workbench's user interface. You use Workbench-provided APIs to dynamically define your new context menus:

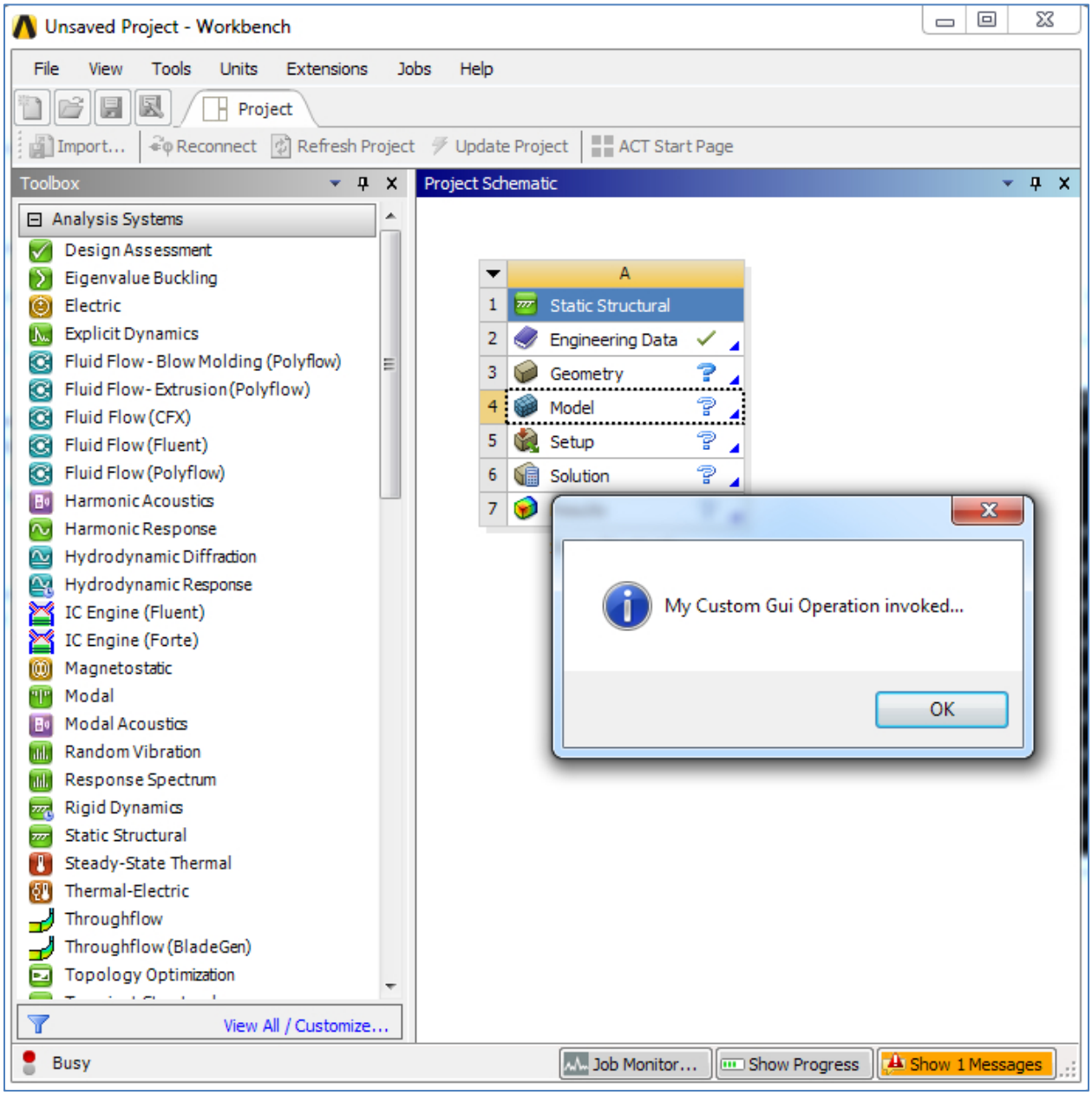
```
import System
import clr
clr.AddReference("Ans.UI")
import Ansys.UI
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit
clr.AddReference("Ans.Core")
import Ansys.Core


def createContextMenu(context):
  uiMgr = Ansys.UI.UIManager.Instance
  menuMgr = uiMgr.ContextMenuManager
  imgLib = Ansys.UI.Toolkit.ImageLibrary()
  for imgDir in ExtAPI.Extension.GetImageDirectories():
    imgLib.AddImagesFromDirectory(imgDir)
  menuMgr.AddDynamicEntity(None, Ansys.UI.MenuEntityType.MenuEntry, "Custom
Entry", False, "customEntryIcon", imgLib, System.Single.Parse('1.0'), None,
```

```
MyCustomGuiOperation(), True, False, Ansys.UI.OpClass.None, False, True,
True)

def removeContextMenu(context):

  uiMgr = Ansys.UI.UIManager.Instance

  menuMgr = uiMgr.ContextMenuManager

  menuMgr.RemoveDynamicEntity("DynamicEntity:ContextMenu:Custom Entry")

class MyCustomGuiOperation(Ansys.UI.Interfaces.IGuiOperation):

  def Invoke(self, context):

    Ansys.UI.Toolkit.MessageBox.Show("My Custom Gui Operation invoked...")

  def GuiItemCallBack(self, context):

    context.Visible = False

    context.Enabled = False

    selection =
context.View.GetSingleUISelection[Ansys.Core.DataModel.ProjectSystem.DataCon
tainerReference]()

    if ExtAPI.DataModel.TaskGroups.Count > 0 and selection != None:

      context.Enabled = True

      context.Visible = True
```

This code adds a visible context menu that is only enabled when one or more task groups exist in the **Project Schematic** and you right-click a task. The `GuiItemCallBack` approach for GuiOperations allows you to leverage applicability previously inaccessible to External Connection Add-in solutions.


Unsaved Project - Workbench
File
View
Tools
Units
Extensions
Jobs
Help
Project
Import...
Reconnect
Refresh Project
Update Project
ACT Start Page
Toolbox
Analysis Systems
Design Assessment
Eigenvalue Buckling
Electric
Explicit Dynamics
Fluid Flow - Blow Molding (Polyflow)
Fluid Flow- Extrusion (Polyflow)
Fluid Flow (CFX)
Fluid Flow (Fluent)
Fluid Flow (Polyflow)
Harmonic Acoustics
Harmonic Response
Hydrodynamic Diffraction
Hydrodynamic Response
IC Engine (Fluent)
IC Engine (Forte)
Magnetostatic
Modal
Modal Acoustics
Random Vibration
Response Spectrum
Rigid Dynamics
Static Structural
Steady-State Thermal
Thermal-Electric
Throughflow
Throughflow (BladeGen)
Topology Optimization
View All / Customize...
Project Schematic
A
1 Static Structural
2 Engineering Data
3 Geometry
4 Model
5 Setup
6 Solution
7 Results
Custom Entry
Edit...
Edit in Read-Only Mode...
Duplicate
Transfer Data From New
Transfer Data To New
Update
Update Upstream Components
Clear Generated Data
Refresh
Reset
Rename
Properties
Quick Help
Add Note
Double-click component to edit.
Job Monitor...
Show Progress
Show 1 Messages

The action displays a message box:



Menu Entries

Much like context menu entries, you use the initialize and terminate callbacks on the extension user interface to define, register, and unregister menu entries:

```
<extension version="1" name="DynamicMenuBarMenus"
icon="images\menuBarMenu.png">
```

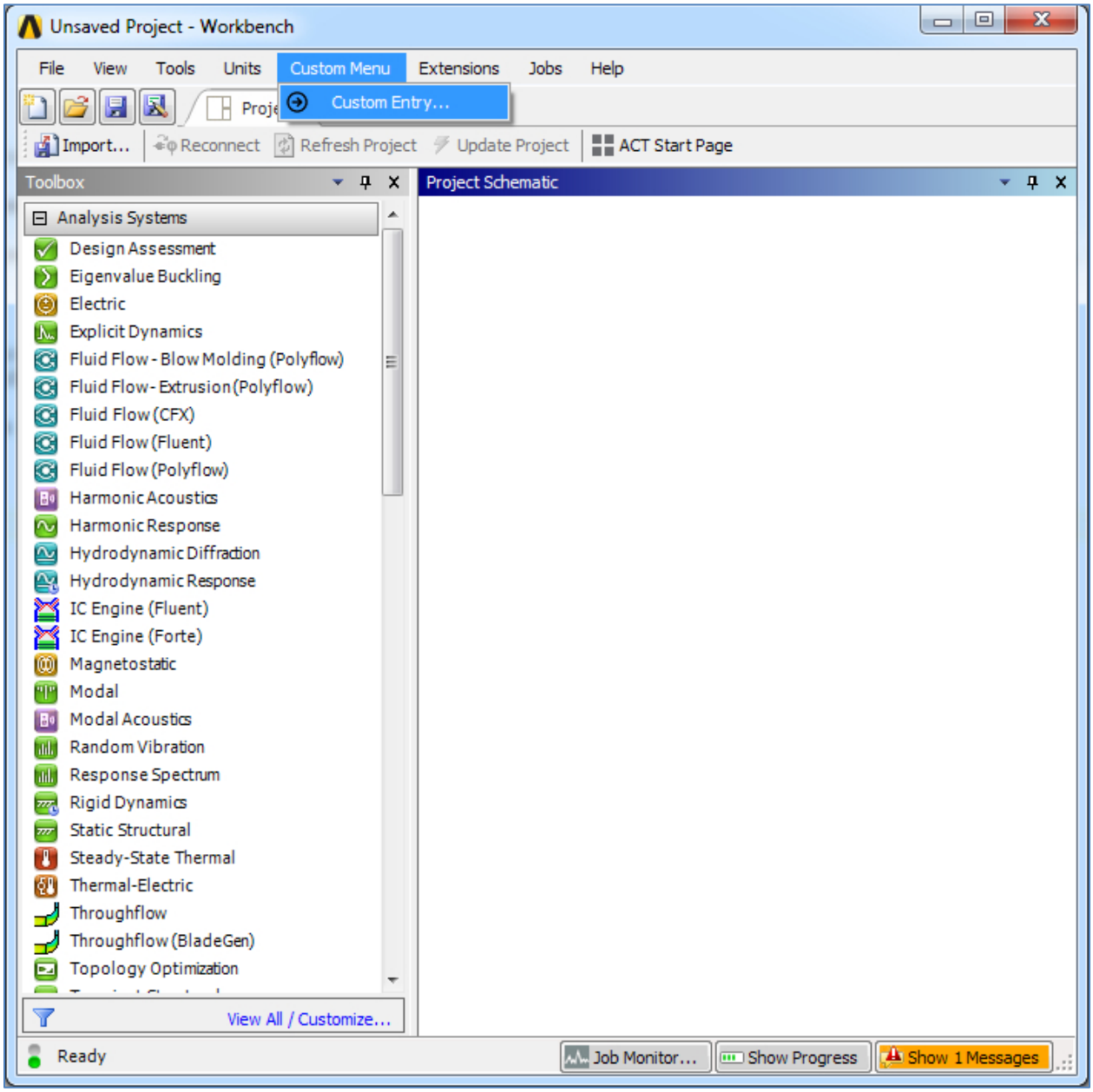
```
  <guid shortid="DynamicMenuBarMenus">886bfb1d-dd9c-42fa-9fcf-
c86ccc73bfbc</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>createMenu</oninit>
      <onterminate>removeMenu</onterminate>
    </callbacks>
  </interface>
</extension>
```

You access the MenuManager in your callbacks to register the custom entries:

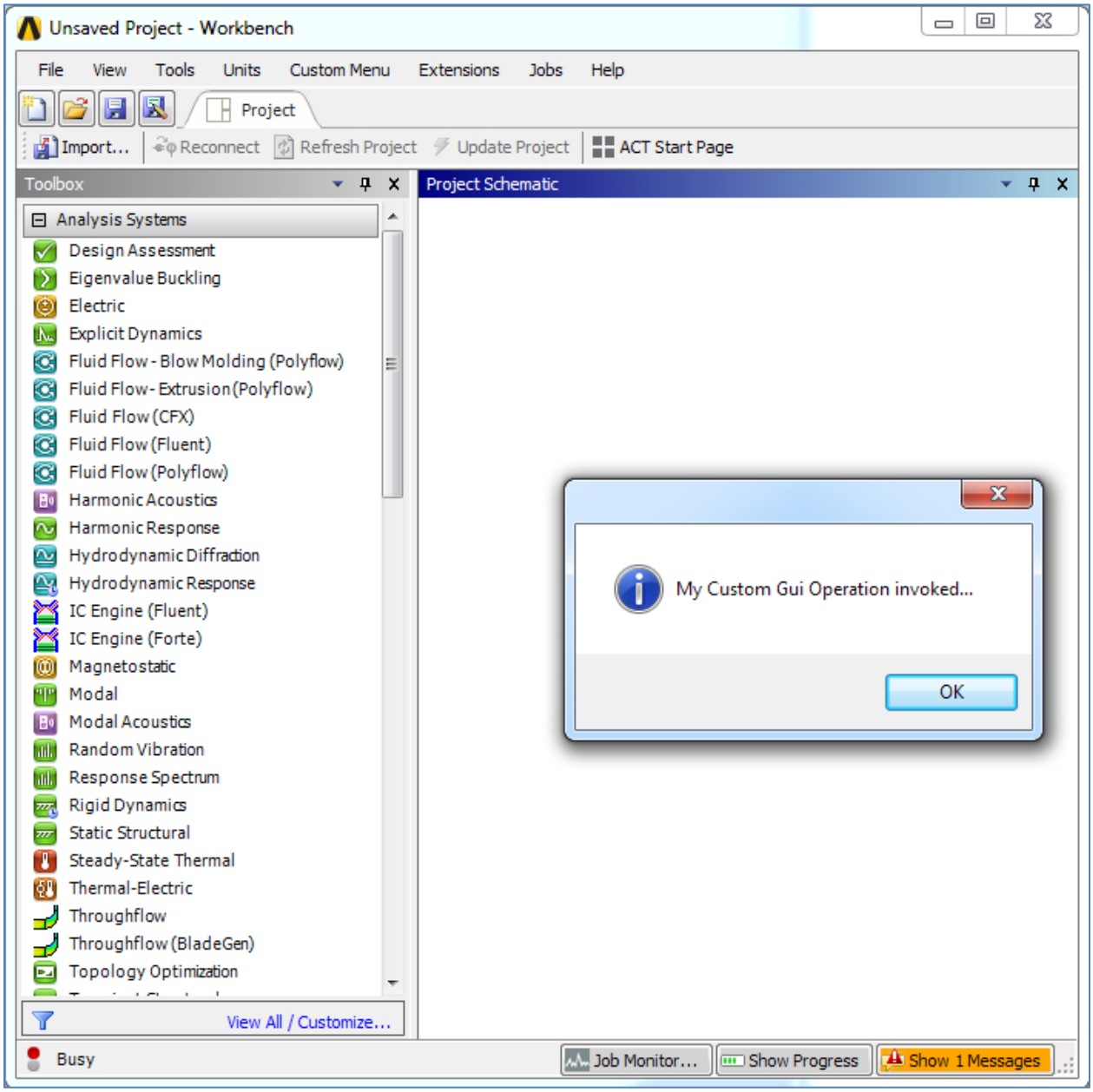
```
import System
import clr
clr.AddReference("Ans.UI")
import Ansys.UI
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit

def createMenu(context):
  uiMgr = Ansys.UI.UIManager.Instance
  menuMgr = uiMgr.MenuManager
  imgLib = Ansys.UI.Toolkit.ImageLibrary()
  for imgDir in ExtAPI.Extension.GetImageDirectories():
    imgLib.AddImagesFromDirectory(imgDir)
  menuLoc = System.Collections.Generic.List[System.String]()
  menuLoc.Add("Custom Menu")
  menuMgr.AddDynamicEntity(Ansys.UI.MenuEntityType.MenuEntry, "Custom
Entry...", "customEntryIcon", imgLib, System.Single(1.0), menuLoc,
```

```
MyCustomGuiOperation())

def removeMenu(context):

  uiMgr = Ansys.UI.UIManager.Instance

  menuMgr = uiMgr.MenuManager


menuMgr.RemoveDynamicEntity("DynamicEntity:Menu:System.Collections.Generic.L
ist`1[System.String]:Custom Entry...")

class MyCustomGuiOperation(Ansys.UI.Interfaces.IGuiOperation):

  def Invoke(self, context):

    Ansys.UI.Toolkit.MessageBox.Show("My Custom Gui Operation invoked...")

  def GuiItemCallBack(self, context):

    context.Visible = True

    context.Enabled = True
```

Workbench displays the new entry when you load your extension:



Clicking the entry displays a message dialog box:

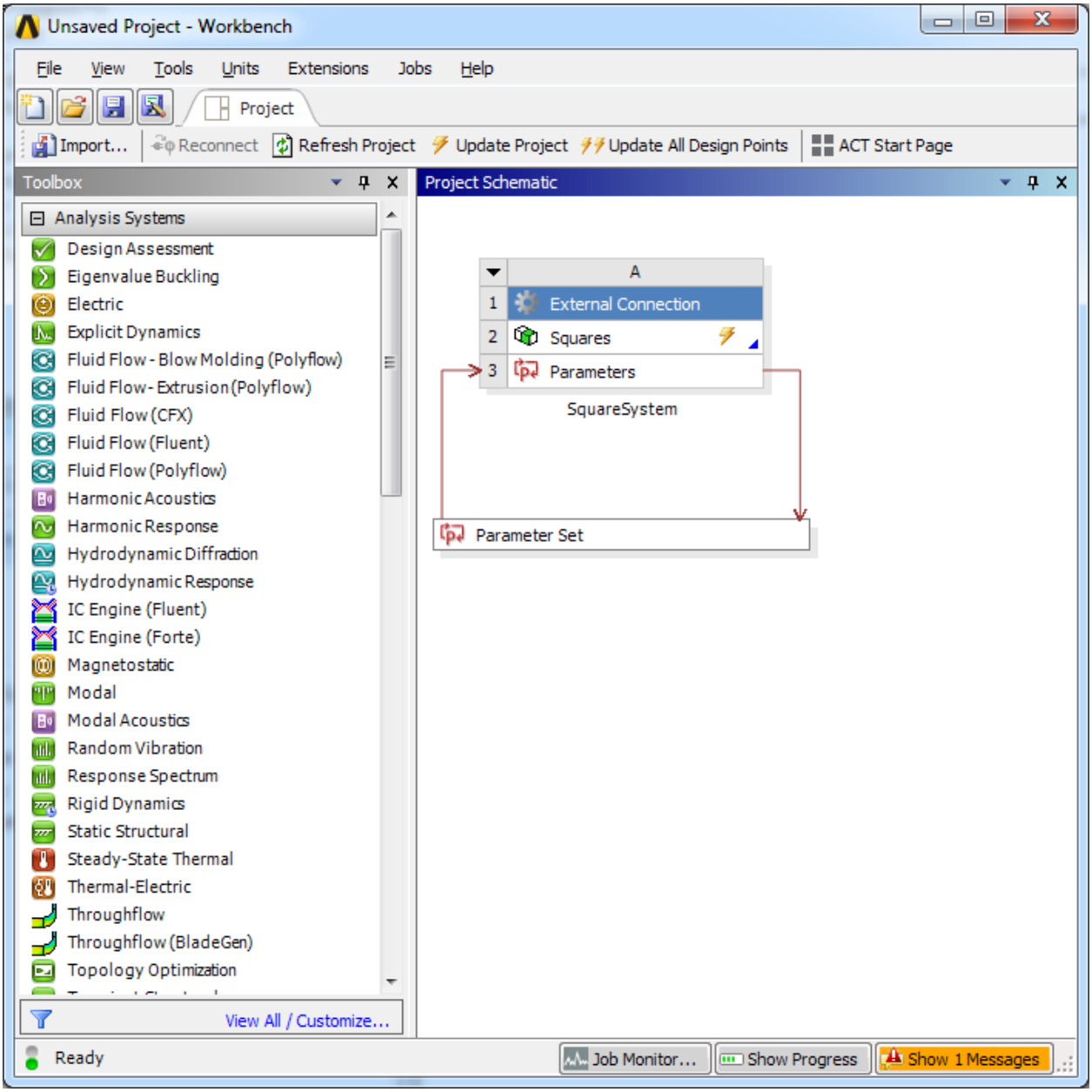


Integrating an External Application

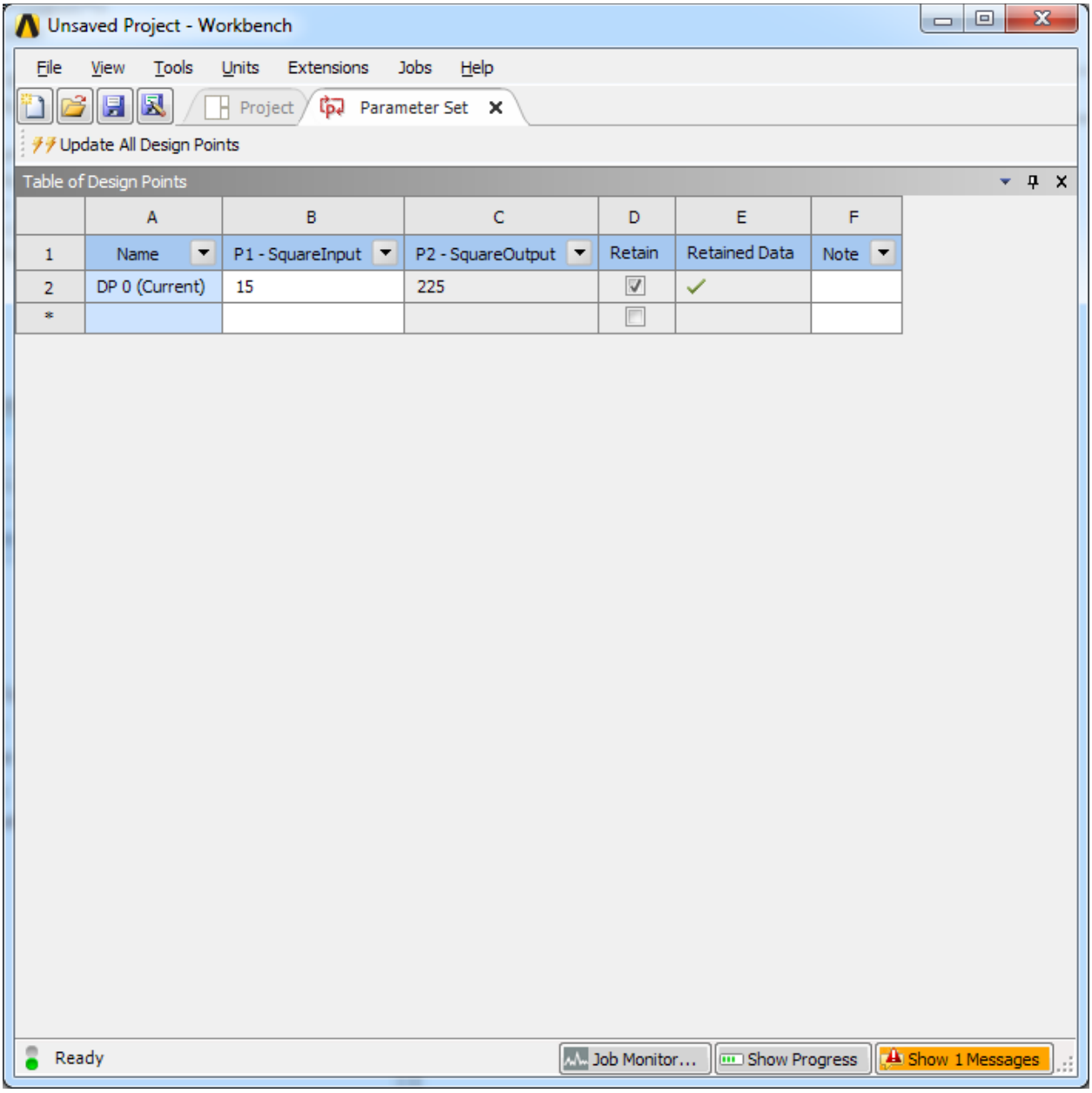
The External Connection Add-in allowed you to access external applications from within the **Project Schematic**. An installed **External Connection** system, primed with a configuration file, could run your application and engage parametric study. For example, a configuration file can be used to integrate a `Squares` application, which takes an input, squares it, and writes out the value:

```
<Configuration CellName="Squares" SystemName="SquareSystem" Version=""
ShowEditConfiguration="True">
  <Instructions WorkingDirectory="path_to_working_directory">
    <Instruction Type="Init">
      <Name></Name>
      <Args></Args>
      <ParameterParsingRules>
        <Parameter Name="SquareInput" Type="Input">
          <Rule Type="File">input.txt</Rule>
          <Rule Type="StartLine">1</Rule>
          <Rule Type="PreString">input=</Rule>
          <Rule Type="DataType">Double</Rule>
        </Parameter>
        <Parameter Name="SquareOutput" Type="Output">
          <Rule Type="File">output.txt</Rule>
          <Rule Type="StartLine">1</Rule>
          <Rule Type="PreString">output=</Rule>
          <Rule Type="DataType">Double</Rule>
        </Parameter>
      </ParameterParsingRules>
    </Instruction>
    <Instruction Type="Update">
      <Name></Name>
      <ExePath>path_to_executable/Squares.exe</ExePath>
      <Args></Args>
    </Instruction>
  </Instructions>
</Configuration>
```

After creating an **External Connection** system in Workbench and setting the configuration file, the **Project Schematic** would look like this:



On updating the `Squares` component, the **Tables** view for the **Parameter Set** bar would display a design point as the output:



With ACT, you define a new custom *task* and *task group* instead of configuring a prebuilt system and component. The extension file contains:

```
<extension version="1" name="Squares" icon="images\squares_extension.png">>
  <guid shortid="Squares">69d0095b-e138-4841-a13a-de12238c83f4</guid>
```

```
  <script src="squares.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
   <workflow name="wf1" context="Project" version="1">
    <tasks>
      <task name="Squares" caption="Squares" icon="squares_component"
version="1">
        <callbacks>
          <onupdate>update</onupdate>
        </callbacks>
        <inputs>
          <input/>
        </inputs>
        <outputs/>
        <parameters>
          <parameter name="Input" caption="Input" usage="Input"
control="float" version="1"/>
          <parameter name="Output" caption="Output" usage="Output"
control="float" version="1"/>
        </parameters>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="Squares" caption="Squares" icon="squares"
category="ACT Workflows" abbreviation="SQRS" version="1">
        <includeTask name="Squares" caption="Squares"/>
      </taskgroup>
    </taskgroups>
   </workflow>
</extension>
```

The key difference with ACT: the script `squares.py` and the update callback handle all text parsing, application execution, and parameter updates. The External Connection Add-in was plagued by

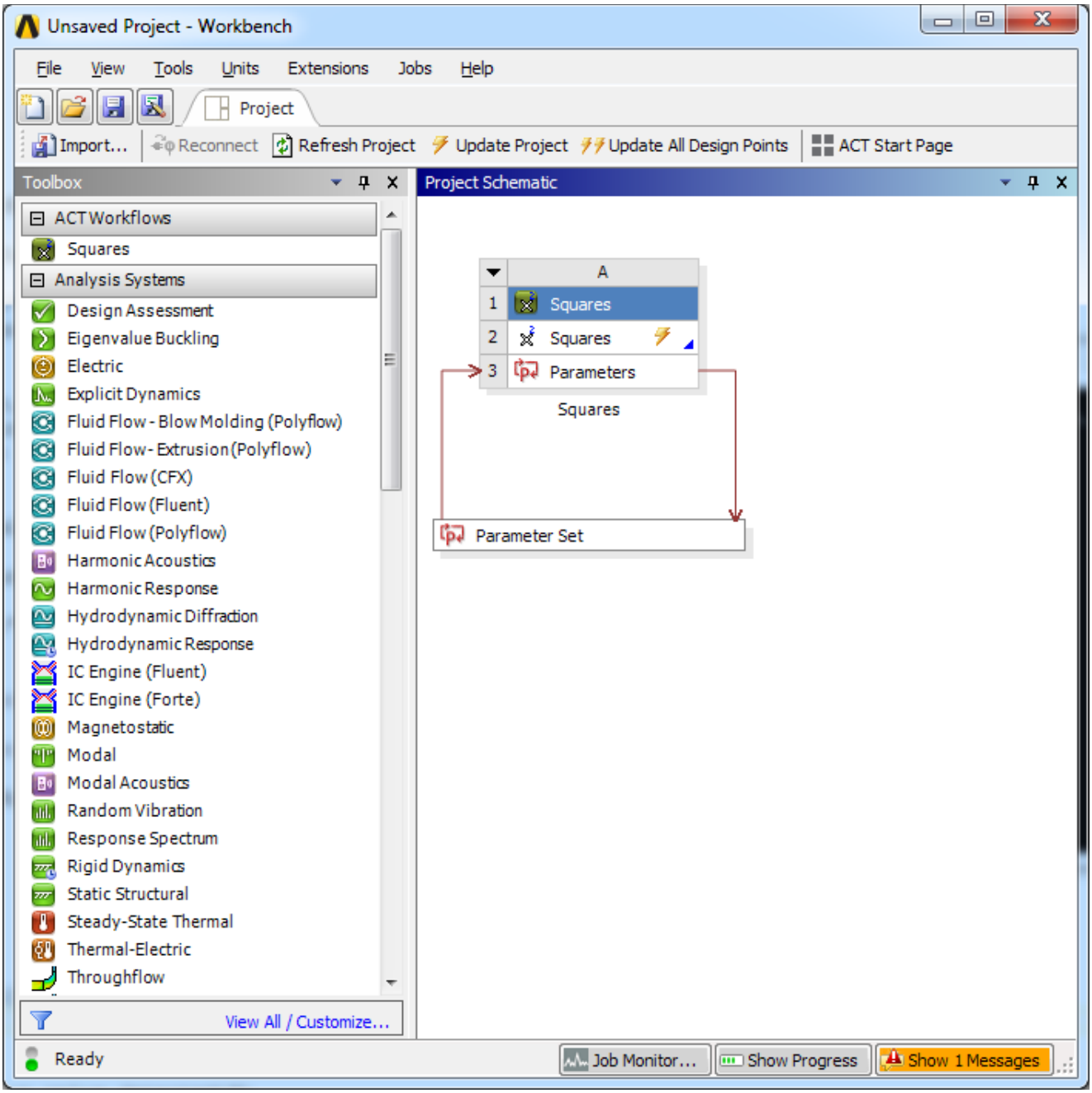
inflexibility and portability issues. By rebalancing the responsibility between the definition and implementation files, you can achieve greater reliability in your customization solutions. The script `squares.py` contains the following:

```
import System
import clr
clr.AddReference("Ans.Utilities")
import Ansys.Utilities


def GetParameterByName(task, id):
  match = None
  parameters = task.Parameters
  for param in parameters:
    if param.DisplayText == id:
      entityPropsDict =
param.GetAssociatedEntityProperties(Container=task.InternalObject)
      for dataRef in entityPropsDict:
        match = dataRef
        break
      break
  return match
def update(task):
  activeDir = task.ActiveDirectory
  extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
  exeName = "ExampleAddinExternalSolver.exe"
  solverPath = System.IO.Path.Combine(extensionDir, exeName)
  inputParam = GetParameterByName(task, "Input")
  outputParam = GetParameterByName(task, "Output")
  inputFileName = "input.txt"
  outputFileName = "output.txt"
  dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)
  dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)
  if inputParam != None and outputParam != None:
```

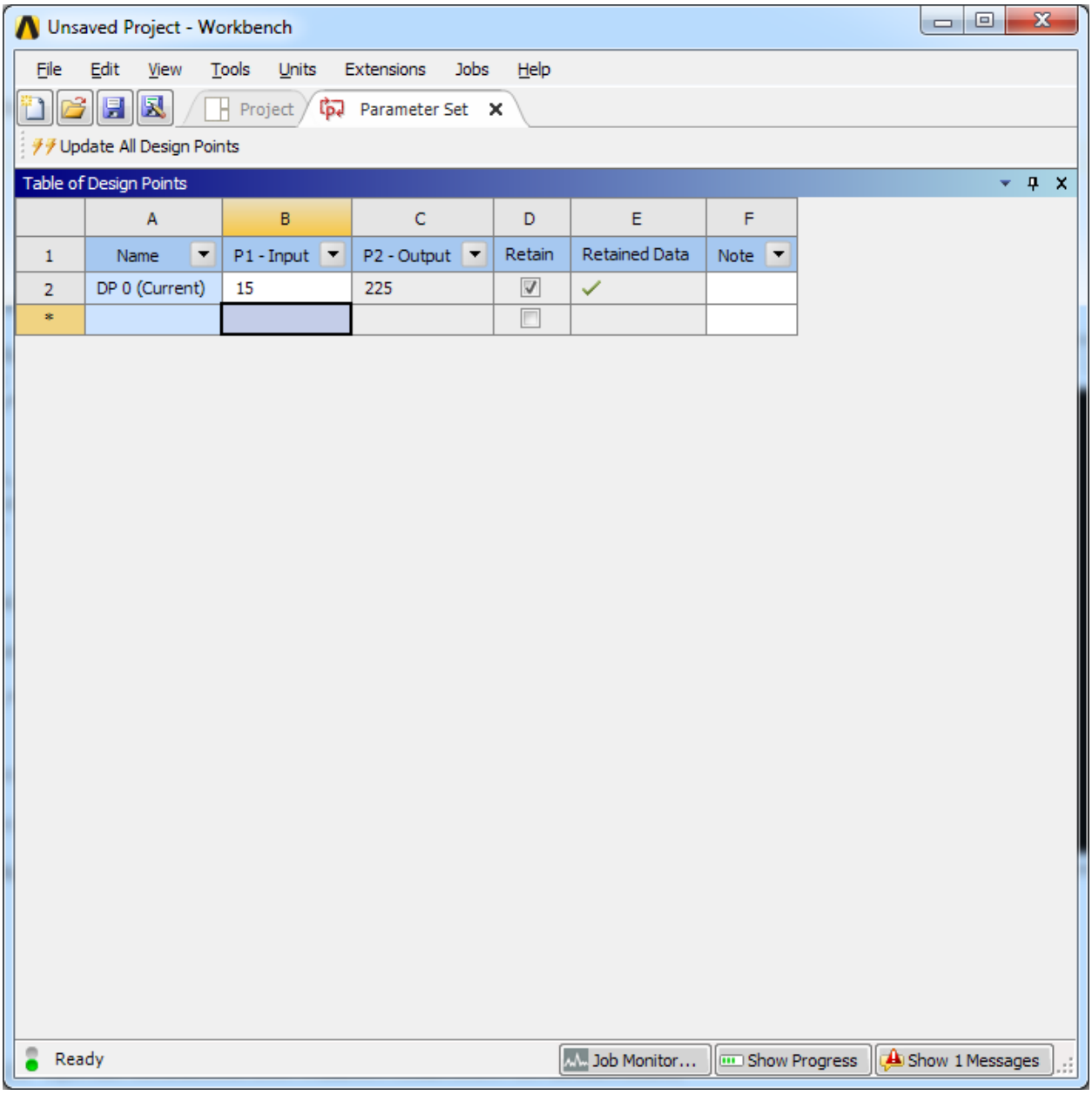
```
    val = inputParam.Value

    f = open(dpInputFile, "w")


f.write('input='+val.ToString(System.Globalization.NumberFormatInfo.Invarian
tInfo))

    f.close()

  runInMono =
Ansys.Utilities.ApplicationConfiguration.DefaultConfiguration.IsRuntimeMono

  monoPath = "mono"

  monoArgs = System.String.Format("{0} \"{1}\" \"{2}\"", solverPath,
dpInputFile, dpOutputFile)

  info = None

  if runInMono:

    info = System.Diagnostics.ProcessStartInfo(monoPath, monoArgs)

  else:

    info = System.Diagnostics.ProcessStartInfo(solverPath,
System.String.Format("\"{0}\" \"{1}\"", dpInputFile, dpOutputFile))

  info.CreateNoWindow = True

  info.WindowStyle = System.Diagnostics.ProcessWindowStyle.Minimized

  p = System.Diagnostics.Process.Start(info)

  p.WaitForExit()

  outValue = None

  f = open(dpOutputFile, "r")

  currLine = f.readline()

  while currLine != "":

    valuePair = currLine.split('=')

    outValue = System.Double.Parse(valuePair[1],
System.Globalization.NumberFormatInfo.InvariantInfo)

    currLine = f.readline()

  f.close()

  if outValue == None:

    raise Exception("Error in update - no output value detected!")

  else:
```

```
    outputParam.Value = outValue
```

The result in Workbench, after loading the extension, would look like this:



Updating the `Squares` component results in a similar design point result:



Creating Custom Systems and Components

Some customization solutions required custom systems and components. Using new system definitions, you could configure custom systems before runtime and expose them as installed **Toolbox** entries. Custom components could establish upstream and downstream connections and participate in Workbench data transfer. A custom system project contained two files: a system definition file and a component configuration file.

The system definition file supplied the External Connection Add-in with component, input, and output information:

```
<System name="GenericMeshTransfer" displayText="Generic Mesh"
abbreviation="GenMeshXfr" imageName="GenericMesh">
  <Components>
    <Component name="Mesher" displayText="Mesher"
imageName="GenericMesh_cell">
      <Inputs>
        <Input/>
        <Input dataType="FluentMesh">MeshingMesh</Input>
      </Inputs>
      <Outputs>
        <Output dataType="FluentMesh">SimulationGeneratedMesh</Output>
      </Outputs>
    </Component>
  </Components>
</System>
```

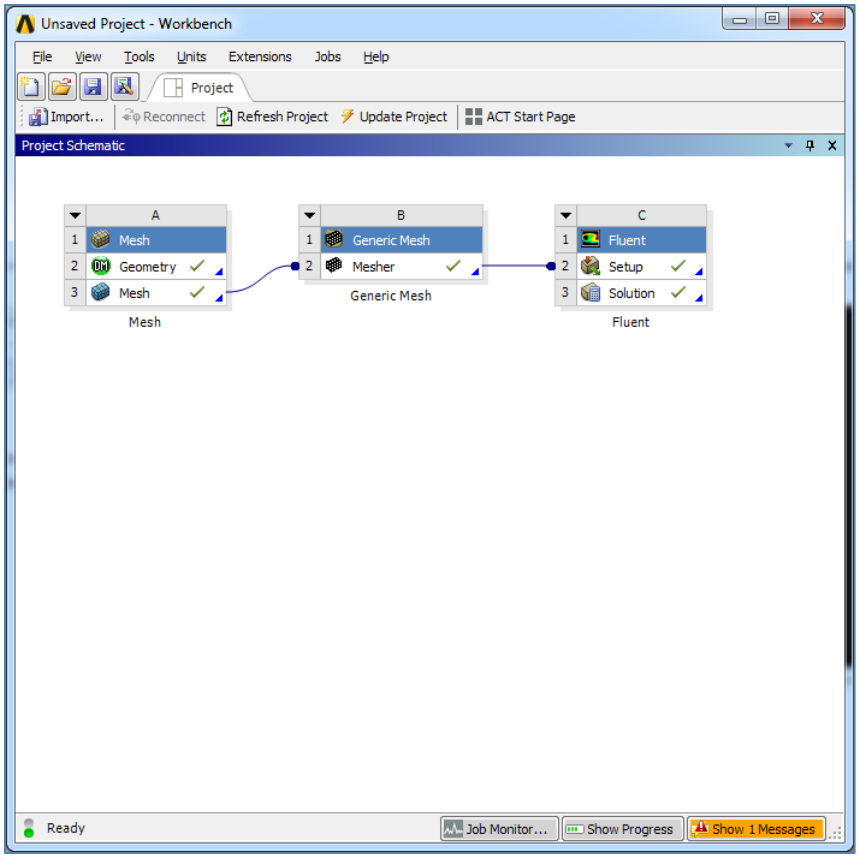
The configuration file provided behavior details:

```
<Configuration ShowEditConfiguration="False">
  <Instructions>
    <Instruction Type="Update">
      <Name></Name>
      <Script>path_to_python_file/test.py</Script>
    </Instruction>
  </Instructions>
</Configuration>
```

The Python script performed the command calls for the component update:

```
container = ExternalConnectionSystemContainer
upstreamData = container.GetInputDataByType(InputType="MeshingMesh")
meshFileRef = None
upstreamDataCount = upstreamData.Count
if upstreamDataCount > 0:
  meshFileRef = upstreamData[0]
  outputRefs = container.GetOutputData()
  meshOutputSet = outputRefs["SimulationGeneratedMesh"]
  meshOutput = meshOutputSet[0]
  meshOutput.TransferFile = meshFileRef
```

You placed the system definition, configuration, script and image files in an install location at `%AWP_ROOTXXX%\Addins\ExternalConnection\SystemDefinitions`. Once copied, you could access the system from Workbench and load the component configuration file:



With ACT, you define the entire solution in one configuration file. No file assignment is required.

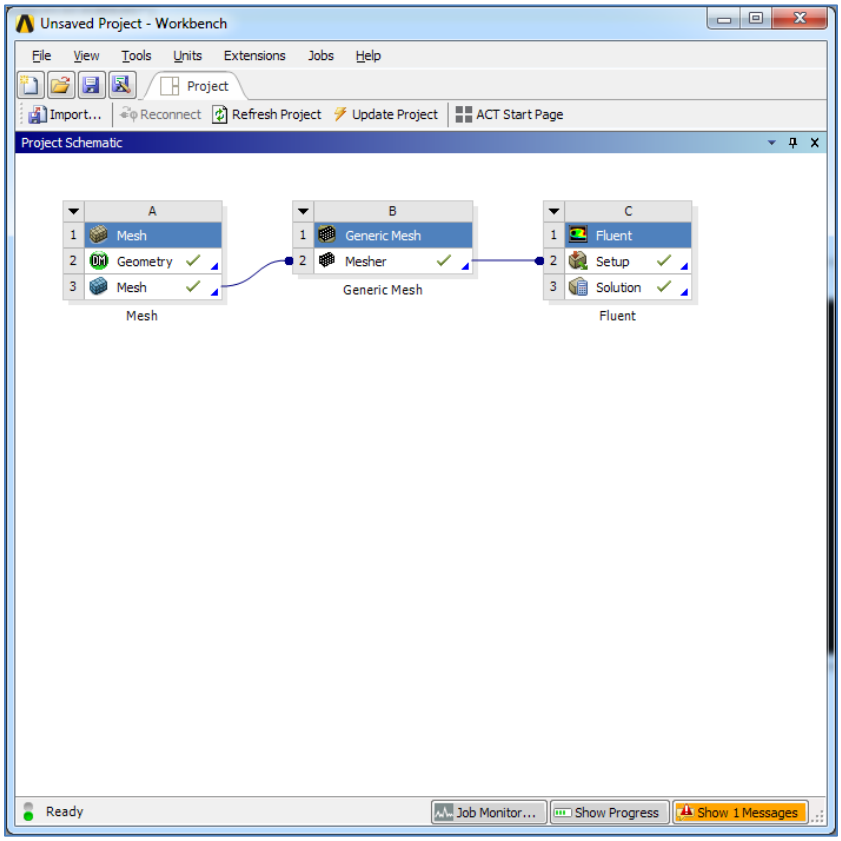
```
<extension version="1" name="GenericMeshTransfer"
icon="images\GenericMesh_extension.png">

  <guid shortid="GenericMeshTransfer">69d0095b-e138-4841-a13a-
de12238c83f7</guid>

  <script src="main.py" />

  <interface context="Project">

    <images>images</images>

  </interface>

   <workflow name="wf1" context="Project" version="1">

    <tasks>

      <task name="Mesher" caption="Mesher" icon="GenericMesh_cell"
version="1">

        <callbacks>

          <onupdate>update</onupdate>

        </callbacks>

        <inputs>

          <input format="FluentMesh" type="MeshingMesh" count="1"/>

          <input/>

        </inputs>

        <outputs>

          <output format="FluentMesh" type="SimulationGeneratedMesh"/>

        </outputs>

      </task>

    </tasks>

    <taskgroups>

      <taskgroup name="GenericMeshTransfer" caption="Generic Mesh"
icon="GenericMesh" category="ACT Workflows" abbreviation="GenMeshXfer"
version="1">

        <includeTask name="Mesher" caption="Mesher"/>

      </taskgroup>

    </taskgroups>

   </workflow>
```

```
</extension>
```

As before, the IronPython script provides the script calls:

```
def update(task):
  upstreamData = task.InputData["MeshingMesh"]
  meshFileRef = None
  upstreamDataCount = upstreamData.Count
  if upstreamDataCount > 0:
    meshFileRef = upstreamData[0]
    meshOutputSet = task.OutputData["SimulationGeneratedMesh"]
    meshOutput = meshOutputSet[0]
    meshOutput.TransferFile = meshFileRef
```

Once you load the extension in Workbench and construct the workflow, the **Project Schematic** contains the same result:



Additionally, ACT provides a convenient process for defining custom context menus on custom components. Many tasks offer a default **Edit** operation that is triggered when you either double-click the left mouse button or right-click and select **Edit** from the context menu. You can define an `<onedit>` task callback to automatically receive an **Edit** entry on the context menu:

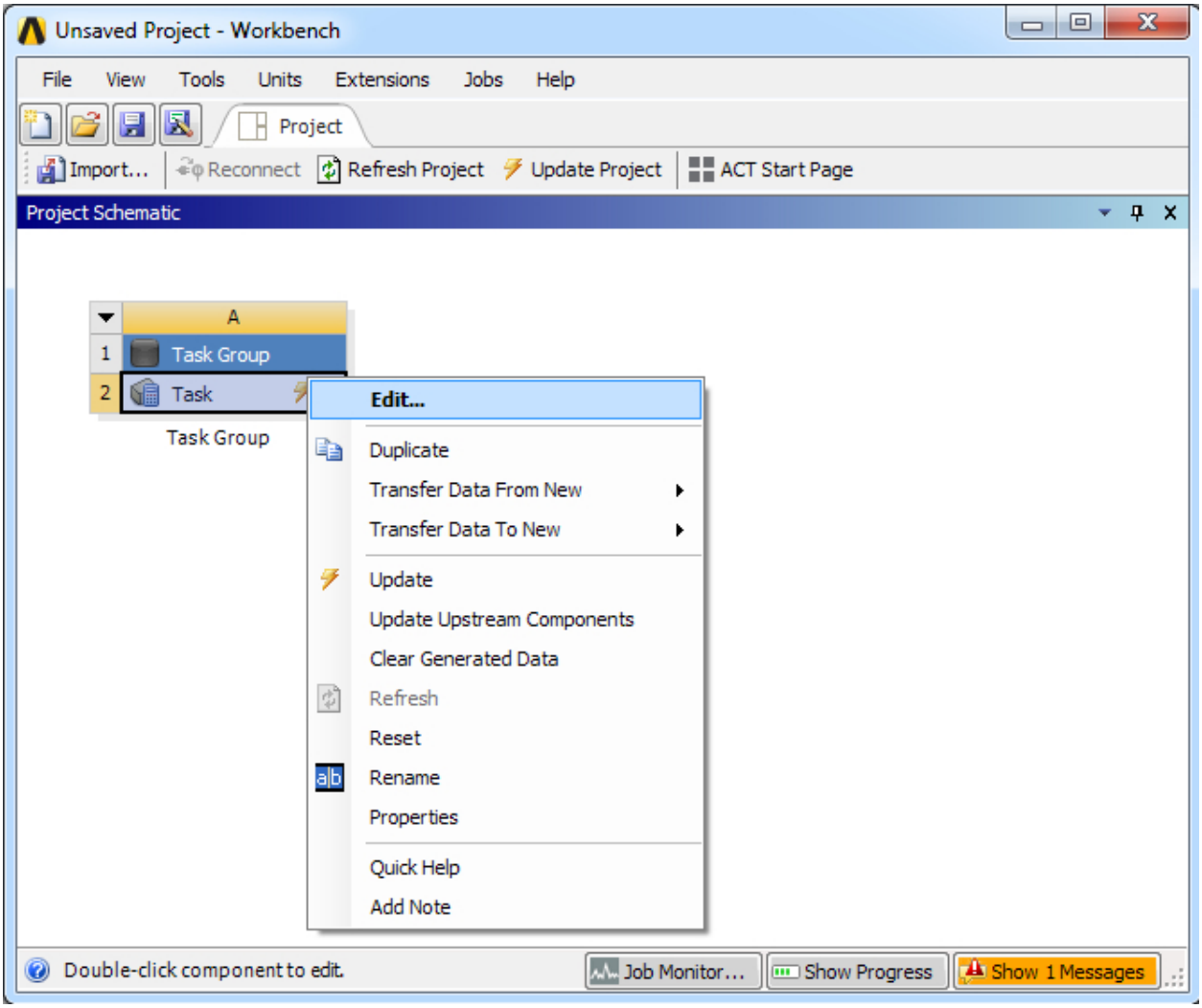
Extension File

```
<task name="Task" caption="Task" icon="task" version="1">
  <callbacks>
    <onupdate>update</onupdate>
    <onedit>edit</onedit>
  </callbacks>
  <inputs>
    <input/>
  </inputs>
  <outputs/>
</task>
```

IronPython

```
def edit(task):
  ExtAPI.Log.WriteMessage("edit...")
```

Project Schematic



You can specify additional task-scoped context menus by declaring entry items within a context menu block on a task:

Extension File

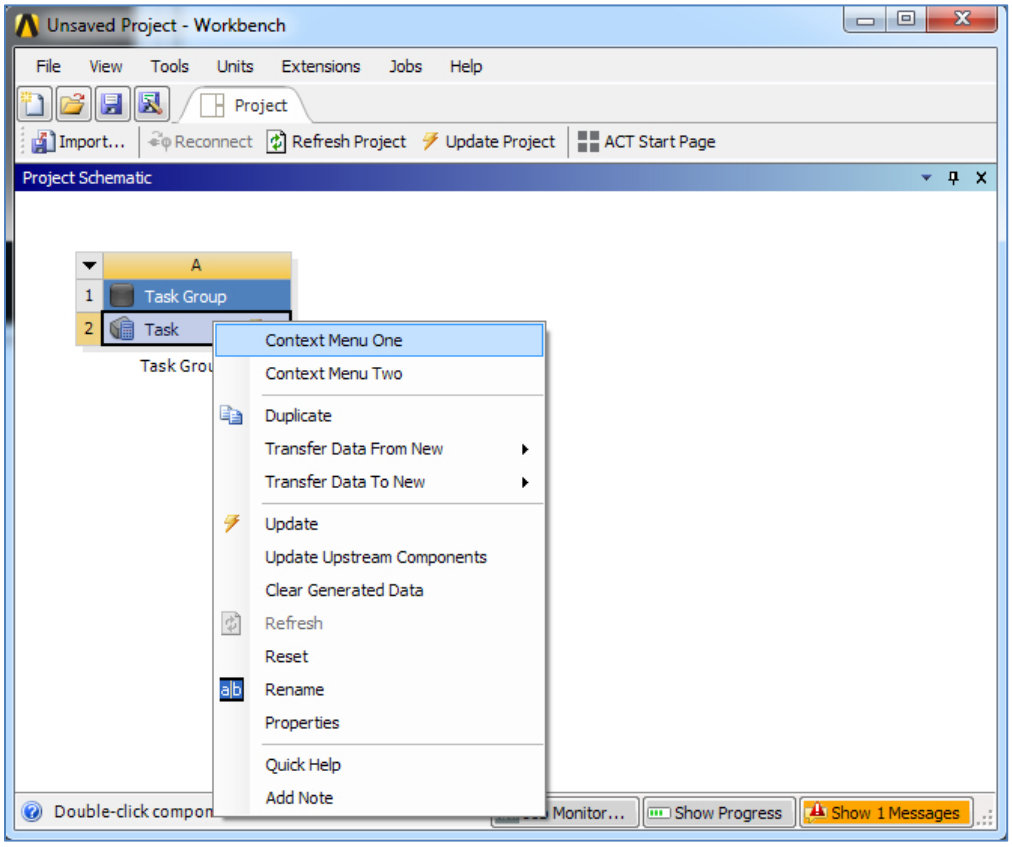
```
<task name="Task" caption="Task" icon="task" version="1">
  <callbacks>
    <onupdate>update</onupdate>
  </callbacks>
  <inputs>
    <input/>
```

```
  </inputs>
  <outputs/>
  <contextmenus>
    <entry name="ContextMenu1" caption="Context Menu One" icon=""
version="1" priority="1.">
      <callbacks>
        <onclick>cm1</onclick>
      </callbacks>
    </entry>
    <entry name="ContextMenu2" caption="Context Menu Two" icon=""
version="1" priority="1.">
      <callbacks>
        <onclick>cm2</onclick>
      </callbacks>
    </entry>
  </contextmenus>
</task>
```

IronPython

```
def cm1(task):
  ExtAPI.Log.WriteMessage("context menu 1...")
def cm2(task):
  ExtAPI.Log.WriteMessage("context menu 2...")
```

Project Schematic



SDK

Some customization projects require more nuanced control of ANSYS Workbench. Your solution before 19.0 -- the ANSYS Workbench Software Development Kit (SDK). However, prohibiting most users from realizing its potential were SDK's prerequisites: advanced programming experience, a commercial Integrated Development Environment, and separate development module. You followed the same development process as ANSYS' own application integration efforts. Whether you wanted to "data-integrate" your application or expose new user interface entries, you created a compiled Workbench add-in.

Add-in Versus Extension

An SDK Add-in required a configuration XML file, associated add-in folder, compiled C# assemblies, and resource files. These files were placed within the *Addins* folder of your ANSYS installation. The configuration file defined the load instructions for Workbench to automatically initialize your add-in:

```
<?xml version="1.0" encoding="utf-8" ?>
<Addins>
 <Configuration Name="Scratch">
  <Addin Name="Ansys.ScratchAddin.Addin"
Location="$(Addins)/ScratchAddin/bin/$(Albion_Running_Platform)/$(Albion_Run
ning_Target)/Ansys.ScratchAddin.dll"/>
  <IncludeConfig Name="Base" />
 </Configuration>
 <Configuration Name="Default">
  <Addin Name="Ansys.ScratchAddin.Addin"
Location="$(Addins)/ScratchAddin/bin/$(Albion_Running_Platform)/$(Albion_Run
ning_Target)/Ansys.ScratchAddin.dll"/>
 </Configuration>
</Addins>
```

An ACT extension file contains all definition information for your customization solution. The extension attributes `loadAsDefault` and `isNative` allow you to obtain the same auto-load functionality as SDK add-ins:

```
<extension loadAsDefault="True" isNative="True" version="1" name="Scratch"
icon="images\scratch.png">
  <guid shortid="Scratch">3c6d8f67-13c7-4049-90c8-e547f988f9db</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>start</oninit>
      <onterminate>end</onterminate>
    </callbacks>
  </interface>
```

```
</extension>
```

As long as you place your extension in one of the three Workbench-aware search paths, the extension load process will resemble an SDK Add-in.

Functionality

SDK-based projects typically incorporated one or more of the SDK's three core features:

- User Interface Customization
- Data and Action Processing
- Workflow and Application Integration

User Interface Customization

As with the External Connection Add-in, you may have used the SDK to add new user interface entries to Workbench. You defined custom `IGuiOperation` implementations for custom toolbar, menu bar, and context menu entries. You could also interact with progress monitoring, custom user interface views, and property display.

GUI Operations

You can define arbitrary user interface entries with ACT. Past SDK implementations would contain the following:

```
[GuiOperation("Example GUI Operation")]
[ContextMenuEntry]
[MenuEntry]
[ToolbarButton]
[Visibility(AttributeOptions.InitialEnablingOff |
AttributeOptions.InitialVisibilityOff | AttributeOptions.UseCallback)]
public class ExampleGuiOperation : IGuiOperation
{
  public void Invoke(GuiOperationContext context)
  {
  }

  public void GuiItemCallBack(GuiInformationContext context)
```

```
  {
  }
}
```

Your SDK add-in would register the GUI operations as a *Load* step:

```
public class Addin : Ansys.Core.Addins.AddinBase, IDefineGui
{
  public void Load(Ansys.UI.GuiDefineContext context)
  {
    //register all GUI operations defined in this Addin
    context.RegisterAllFromExecutingAssembly();
  }
}
```

With ACT, you define these entries dynamically from the extension's User Interface initialize and terminate callbacks as well as at the interface level (for toolbars):

```
<extension version="1" name="GuiOperations" icon="images\guiOperations.png">
  <guid shortid="GuiOperations">886bfb1d-dd9c-42fa-9fcf-c86ccc73bfbc</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>createOperations</oninit>
      <onterminate>removeOperations</onterminate>
    </callbacks>
   <toolbar name="MyToolbar" caption="My Toolbar">
   <entry name="Button1" caption="Button One" icon="mybutton1">
    <callbacks>
     <onclick>button1Click</onclick>
    </callbacks>
```

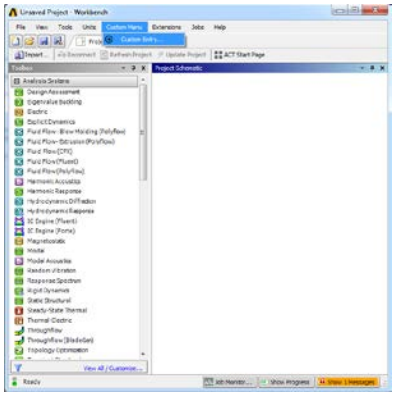
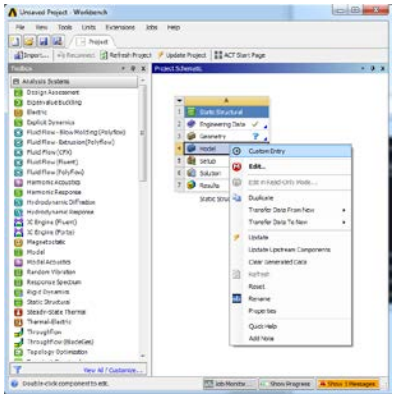
```
   </entry>
   </toolbar>
  </interface>
</extension>
```

You access the MenuManager in your callbacks to manage the custom entries:

```
import System
import clr
clr.AddReference("Ans.UI")
import Ansys.UI
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit


def createOperations(context):
  uiMgr = Ansys.UI.UIManager.Instance
  menuMgr = uiMgr.MenuManager
  imgLib = Ansys.UI.Toolkit.ImageLibrary()
  for imgDir in ExtAPI.Extension.GetImageDirectories():
    imgLib.AddImagesFromDirectory(imgDir)
  menuLoc = System.Collections.Generic.List[System.String]()
  menuLoc.Add("Custom Menu")
  menuMgr.AddDynamicEntity(Ansys.UI.MenuEntityType.MenuEntry, "Custom
Entry...", "customEntryIcon", imgLib, System.Single(1.0), menuLoc,
MyCustomGuiOperation())
  menuMgr = uiMgr.ContextMenuManager
  menuMgr.AddDynamicEntity(None, Ansys.UI.MenuEntityType.MenuEntry, "Custom
Entry", False, "customEntryIcon", imgLib, System.Single.Parse('1.0'), None,
MyCustomGuiOperation(), True, False, Ansys.UI.OpClass.None, False, True,
True)


def removeOperations(context):
  uiMgr = Ansys.UI.UIManager.Instance
```

```
  menuMgr = uiMgr.MenuManager
menuMgr.RemoveDynamicEntity("DynamicEntity:Menu:System.Collections.Generic.List`1[System.String]:Custom Entry...")
  menuMgr = uiMgr.ContextMenuManager
  menuMgr.RemoveDynamicEntity("DynamicEntity:ContextMenu:Custom Entry")


class MyCustomGuiOperation(Ansys.UI.Interfaces.IGuiOperation):
  def Invoke(self, context):
    Ansys.UI.Toolkit.MessageBox.Show("My Custom Gui Operation invoked...")
  def GuiItemCallBack(self, context):
    context.Visible = True
    context.Enabled = True
```

The **Project Schematic** results are similar to those shown in External Connection Add-in:

Progress Monitoring

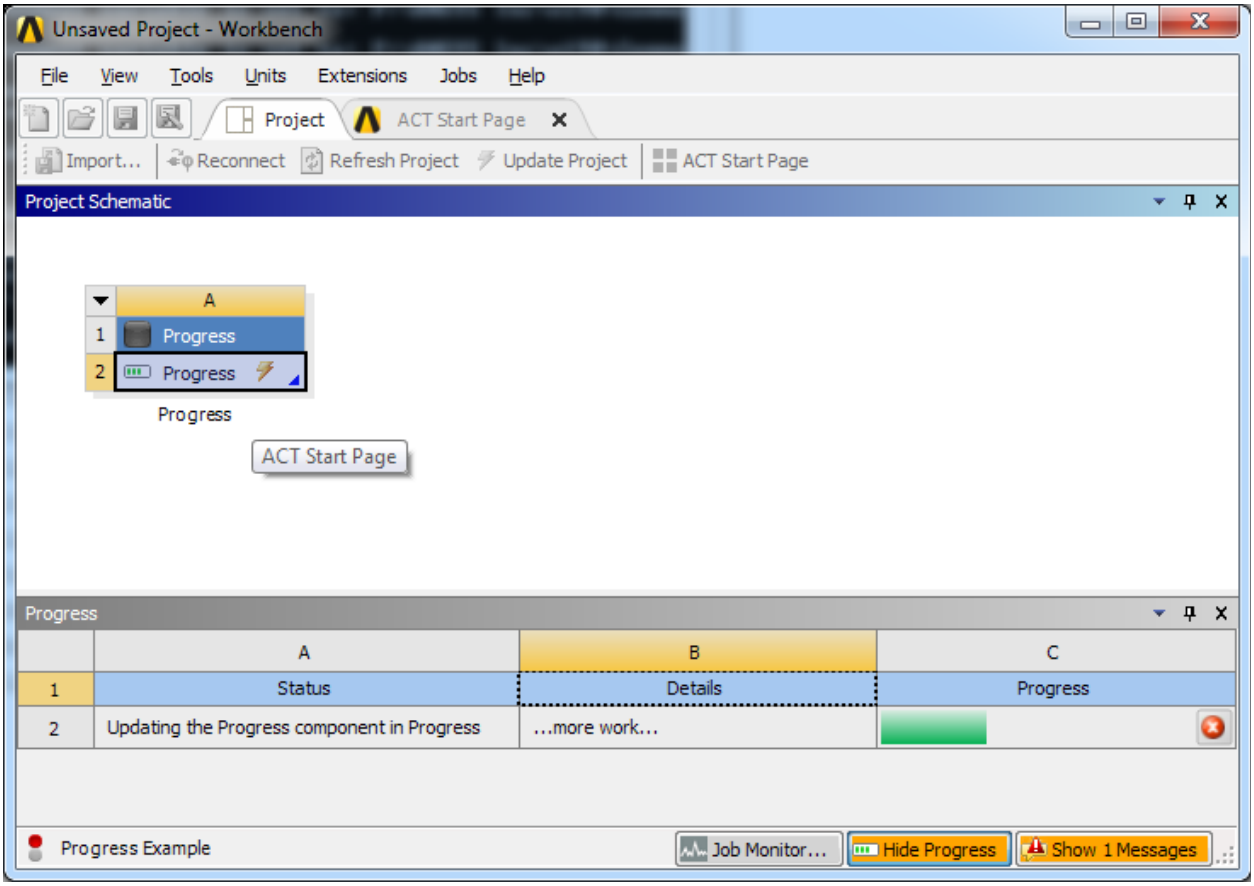
Within an ACT callback, you can access the Progress Monitor to report message updates to the user:

IronPython

```
def update(task):
  monitor = ExtAPI.UserInterface.ProgressMonitor
  monitor.BeginWork("Progress Example", 3)
  monitor.WorkDetails = "Work started!"
```

```
System.Threading.Thread.Sleep(2000)
monitor.UpdateWork(1)
monitor.WorkDetails = "...more work..."
System.Threading.Thread.Sleep(2000)
monitor.UpdateWork(1)
monitor.WorkDetails = "...almost done..."
System.Threading.Thread.Sleep(2000)
monitor.UpdateWork(1)
monitor.WorkDetails = "Work completed!"
System.Threading.Thread.Sleep(2000)
monitor.EndWork()
```

Project Schematic



Custom User Interface Views

Some advanced projects require custom tabs and views within the Workbench **Project Schematic**. The advanced nature of these custom elements require lower-level code use. As before, you exercise the extension's user interface initialize and terminate callbacks:

Extension File

```
<extension version="1" name="CustomTab" icon="images\tab.png">
 <guid shortid="CustomTab">80FE32DB-1FEB-47B2-9782-6B5A022F9D99</guid>
 <script src="main.py" />
 <interface context="Project">
  <images>images</images>
  <callbacks>
   <oninit>init</oninit>
  </callbacks>
  <toolbar name="CustomTab" caption="CustomTab">
   <entry name="Custom Tab" caption="Custom Tab" icon="ansys">
    <callbacks>
     <onclick>showTab</onclick>
    </callbacks>
   </entry>
  </toolbar>
 </interface>
</extension>
```

IronPython

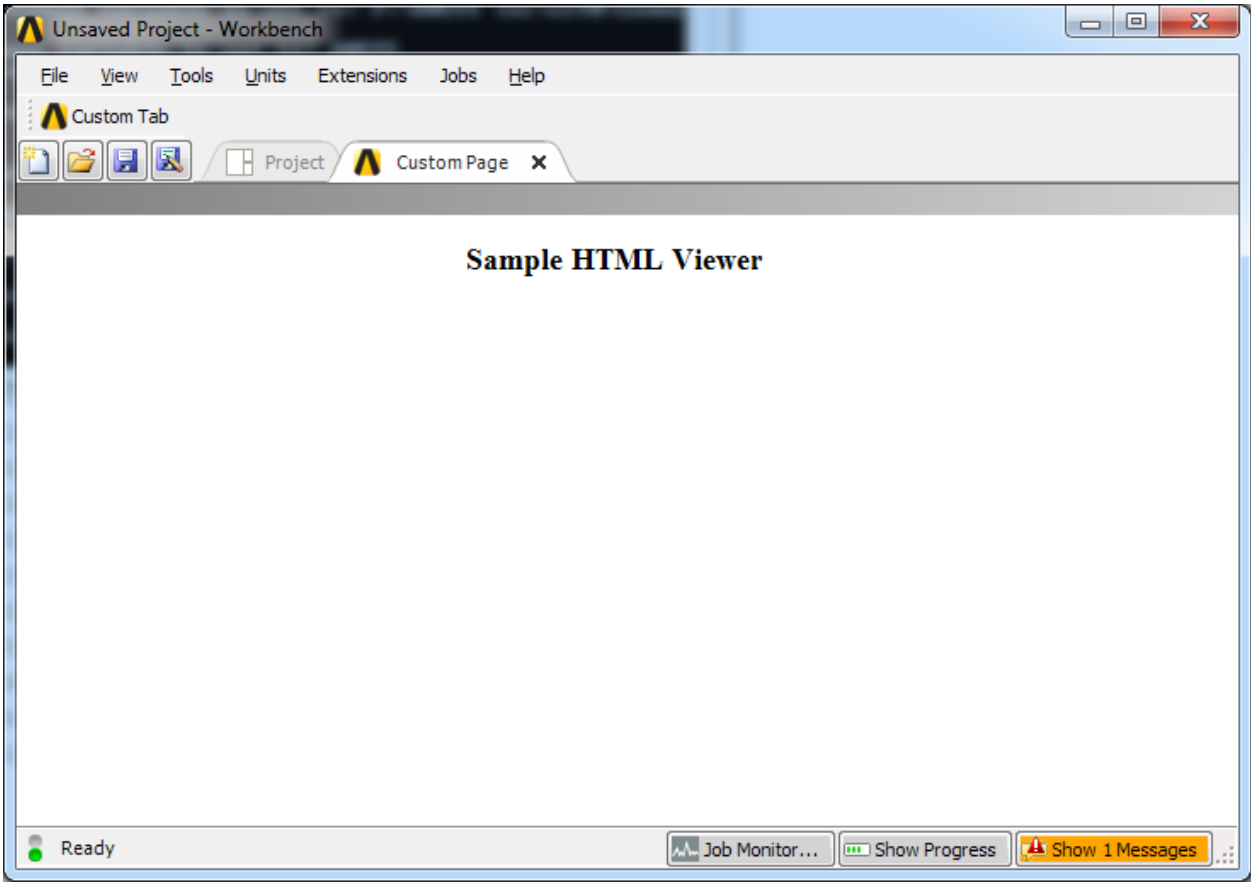
```
import System
import clr
clr.AddReference("Ans.UI")
import Ansys.UI
clr.AddReference("Ans.Utilities")
import Ansys.Utilities
```

```
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit
import Ansys.UI.Toolkit.Base


def init(context):
  ExtAPI.Log.WriteMessage("Init Scratch Extension ...")
  uiManager = Ansys.UI.UIManager.Instance
  t = uiManager.GetType()
  p = t.GetProperty("ViewRegistry", System.Reflection.BindingFlags.Instance
| System.Reflection.BindingFlags.NonPublic)
  registry = p.GetValue(uiManager)
  t = registry.GetType()
  m = t.GetMethod("RegisterView", System.Reflection.BindingFlags.Instance |
System.Reflection.BindingFlags.NonPublic)
  args = System.Collections.Generic.List[System.Object]()
  args.Add(clr.GetClrType(CustomView))
  m.Invoke(registry, args.ToArray())
def showTab(entity):
  global _p
  ExtAPI.Log.WriteMessage("Button clicked...")
  w = Ansys.UI.UIManager.Instance.GetActiveWorkspace()
  v = w.GetView("CustomView")
  if v == None:
    builder = w.PanesBuilder
    customPane = builder.CreateCustomerPane(w, "Custom")
    customPane.AllowMinimized = False
    customPane.AllowFloating = False
    customPane.AllowDocking = False
    header = customPane.Header
    header.CloseVisible = False
    header.AuxDropDownVisible = False
```

```
    header.DropDownVisible = False

    header.PinVisible = False

    header.Active = False

    header.Enabled = False

    header.ArrangeVisible = False

    header.RestoreVisible = False

    v = CustomView("CustomView")

    addMethod = w.GetType().GetMethod("AddViewToPane",
System.Reflection.BindingFlags.Instance |
System.Reflection.BindingFlags.NonPublic)

    argsList = System.Collections.Generic.List[System.Object]()

    argsList.Add(customPane)

    argsList.Add(v)

    addMethod.Invoke(w, argsList.ToArray())

    layout = builder.CreateNewLayout("CustomLayout", customPane)

    header.Text = ""

  Ansys.UI.Workspaces.Workspace.ActivateLayout("CustomLayout", "", "Custom
Page", None)

class CustomView(Ansys.UI.Views.View):

  def __init__(self, name):

    self._mainPanel = None

    self.Initialize()

  def Initialize(self):

    if self._mainPanel == None:

      self._mainPanel = Ansys.UI.Toolkit.TableLayoutPanel()

    else:

      self._mainPanel.Controls.Clear()

      self._mainPanel.Rows.Clear()

      self._mainPanel.Columns.Clear()

    self._mainPanel.Rows.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
100)


self._mainPanel.Columns.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
```

```
100)
    self._mainPanel.BackColor = Ansys.Utilities.Color(83, 83, 83)
    htmlViewer = Ansys.UI.Toolkit.HtmlViewer()
    htmlViewer.MinimumSize = self._mainPanel.Size
    htmlViewer.Html = r"<html><head><title>Custom
Tab</title></head><body><center><h3>Sample HTML
Viewer</h3></center></body></html>"
    self._mainPanel.Controls.Add(htmlViewer, 0, 0)
    self.Control = self._mainPanel
```

Project Schematic



Property Display

You can use ACT to change the display of properties within the **Project Schematic**. For example, you can instruct a property to display a dialog box for user input instead of direct property manipulation.

Extension File

```
<extension version="1" name="CustomPropertyAction"
icon="images\buttonPress.png">
  <guid shortid="CustomPropertyAction">69d1234b-f235-4841-a13a-
de20538c83f2</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>registerPropertyAction</oninit>
    </callbacks>
  </interface>
</extension>
```

IronPython

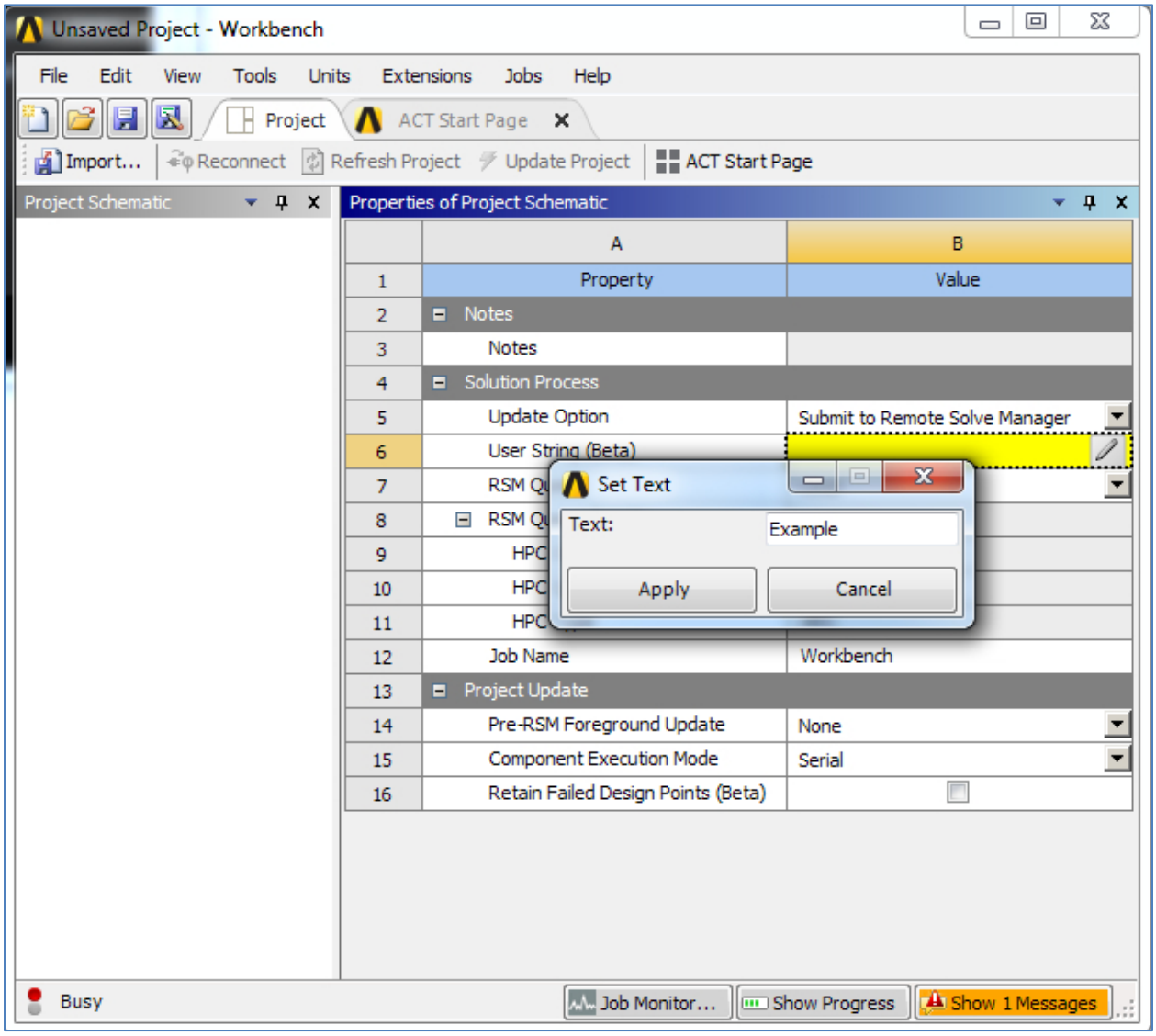
```
import System
import clr
clr.AddReference("Ans.Core")
import Ansys.Core
clr.AddReference("Ans.UI")
import Ansys.UI
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit
import Ansys.UI.Toolkit.Base
import Ansys.UI.Toolkit.Drawing


def registerPropertyAction(context):
  ExtAPI.Log.WriteMessage("Registering custom property action...")
  uiMgr = Ansys.UI.UIManager.Instance
  attributes = System.Collections.Generic.List[System.Attribute]()
  attr =
```

```
Ansys.UI.Attributes.GuiOperationAttribute("ShowMyCustomPropertyAction")

  attributes.Add(attr)

  defineContext = uiMgr.CreateGuiDefineContext()

  defineContext.RegisterOperation(MyCustomPropertyActionGuiOperation(),
attributes)


Ansys.UI.PropertyEditActionRegistry.RegisterAction("DesignPointSolveSettings
", "RsmUserString", "ShowMyCustomPropertyAction",
Ansys.UI.PropertyEditActionRegistry.ActivationMode.ByButton)


Ansys.UI.PropertyEditActionRegistry.RegisterAction("ComponentSolveSettingsFo
rAddin", "RsmUserString", "ShowMyCustomPropertyAction",
Ansys.UI.PropertyEditActionRegistry.ActivationMode.ByButton)

class MyCustomPropertyActionGuiOperation(Ansys.UI.Interfaces.IGuiOperation):

  def Invoke(self, context):

    selection =
context.View.GetSingleUISelection[Ansys.Core.DataModel.ProjectSystem.DataCon
tainerReference]()

    if selection == None:

      uiMgr = Ansys.UI.UIManager.Instance

      workspace = uiMgr.GetActiveWorkspace()

      view = workspace.GetView("ProjectSchematic")

      selection =
view.GetSingleUISelection[Ansys.Core.DataModel.ProjectSystem.DataContainerRe
ference]()

    dialog = MyCustomPropertyDialog(selection)

    dialog.ShowDialog()

  def GuiItemCallBack(self, context):

    context.Visible = True

    context.Enabled = True

class MyCustomPropertyDialog(Ansys.UI.Toolkit.Dialog):

  mainPanel = None

  userString = None

  container = None

  def __init__(self, container):
```

```
    self.container = container
    width = 250;
    height = 100;
    self.Text = "Set Text"
    self.Width = width
    self.Height = height
    self.MaximizeBox = False
    self.MinimumSize = Ansys.UI.Toolkit.Drawing.Size(width, height)
    self.MaximumSize = Ansys.UI.Toolkit.Drawing.Size(width, height)
    self.__BuildUI()
    self.SetControl(self.mainPanel)
  def __BuildUI(self):
    self.mainPanel = Ansys.UI.Toolkit.TableLayoutPanel()
    self.mainPanel.Rows.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
50)
    self.mainPanel.Rows.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
50)
    self.mainPanel.Columns.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
50)
    self.mainPanel.Columns.Add(Ansys.UI.Toolkit.TableLayoutSizeType.Percent,
50)
    l = Ansys.UI.Toolkit.Label("Text:")
    self.mainPanel.Controls.Add(l, 0, 0)
    self.userString = Ansys.UI.Toolkit.TextBox()
    self.mainPanel.Controls.Add(self.userString, 0, 1)
    apply = Ansys.UI.Toolkit.Button("Apply")
    apply.Click += Ansys.UI.Toolkit.EventDelegate(self.apply_Click)
    self.mainPanel.Controls.Add(apply, 1, 0)
    cancel = Ansys.UI.Toolkit.Button("Cancel")
    cancel.Click += Ansys.UI.Toolkit.EventDelegate(self.cancel_Click)
    self.mainPanel.Controls.Add(cancel, 1, 1)
  def cancel_Click(self, sender, args):
    self.Close()
```

```
  def apply_Click(self, sender, args):
    context = Ansys.Core.Commands.CurrentContext.AsQueryContext
    dataRef = None
    dataType = ''
    if self.container != None:
      dataType = "ComponentSolveSettingsForAddin"
    else:
      dataType = "DesignPointSolveSettings"
      self.container = context.Project.GetStaticContainer("Schematic")
    with context.ContainerReadLock(self.container):
        dataRef = context.Project.GetDataReferencesByType(self.container,
dataType)[0]
    if dataRef != None:

Ansys.Core.Commands.Standard.SetEntityPropertyCommand.InvokeAndWait(dataRef,
"RsmUserString", self.userString.Text)
    self.Close()
```

Project Schematic



Data and Action Processing

Sometimes you would add custom processing for application-wide Workbench events or **Project Schematic** actions. If you used the SDK, you would have to subscribe event handlers within your add-in's core `Load` method:

```
public class Addin : Ansys.Core.Addins.PersistableAddinBase, IDefineGui,
IEventObserver, IEventFilter

{

  public override void Load(Ansys.Core.Addins.AddinLoadContext context)
```

```
  {


    Ansys.Core.Events.EventManager.FrameworkInstance.Subscribe(this, this);


context.CommandManager.RootContext.Project.EventSource.PersistenceStateChang
ed += new
EventHandler<Ansys.Core.Persistence.PersistenceActionsEventArgs>(EventSource
_PersistenceStateChanged);
  }


  void EventSource_PersistenceStateChanged(object sender,
Ansys.Core.Persistence.PersistenceActionsEventArgs e)
  {
    if (e.Action == Ansys.Core.Persistence.PersistenceActions.Replace)
    {
      //...handle project replaced event.
    }
  }


  public void Receive(IEvent eventInformation)
  {
    //handle matching event...
  }


  public bool IsMatch(IEvent eventInformation)
  {
    if (condition_for_match)
    {
      return true;
    }
    return false;
  }
}
```

In an ACT extension, you can either use the pre-defined workflow callbacks to handle **Project Schematic** actions or register/unregister event handlers from the extension's user interface initialize and terminate callbacks.

Event Handlers

XML File

```
<extension version="1" name="Events" icon="images\events.png">
  <guid shortid="Events">03966969-dcae-42ba-9bf0-96ed03d75d61</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
      <onterminate>terminate</onterminate>
    </callbacks>
  </interface>
</extension>
```

IronPython

```
import System
import clr
clr.AddReference('Ans.Core')
import Ansys.Core
clr.AddReference('Ans.UI')
import Ansys.UI

saveEventHandler = None
exitEventHandler = None
def init(context):
  global saveEventHandler
  global exitEventHandler
```

```
  ExtAPI.Log.WriteMessage("Starting Events App...")

  context = ExtAPI.DataModel.Context

  saveEventHandler =
System.EventHandler[Ansys.Core.Persistence.PersistenceActionsEventArgs](save
Handler)

  context.Project.EventSource.PersistenceStateChangeRequested +=
saveEventHandler

  if Ansys.UI.UIManager.IsRunningInteractively:

    exitEventHandler = System.EventHandler[System.EventArgs](exitHandler)

    Ansys.UI.UIManager.Instance.UIEventSource.ApplicationAboutToExit +=
exitEventHandler

def saveHandler(sender, args):

  if args.Action == Ansys.Core.Persistence.PersistenceActions.Save:

    ExtAPI.Log.WriteMessage("Project Save detected by Events App")

def exitHandler(sender, args):

  ExtAPI.Log.WriteMessage("Application exit detected by Events App")

def terminate(context):

  global saveEventHandler

  global exitEventHandler

  ExtAPI.Log.WriteMessage("Terminating Events App...")

  if saveEventHandler != None:

    context = ExtAPI.DataModel.Context

    context.Project.EventSource.PersistenceStateChanged -= saveEventHandler

  if exitEventHandler != None:

    if Ansys.UI.UIManager.IsRunningInteractively:

      Ansys.UI.UIManager.Instance.UIEventSource.ApplicationAboutToExit -=
exitEventHandler
```

Workflow Callbacks

XML File

```
<extension version="1" name="WorkflowCallbacksDemo">

  <guid shortid="WorkflowCallbacksDemo">96d0195b-e138-4841-a13a-
de12238c83f2</guid>
```

```
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
  <workflow name="WorkflowDemo1" context="Project" version="1">
    <callbacks>
      <onbeforetaskreset>onBeforeReset</onbeforetaskreset>
      <onaftertaskreset>onAfterReset</onaftertaskreset>
      <onbeforetaskrefresh>onBeforeRefresh</onbeforetaskrefresh>
      <onaftertaskrefresh>onAfterRefresh</onaftertaskrefresh>
      <onbeforetaskupdate>onBeforeUpdate</onbeforetaskupdate>
      <onaftertaskupdate>onAfterUpdate</onaftertaskupdate>
      <onbeforetaskduplicate>onBeforeDuplicate</onbeforetaskduplicate>
      <onaftertaskduplicate>onAfterDuplicate</onaftertaskduplicate>

<onbeforetasksourceschanged>onBeforeSourcesChanged</onbeforetasksourceschang
ed>

<onaftertasksourceschanged>onAfterSourcesChanged</onaftertasksourceschanged>
      <onbeforetaskcreation>onBeforeCreate</onbeforetaskcreation>
      <onaftertaskcreation>onAfterCreate</onaftertaskcreation>
      <onbeforetaskdeletion>onBeforeDelete</onbeforetaskdeletion>
      <onaftertaskdeletion>onAfterDelete</onaftertaskdeletion>

<onbeforetaskcanusetransfer>onBeforeCanUseTransfer</onbeforetaskcanusetransf
er>

<onaftertaskcanusetransfer>onAfterCanUseTransfer</onaftertaskcanusetransfer>

<onbeforetaskcanduplicate>onBeforeCanDuplicate</onbeforetaskcanduplicate>
      <onaftertaskcanduplicate>onAfterCanDuplicate</onaftertaskcanduplicate>
      <onbeforetaskstatus order="1">onBeforeStatus1</onbeforetaskstatus>
      <onaftertaskstatus order="2">onAfterStatus1</onaftertaskstatus>

<onbeforetaskpropertyretrieval>onBeforePropertyRetrieval</onbeforetaskproper
```

```
tyretrieval>

<onaftertaskpropertyretrieval>onAfterPropertyRetrieval</onaftertaskpropertyr
etrieval>
    </callbacks>
  </workflow>
</extension>
```

IronPython

```
def onBeforeReset(task):
  msg = getPrintMessage('pre-reset', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterReset(task):
  msg = getPrintMessage('post-reset', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeRefresh(task):
  msg = getPrintMessage('pre-refresh', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterRefresh(task):
  msg = getPrintMessage('post-refresh', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeUpdate(task):
  msg = getPrintMessage('pre-update', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterUpdate(task):
  msg = getPrintMessage('post-update', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeDuplicate(task):
  msg = getPrintMessage('pre-duplicate', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterDuplicate(task):
```

```
  msg = getPrintMessage('post-duplicate', task)
  ExtAPI.Log.WriteMessage(msg)
  return False
def onBeforeSourcesChanged(task):
  msg = getPrintMessage('pre-sources-changed', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterSourcesChanged(task):
  msg = getPrintMessage('post-sources-changed', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeCreate(templateTask):
  msg = getPrintMessage('pre-creation', templateTask)
  ExtAPI.Log.WriteMessage(msg)
def onAfterCreate(task):
  msg = getPrintMessage('post-creation', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeDelete(task):
  msg = getPrintMessage('pre-deletion', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterDelete(task):
  msg = getPrintMessage('post-deletion', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforeCanUseTransfer(sourceTask, targetTask):
  msg = 'in pre-can-use-transfer with source task ' + sourceTask.Name + '
and target task ' + targetTask.Name
  ExtAPI.Log.WriteMessage(msg)
def onAfterCanUseTransfer(sourceTask, targetTask, args):
  msg = 'in post-can-use-transfer with source task ' + sourceTask.Name + '
and target task ' + targetTask.Name
  ExtAPI.Log.WriteMessage(msg)
  args.Clear()
  return args
def onBeforeCanDuplicate():
```

```
  msg = getPrintMessage('pre-can-duplicate', None)
  ExtAPI.Log.WriteMessage(msg)
def onAfterCanDuplicate(args):
  msg = getPrintMessage('post-can-duplicate', None)
  ExtAPI.Log.WriteMessage(msg)
  return False;
def onBeforeStatus1(task):
  msg = getPrintMessage('pre-status 1', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterStatus1(task, args):
  msg = getPrintMessage('post-status 1', task)
  ExtAPI.Log.WriteMessage(msg)
def onBeforePropertyRetrieval(task):
  msg = getPrintMessage('pre-property', task)
  ExtAPI.Log.WriteMessage(msg)
def onAfterPropertyRetrieval(task, args):
  msg = getPrintMessage('post-property', task)
  ExtAPI.Log.WriteMessage(msg)
  return None
def getPrintMessage(msg, task):
  taskName = 'none'
  if task != None:
    taskName = "Unknown"
    if task.Name != "" and task.Name != None:
      taskName = task.Name
  return 'in ' + msg + ' callback for task ' + taskName
```

Workflow and Application Integration

The keystone role of the SDK was Workflow and Application Integration. Utilizing the same coding paths as ANSYS' own internal development, you would develop your add-in to "data integrate" your application inside Workbench. "Data integration" simply means the exposure of your application as a drag-and-drop connectable block within the **Project Schematic**, while retaining the original user-

interface of your application. Your application used the custom add-in as an intermediary with Workbench to synchronize data and process action requests.

Systems and Components

With the SDK, you would define one or more components and one or more systems to represent your workflow or application:

```
public class Addin : Ansys.Core.Addins.PersistableAddinBase, IDefineGui
{
  public override void Load(Ansys.Core.Addins.AddinLoadContext context)
  {
    CreateTemplates(context);
  }

  internal void CreateTemplates(Ansys.Core.Addins.AddinLoadContext context)
  {
    List<RequiredInput> Inputs_Empty = new List<RequiredInput>();
context.TemplateRepository.AddComponentTemplate("ScratchTemplate")
      .SetRequiredInputTypes(new IEnumerable<RequiredInput>[] {
        Inputs_Empty })
      .SetCreatingCommand(typeof(Commands.CreateScratchComponentCommand))
      .SetRefreshCommand(typeof(Commands.RefreshScratchComponentCommand))
      .SetUpdateCommand(typeof(Commands.UpdateScratchComponentCommand))

.SetDuplicateContainerCommand(typeof(Commands.DuplicateScratchComponentComma
nd))

.SetComponentPropertyDataQuery(typeof(Queries.GetScratchComponentPropertiesQ
uery))

.SetComponentStatusQuery(typeof(Queries.GetScratchComponentStatusQuery))
      .SetCreateAllProvidesConnectionsToNewSystem(true)
      .SetIsShareable(false)
      .SetOutputTypes(new string[] {
Core.AddinConstants.ScratchDataTypeString })
      .SetDisplayName("");
```

```
    context.TemplateRepository.AddSystemTemplate("ScratchTemplate")
      .SetToolboxGroup("Scratch")
      .SetComponentTemplateNames(new string[] { "ScratchTemplate" })
      .SetComponentNames(new string[] { "Scratch" })
      .SetSystemTypeAbbreviation("ScratchEX")
      .SetDefaultSystemName("Scratch Example")
      .SetToolboxDisplayText("Scratch Example")
      .SetImage("ScratchImage");
  }
}
```

ACT uses the terms *task* and *tas kgroup* instead of *component* and *system*. In your ACT extension, you define tasks and task groups in the XML definition file:

```
<extension version="1" name="TaskGroup"
icon="images\taskgroupextension.png">
  <guid shortid="TaskGroup">4b782a26-67de-4410-8e7e-902cad138c37</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
    <callbacks>
      <oninit>initialize</oninit>
      <onterminate>terminate</onterminate>
    </callbacks>
  </interface>
  <workflow name="Workflow" context="Project" version="1">
    <tasks>
      <task name="Task" caption="Task" icon="task" version="1">
        <callbacks>
          <onupdate>update</onupdate>
        </callbacks>
```

```
        <inputs>
          <input/>
        </inputs>
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="TaskGroup" caption="Task Group" icon="taskgroup"
category="ACT Workflows" headertext="Task Group" abbreviation="TG"
version="1">
        <includeTask name="Task" caption="Task"/>
      </taskgroup>
    </taskgroups>
  </workflow>
</extension>
```

Instead of defining `ICommand` and `IQuery` classes with the SDK, you define IronPython callbacks on your task:

SDK

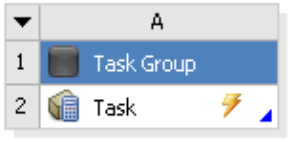
```
[Command("UpdateScratchComponent")]
public partial class UpdateScratchComponentCommand : ICommand
{
  [Parameter]
  public DataContainerReference Container;

  public void Execute(IFullContext context)
  {
    DataReference dr = null;
    double input = 0;
    using (context.ContainerReadLock(Container))
    {
      dr = context.Project.GetDataReferencesByType(Container,
```

```
Core.AddinConstants.ScratchDataTypeString)[0];
      input = (double)context.Project.GetProperty(dr, "inValue");


    }
    using (context.ContainerWriteLock(Container))
    {
      context.Project.SetProperty(dr, "outValue", input * input);
      DataObjectContainer containerObj =
context.Project.GetContainerObject(Container, Addin.key) as
DataObjectContainer;
      containerObj.SignalOutputsGenerated();
    }
  }
}
```

ACT

```
def update(task):
  inputValue = task.Properties["Inputs"].Properties["Input"].Value
  task.Properties["Outputs"].Properties["Output"].Value = inputValue *
inputValue
```

The **Project Schematic** result is the same:



Task Group

Additionally, you can re-use pre-installed tasks to compose new analysis task groups, as long as you maintain data transfer input and output rules:

```
<extension version="1" name="CustomStructural"
icon="images\custom_structural_extension.png">
  <guid shortid="CustomStructural">69d0095b-e138-4841-a13a-
de12238c85f9</guid>
```

```
  <script src="main.py" />

  <interface context="Project">

    <images>images</images>

  </interface>

   <workflow name="wf1" context="Project" version="1">

    <taskgroups>

      <taskgroup name="CustomStructural" caption="Custom Structural Group"
icon="custom_structural" category="ACT Workflows" abbreviation="CSTRUCT"
version="1">

        <includeTask external="True" name="EngDataCellTemplate" caption="My
Engineering Data"/>

        <includeTask external="True" name="GeometryCellTemplate" caption="My
Geometry"/>

        <includeTask external="True" name="SimulationModelCellTemplate"
caption="My Model"/>

        <includeTask external="True"
name="SimulationSetupCellTemplate_StructuralStaticANSYS" caption="My
Setup"/>

        <includeTask external="True"
name="SimulationSolutionCellTemplate_StructuralStaticANSYS" caption="My
Solution"/>

        <includeTask external="True"
name="SimulationResultsCellTemplate_StructuralStaticANSYS" caption="My
Results"/>

        <attributes SolverType="ANSYS" SystemName="Static Structural
(ANSYS)"/>

      </taskgroup>

    </taskgroups>

   </workflow>

</extension>
```

The **Project Schematic** result:

Custom Structural Group

Workflow Connections

The **Project Schematic** exposes a powerful workflow environment through the chaining together of simulation blocks. Connections between upstream and downstream tasks allow data transfer to flow from one application to another. With the SDK, you were required to implement data transfer by specifying input and output combinations on your component template, implementing a complex `DataSourcesListChanged` query, defining output data, and implementing the three data transfer methods: Acquire, Retrieve, and Release.

ACT simplifies the connection process. If you want to connect with other ANSYS tasks, you reference a list of input and output types from documentation.

Once you identify the connection type, you can list it as input to your task:

Extension File

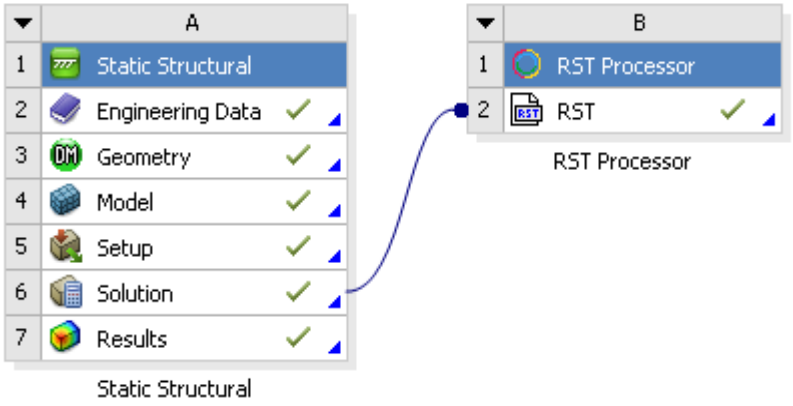
```
<extension version="1" name="RSTProcessor" icon="images\rst_extension.png">>
  <guid shortid="RSTProcessor">e7eed2a6-8498-4e11-bc3b-5cc3ba9fcc1d</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
   <workflow name="wf1" context="Project" version="1">
    <tasks>
      <task name="RSTProcessor" caption="RST" icon="rst_task" version="1">
        <callbacks>
          <onupdate>update</onupdate>
```

```
        </callbacks>
        <inputs>
          <input/>
          <input type="MechanicalSolution" format=""/>
        </inputs>
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="RSTProcessor" caption="RST Processor"
icon="rst_taskgroup" category="ACT Custom Workflows" abbreviation="RSTP"
version="1">
        <includeTask name="RSTProcessor" caption="RST"/>
      </taskgroup>
    </taskgroups>
   </workflow>
</extension>
```

IronPython

```
def update(task):
  ExtAPI.Log.WriteMessage("Updating " + task.Name)
  if task.InputData.ContainsKey("MechanicalSolution"):
    rstFile = task.InputData["MechanicalSolution"][0]
    currRSTFile = GetDesignPointFile(rstFile)
    rstFilePath = currRSTFile.Location
    ExtAPI.Log.WriteMessage("Discovered RST File - " + rstFilePath)
```

Project Schematic



Similarly, if your task will output data to be consumed downstream, you can find the appropriate type and enter it as output from your task. You can also define custom types to connect and transfer to tasks only defined by your extension and mix the types within a workflow:

Extension File

```
<extension version="1" name="CustomCFD" icon="images\CustomCFD.png">
  <guid shortid="CustomCFD">69d0095b-e138-3475-a13a-de12238c83f2</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
  <workflow name="CustomCFDWorkflow" context="Project" version="1">
    <tasks>
      <task name="Geometry" caption="Geometry Creation"
icon="geometry_component" version="1">
        <callbacks>
          <onupdate>updateGeometry</onupdate>
        </callbacks>
        <inputs>
          <input/>
        </inputs>
```

```
        <outputs>
          <output type="CustomGeometry" format=""/>
        </outputs>
        <propertygroup name="Inputs">
          <property name="dim1" caption="Length" control="double"
default="10.0" readonly="False" needupdate="true" visible="True"
persistent="True" parameterizable="True">
            <callbacks>
              <isvalid>isValid1</isvalid>
            </callbacks>
          </property>
          <property name="dim2" caption="Width" control="double"
default="3.0" readonly="False" needupdate="true" visible="True"
persistent="True" parameterizable="True">
            <callbacks>
              <isvalid>isValid2</isvalid>
            </callbacks>
          </property>
        </propertygroup>
      </task>
      <task name="Meshing" caption="Meshing Creation"
icon="meshing_component" version="1">
        <callbacks>
          <onupdate>updateMesh</onupdate>
        </callbacks>
        <inputs>
          <input type="CustomGeometry" format=""/>
          <input/>
        </inputs>
        <outputs>
          <output type="CFXMesh" format=""/>
        </outputs>
      </task>
```

```
    </tasks>
    <taskgroups>
      <taskgroup name="CFDGeomtryGroup" caption="CFD Geometry"
icon="geometry_system" category="ACT Workflows" abbreviation="CFDCAD"
version="1">
        <includeTask name="Geometry" caption=""/>
      </taskgroup>
      <taskgroup name="CFDMeshGroup" caption="CFD Mesh"
icon="meshing_system" category="ACT Workflows" abbreviation="CFDMSH"
version="1">
        <includeTask name="Meshing" caption=""/>
      </taskgroup>
    </taskgroups>
  </workflow>
</extension>
```

IronPython

```
import clr
import os
clr.AddReference("Ans.Utilities")
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.toolkit.Base")
import System
import sys
import subprocess
import shutil
global myres
import math
import time


from Ansys.Utilities import *
from System.IO import *
```

```
from System.Diagnostics import *
from Ansys.UI.Toolkit import *


def updateGeometry(task):
  icemgeoAddinPath = task.Extension.InstallDir
  installPath =
ApplicationConfiguration.DefaultConfiguration.WorkbenchInstallRootDirectoryP
ath
  activeDir = task.ActiveDirectory
  #geom replay contents
  geomReplayContents = """
  #you can find full replay file contents in template files.
  """
  #Create the replay file in the project dir
  replayFileName = "create_geom.rpl"
  replayFilePath = Path.Combine(activeDir, replayFileName)
  #Assembling the replay file with correct storage location
  dim1 = task.Properties["Inputs"].Properties["dim1"].Value
  dim2 = task.Properties["Inputs"].Properties["dim2"].Value
  CreateRPLFile = open(replayFilePath,'w')
  CreateRPLFile.write("set inp1 " + str(dim1) + '\n')
  CreateRPLFile.write("set inp2 " + str(dim2) + '\n')
  CreateRPLFile.write(geomReplayContents + '\n')
  #Define the path for the tin file to save and write to the replay file
  geoFileName = "channel.tin"
  destinationPath = Path.Combine(activeDir, geoFileName)
  #Write the save command to the rpl file
  strReplace=destinationPath.replace('\\','/')
  CreateRPLFile.write("ic_save_tetin " + strReplace + " 0 0 {} {} 0 0
1"+'\n')
  CreateRPLFile.close()
  #Launch icem
  icemBin = Path.Combine(installPath, Path.Combine("icemcfd",
```

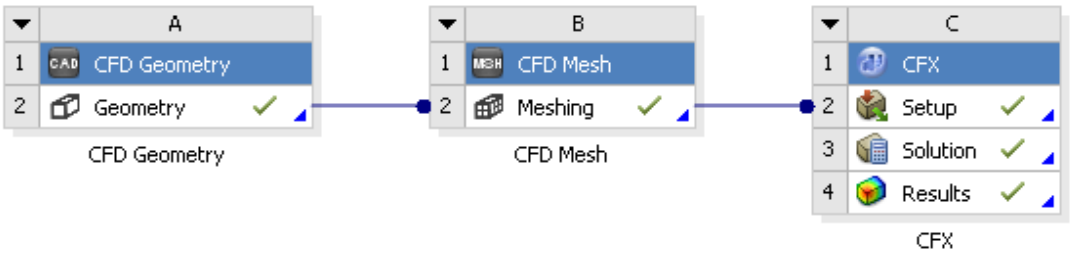
```
Path.Combine("win64_amd","bin")))
  icemExeName = "icemcfd.bat"
  icemExePath = Path.Combine(icemBin, icemExeName)
  if File.Exists(icemExePath) == True:
    info = ProcessStartInfo(icemExePath)
    info.Arguments = System.String.Format("-batch -script \"{0}\"",
replayFilePath)
    Process.Start(info).WaitForExit()
  #Register the .tin file
  geoFile = None
  isRegistered = IsFileRegistered(FilePath=destinationPath)
  if isRegistered == True:
     geoFile = GetRegisteredFile(destinationPath)
  else:
     geoFile = task.RegisterFile(destinationPath)
  #Transfer file downstreams
  geoRefs = task.OutputData
  geoSet = geoRefs["CustomGeometry"]
  geoData = geoSet[0]
  geoData.TransferFile = geoFile
def updateMesh(task):
  #Calculate Paths
  installPath =
ApplicationConfiguration.DefaultConfiguration.WorkbenchInstallRootDirectoryP
ath
  icemmeshAddinPath = task.Extension.InstallDir
  activeDir = task.ActiveDirectory
  cfxFormattedActiveDir = activeDir.replace('\\','/')
  #Obtain upstream data
  upstreamData = task.InputData["CustomGeometry"]
  geoFileRef = None
  upstreamDataCount = upstreamData.Count
```

```
  if upstreamDataCount > 0:
    geoFileRef = upstreamData[0]
  replayContents = """
  #you can find full replay contents in template files
  """
  #Create the replay file
  replayFileName = "create_mesh.rpl"
  mshFileName = "channel.msh"
  replaydestinationPath = Path.Combine(activeDir, replayFileName)
  mshFilePath = Path.Combine(activeDir, mshFileName)
  CreateRPLFile = open(replaydestinationPath,'w')
  strgeoFile = geoFileRef.Location
  strgeoFileReplace=strgeoFile.replace('\\','/')
  CreateRPLFile.write("ic_load_tetin " + strgeoFileReplace + '\n')
  CreateRPLFile.write(replayContents + '\n')
  #Add the path to the save location for mesh file
  mshFileReplace=mshFilePath.replace('\\','/')
  cfx5Path = Path.Combine(installPath, Path.Combine("icemcfd",
Path.Combine("win64_amd", Path.Combine("icemcfd", Path.Combine("output-
interfaces", "cfx5")))))
  saveContent = "ic_exec {" + cfx5Path + "} -dom project1.uns -b
project1.fbc -ascii -db -internal_faces " + mshFileReplace + '\n'
  CreateRPLFile.write(saveContent)
  CreateRPLFile.close()
  #Launch icem
  icemBin = Path.Combine(installPath, Path.Combine("icemcfd",
Path.Combine("win64_amd","bin")))
  icemExeName = "icemcfd.bat"
  icemExePath = Path.Combine(icemBin, icemExeName)
  if File.Exists(icemExePath) == True:
    info = ProcessStartInfo(icemExePath)
    info.Arguments = System.String.Format("-batch -script \"{0}\"",
replaydestinationPath)
```

```
    Process.Start(info).WaitForExit()
  #Register the .msh file
  mshFile = None
  isRegistered = IsFileRegistered(FilePath=mshFilePath)
  if isRegistered == True:
     mshFile = GetRegisteredFile(mshFilePath)
  else:
     mshFile = task.RegisterFile(mshFilePath)
  #Convert File into a gtm file readable by the cfx system
  targetGTMName = "channel_converted.gtm"
  targetGTMPath = Path.Combine(activeDir, targetGTMName)
  cfxBin = Path.Combine(installPath, Path.Combine("CFX", "bin"))
  cfxConverterExeName = "cfx5gtmconv.exe"
  cfxConverterExePath = Path.Combine(cfxBin, cfxConverterExeName)
  if File.Exists(cfxConverterExePath) == True:
    info = ProcessStartInfo(cfxConverterExePath)
    info.Arguments = System.String.Format("-icem \"{0}\" \"{1}\"",
mshFilePath, targetGTMPath)
    Process.Start(info).WaitForExit()
  gtmFile = None
  if File.Exists(targetGTMPath) == True:
    if IsFileRegistered(FilePath=targetGTMPath) == False:
      gtmFile = task.RegisterFile(targetGTMPath)
    else:
      gtmFile = GetRegisteredFile(FilePath=targetGTMPath)
  outputRefs = task.OutputData
  meshOutputSet = outputRefs["CFXMesh"]
  meshOutput = meshOutputSet[0]
  meshOutput.FileName = gtmFile
  meshOutput.PreFileType = "GTM"
def isValid1(entity, property):
```

```
  if property.Value < 0.0 or property.Value > 20.0:
    return False
  else:
    return True
def isValid2(entity, property):
  if property.Value < 0.0 or property.Value > 3.0:
    return False
  else:
    return True
```

Project Schematic



A more simplistic example when you define custom transfer types for your own tasks and workflows:

Extension File

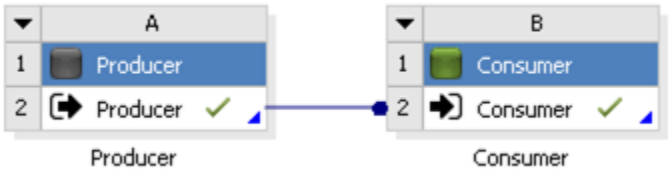
```
<extension version="1" name="CustomTransfer"
icon="images\custom_transfer_256p.png">
  <guid shortid="CustomTransfer">69d0095b-e138-4841-a13a-de12238c83f3</guid>
  <script src="customtransfer.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
   <workflow name="wf4" context="Project" version="1">
    <tasks>
      <task name="Producer" caption="Producer" icon="producer_task"
version="1">
        <callbacks>
```

```
          <onupdate>producer_update</onupdate>
        </callbacks>
        <inputs>
          <input/>
        </inputs>
        <outputs>
          <output format="" type="MyData"/>
        </outputs>
      </task>
      <task name="Consumer" caption="consumer" icon="consumer_task"
version="1">
        <callbacks>
          <onupdate>consumer_update</onupdate>
        </callbacks>
        <inputs>
          <input/>
          <input format="" type="MyData"/>
        </inputs>
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="Producer" caption="Producer" icon="producer_system"
category="ACT Workflows" abbreviation="Producer" version="1">
        <includeTask name="Producer" caption="Producer"/>
      </taskgroup>
      <taskgroup name="Consumer" caption="Consumer" icon="consumer_system"
category="ACT Workflows" abbreviation="Consumer" version="1">
        <includeTask name="Consumer" caption="Consumer"/>
      </taskgroup>
   </workflow>
</extension>
```

IronPython

```
import System

def consumer_update(task):
  ExtAPI.Log.WriteMessage("updating " + task.Name)
  upstreamData = task.InputData["MyData"]
  fileRef = None
  upstreamDataCount = upstreamData.Count
  if upstreamDataCount > 0:
    fileRef = upstreamData[0]
    AssociateFileWithContainer(fileRef, task.InternalObject)
    ExtAPI.Log.WriteMessage("Recieved file "+fileRef.Location+" from
upstream task.")
  #if no new data...nothing to process from upstream sources.
def producer_update(task):
  ExtAPI.Log.WriteMessage("updating " + task.Name)
  extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
  filePath = System.IO.Path.Combine(extensionDir, "Sample_Materials.xml")
  fileRef = task.RegisterFile(filePath)
  outputSet = task.OutputData["MyData"]
  myData = outputSet[0]
  myData.TransferFile = fileRef
```

Project Schematic



ACT additionally exposes a generic transfer mechanism. When operating under general transfer, you need not specify inputs or outputs on your tasks. Instead, ACT defines and manages the transfer types for you. You can use generic transfer to connect one or more custom tasks as well as any ANSYS-based **Model**, **Setup**, or **Solution** task inside an analysis system.

Re-writing the simple custom connection extension from earlier by using generic transfer makes lightweight data exchange easier. You can push data into a *data store* for access by the downstream tasks:

Extension File

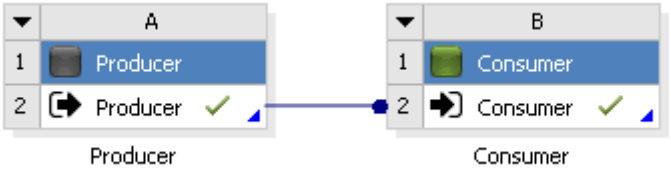
```
<extension version="1" name="GenericTransfer"
icon="images\general_transfer_256p.png">

  <guid shortid="GenericTransfer">86c99dad-9f06-4eb5-bb79-
8139e4ce9f3a</guid>

  <script src="main.py" />

  <interface context="Project">

    <images>images</images>

  </interface>

   <workflow name="generalTest" context="Project" version="1">

    <tasks>

      <task name="Producer" caption="Producer" icon="producer_task"
version="1" enableGenericTransfer="True">

        <callbacks>

          <onupdate>producer_update</onupdate>

        </callbacks>

        <inputs>

          <input/>

        </inputs>

        <outputs/>

      </task>

      <task name="Consumer" caption="consumer" icon="consumer_task"
version="1">

        <callbacks>

          <onupdate>consumer_update</onupdate>

        </callbacks>

        <inputs>

          <input/>

          <input format="" type="GeneralTransfer" count="2"/>

        </inputs>
```

```
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="Producer" caption="Producer" icon="producer_system"
category="ACT Workflows" abbreviation="Producer" version="1">
        <includeTask name="Producer" caption="Producer"/>
      </taskgroup>
      <taskgroup name="Consumer" caption="Consumer" icon="consumer_system"
category="ACT Workflows" abbreviation="Consumer" version="1">
        <includeTask name="Consumer" caption="Consumer"/>
      </taskgroup>
    </taskgroups>
   </workflow>
</extension>
```

IronPython

```
def consumer_update(task):
  if task.SourceTasks.Count == 1:
    data = task.SourceTasks[0].TransferData["Test"]
def producer_update(task):
  task.TransferData["Test"] = "This is a test from " + task.Name
```

Project Schematic



Defining an extension for connection with ANSYS-based simulation tasks is just as easy:

Extension File

```
<extension version="1" name="CustomLoad"
```

```
icon="images\customload_extension.png">
  <guid shortid="CustomLoad">69d0590b-e138-4841-a13a-de12238c83f2</guid>
  <script src="main.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
  <interface context="Mechanical">
    <images>images</images>
    <toolbar name="CustomLoad" caption="">
      <entry name="CustomLoad" icon="">
        <callbacks>
          <onclick>customLoadClick</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
  <workflow name="MyWorkflow" context="Project" version="1">
    <callbacks>
      <onaftertaskrefresh>pushGenericData</onaftertaskrefresh>
      <onbeforetaskupdate>pushGenericData</onbeforetaskupdate>
    </callbacks>
    <tasks>
      <task name="CustomLoad" caption="Custom Load" icon="customload_task"
version="1">
        <property name="Pressure" caption="Pressure" control="float"
default="1.0" readonly="False" needupdate="true" visible="True"
persistent="True" parameterizable="True" />
        <callbacks>
          <onupdate>update</onupdate>
        </callbacks>
        <inputs>
          <input/>
```

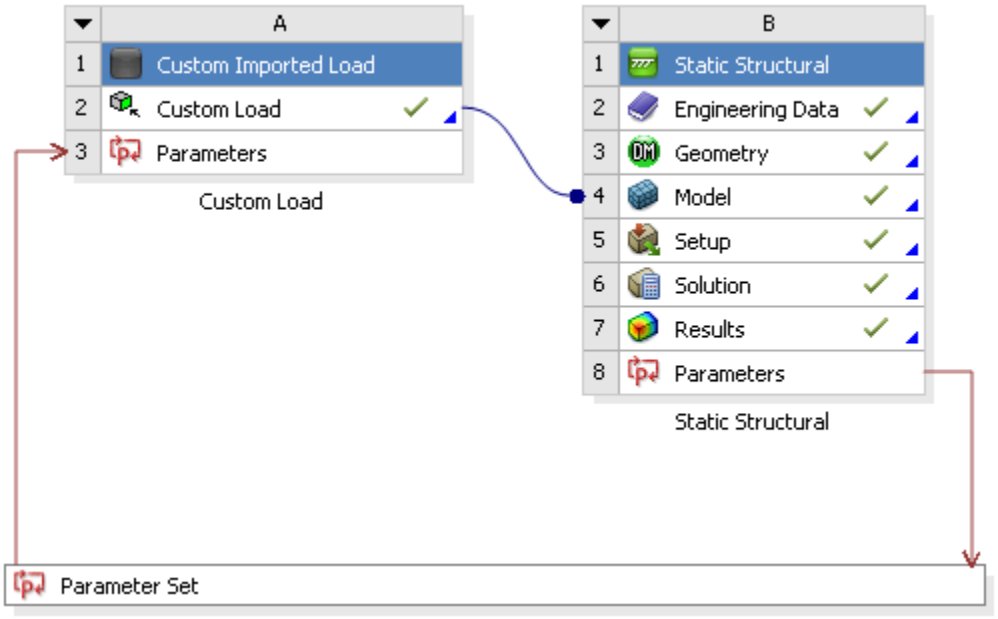
```
        </inputs>
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="CustomLoad" caption="Custom Load" icon="customload"
category="ACT Workflows" headertext="Custom Imported Load" abbreviation="C-
LOAD" version="1">
        <includeTask name="CustomLoad" caption="Custom Load"/>
      </taskgroup>
    </taskgroups>
  </workflow>
</extension>
```

IronPython

```
def updateLoad(guid, pressureValue):
  ExtAPI.Log.WriteMessage("Entered update load")
  dict = {}
  if ExtAPI.Extension.Attributes.Contains("Loads"):
    dict = ExtAPI.Extension.Attributes["Loads"]
  else:
    ExtAPI.Extension.Attributes.SetValue("Loads", dict)

  load = None
  if not dict.ContainsKey(guid):
    load = ExtAPI.DataModel.AnalysisList[0].AddPressure()
    dict.Add(guid, load.ObjectId)
    ExtAPI.Log.WriteMessage("created pressure load")
  else:
    load = ExtAPI.DataModel.GetObjectById(dict[guid])
  ExtAPI.Log.WriteMessage("set pressure value")
```

```
  load.Magnitude.Output.DiscreteValues=[Quantity("0
[Pa]"),Quantity(pressureValue+" [Pa]")]

def pushGenericData(task):

  cmd = ""

  sourceTask = None

  if task.Name.Contains("Model"):

    for source in task.SourceTasks:

      if source.Name.Contains("CustomLoad"):

        sourceTask = ACT.GetACTTaskForContainer(source.Container)

        break

  if sourceTask != None:

    outputData = sourceTask.OutputData["GeneralTransfer"][0]

    guid = outputData.Guids[0]

    pressureVal = sourceTask.Properties['Pressure'].Value

    task.ExecuteCommand("SendCommand", {
'Command':"ExtAPI.ExtensionManager.GetExtensionByName('"+sourceTask.Extensio
n.Name+"').GetModule().updateLoad(\""+str(guid)+"\",\""+str(pressureVal)+"\"
)", 'Language':"Python" })

def update(task):

  ExtAPI.Log.WriteMessage('Updating ' + task.Name)

def customLoadClick(args):

  ExtAPI.Log.WriteMessage('Custom Load Toolbar Button Clicked')
```

Project Schematic



Data Entities and Properties

To store and expose your application data within Workbench with the SDK, you would define one or more data entities. Within each data entity you would declare Spec properties, specifying display, storage, and behavior metadata:

```
[DataEntity(Core.AddinConstants.ScratchDataTypeString, Exposure =
ExposureLevel.AnyUser)]

public class ScratchData : DataObject

{

  [Spec(Label = "Integer Input", Group = "Inputs", Default = 1, Usage =
SpecUsage.Input, Exposure = ExposureLevel.AnyUser, Access =
SpecAccess.UserWritable)]

  public double inValue;


  [Spec(Label = "Integer Sqaure Output", Group = "Output", Usage =
SpecUsage.Output, Exposure = ExposureLevel.AnyUser, Access =
SpecAccess.ReadOnly)]

  public double outValue;


  public ScratchData(string name, DataObjectContainer Container)
```

```
    : base(name, Container)
  {
  }

  public ScratchData()
    : this(null, null)
  {
  }
}
```

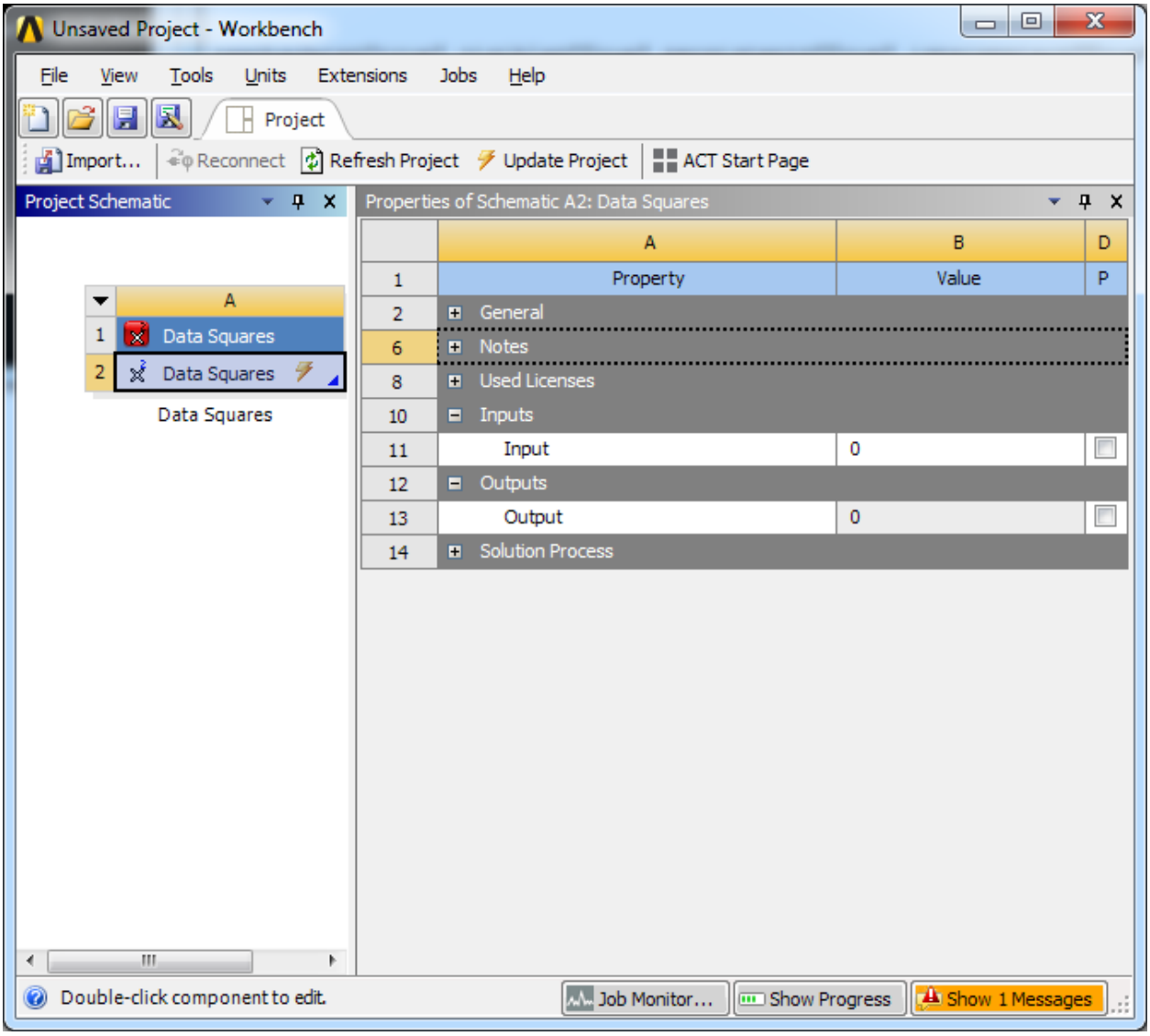
ACT automatically creates a dynamic data entity to store all of your properties. You can declare properties per task within your extension file. You can also use property groups to organize multiple properties and their display within the Workbench **Properties** view:

Extension File

```
<extension version="1" name="DataSquares"
icon="images\dsquares_extension.png">
  <guid shortid="DataSquares">69d0095b-e138-4841-a13a-de12238c83f2</guid>
  <script src="datasquares_complete.py" />
  <interface context="Project">
    <images>images</images>
  </interface>
  <workflow name="MyWorkflow" context="Project" version="1">
    <tasks>
      <task name="DataSquares" caption="Data Squares"
icon="dsquares_component" version="1">
        <callbacks>
          <onupdate>update</onupdate>
          <onstatus>status</onstatus>
          <onreset>reset</onreset>
        </callbacks>
        <propertygroup name="Inputs">
          <property name="Input" caption="Input" control="float"
default="0.0" readonly="False" needupdate="true" visible="True"
```

```
persistent="True" isparameter="True" />
        </propertygroup>
        <propertygroup name="Outputs">
          <property name="Output" caption="Output" control="float"
default="0.0" readonly="True" visible="True" persistent="True"
isparameter="True" />
        </propertygroup>
        <inputs>
          <input/>
        </inputs>
        <outputs/>
      </task>
    </tasks>
    <taskgroups>
      <taskgroup name="DataSquares" caption="Data Squares" icon="dsquares"
category="ACT Workflows" headertext="Data Squares" abbreviation="DSQRS"
version="1">
        <includeTask name="DataSquares" caption="Data Squares"/>
      </taskgroup>
    </taskgroups>
  </workflow>
</extension>
```

Project Schematic



ACT also simplifies property access. With the SDK, you had to obtain a container lock to read or write property values:

```
[Command("UpdateScratchComponent")]
public partial class UpdateScratchComponentCommand : ICommand
{
  [Parameter]
  public DataContainerReference Container;
```

```
  public void Execute(IFullContext context)
  {
    DataReference dr = null;
    double input = 0;
    using (context.ContainerReadLock(Container))
    {
      dr = context.Project.GetDataReferencesByType(Container,
Core.AddinConstants.ScratchDataTypeString)[0];
      input = (double)context.Project.GetProperty(dr, "inValue");


    }
    using (context.ContainerWriteLock(Container))
    {
      context.Project.SetProperty(dr, "outValue", input * input);
      DataObjectContainer containerObj =
context.Project.GetContainerObject(Container, Addin.key) as
DataObjectContainer;
      containerObj.SignalOutputsGenerated();
    }
  }
}
```

In your extension callbacks, you access properties directly off of your task and can manipulate the values with no extra actions. Note how a property group inserts an extra level for property access:

```
def update(task):
  inputValue = task.Properties["Inputs"].Properties["Input"].Value
  task.Properties["Outputs"].Properties["Output"].Value = inputValue *
inputValue
```

Parameters

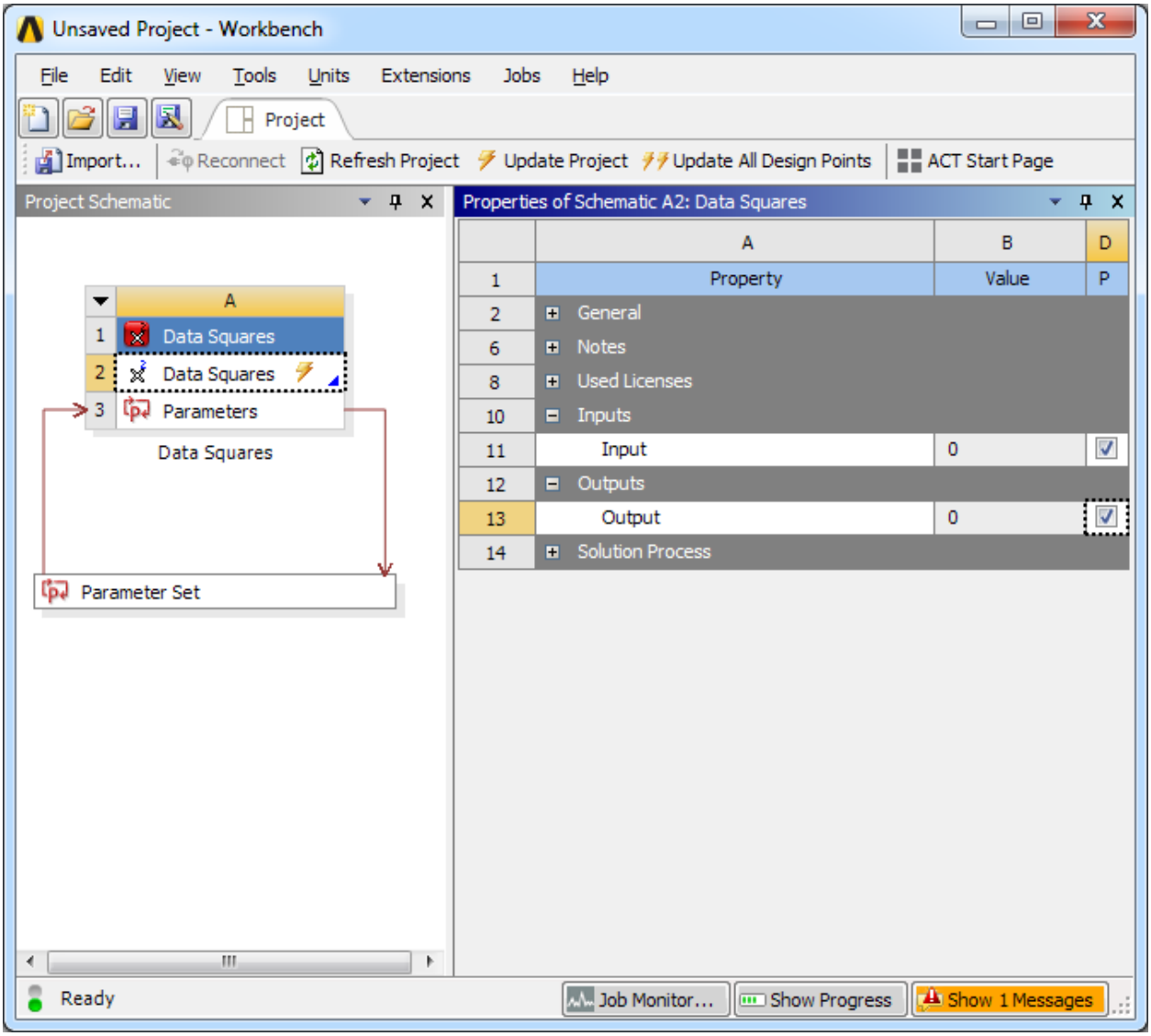
Data entity properties automatically support parameterization when you designate them as input/output and parameterizable. With the SDK, you supplied this data within the property's Spec attribute:

```
[Spec(... Usage = SpecUsage.Input ...)]
public double inValue;

[Spec(... Usage = SpecUsage.Output ...)]
public double outValue;
```

In your extension, you specify XML attributes on your property tags:

```
<property name="Input" ... needupdate="true" isparameter="True" />
<property name="Output" ... readonly="True" isparameter="True" />
```

The **Project Schematic** result is the same. Check boxes appear in the last column of the **Properties** view. Selecting these boxes will create new parameters and display the **Parameter Set** bar in the **Project Schematic**:



Sometimes your application may define parameters on-the-fly at runtime. With the SDK, you would define dynamic data objects, add spec information, and create the new entities within your component container. The process is similar in ACT:

```
import System
import clr
clr.AddReference("Ansys.ACT.WorkBench")
import Ansys.ACT.WorkBench
clr.AddReference("Ans.Utilities")
import Ansys.Utilities
```

```
clr.AddReference("Ans.UI.Toolkit")

clr.AddReference("Ans.UI.Toolkit.Base")

from Ansys.UI.Toolkit import *

from Ansys.UI.Toolkit.Base import *

from Ansys.UI.Toolkit.Drawing import *

clr.AddReference("Ans.Core")

import Ansys.Core


def createParameter(task, name, type, value, isOutput):

  ExtAPI.Log.WriteMessage("Creating parameter for " + task.Name)

  valueString = "NONE"

  if value != None:

    valueString = str(value)

  context = ExtAPI.DataModel.Context

  container = task.InternalObject

  installPath =
Ansys.Utilities.ApplicationConfiguration.DefaultConfiguration.AwpRootEnviron
mentVariableValue

  platform =
Ansys.Utilities.ApplicationConfiguration.DefaultConfiguration.Platform

  assmPath = System.IO.Path.Combine(installPath,
System.IO.Path.Combine("Addins", System.IO.Path.Combine("ACT",
System.IO.Path.Combine("bin", System.IO.Path.Combine(platform,
"Ansys.ACT.WorkBench.dll")))))

  assm = System.Reflection.Assembly.LoadFrom(assmPath)

  utilitiesType = assm.GetType("Ansys.ACT.WorkBench.Schematic.Utilities")

  field = utilitiesType.GetField("ContainerKey",
System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static)

  key = field.GetValue(None)

  containerObject = context.Project.GetContainerObject(container, key)

  pType = None

  if type == "Integer":

    pType = clr.GetClrType(System.Int32)
```

```
  elif type == "Float":
    pType = clr.GetClrType(System.Single)
  else:
    pType = clr.GetClrType(Ansys.Core.Units.Quantity)
  spec = Ansys.Core.DataModel.DataObjects.ParameterAdapterTypeSpec(pType,
containerObject)
  spec.DefineAdapterProperty("ParameterName",
Ansys.Core.DataSpecs.StringSpec())
  spec.DefineAdapterProperty("IsOutput", Ansys.Core.DataSpecs.BooleanSpec())
  spec.DefineAdapterProperty("ParameterType",
Ansys.Core.DataSpecs.StringSpec())
  parameterSourceObject =
Ansys.Core.DataModel.DataObjects.DynamicDataObject(spec)
  parameterSourceObject.SetProperty("ParameterName", name)
  parameterSourceObject.SetProperty("IsOutput", isOutput)
  parameterSourceObject.SetProperty("ParameterType", type)
  containerObject.AddDataObject(parameterSourceObject)
  paramName = container.Name + "::" + name
  if value == None:
    value = string.Empty
  else:
    value = value.ToString()

Parameters.CreateParameter(Entity=parameterSourceObject.GetDataReference(),
PropertyName="Value", IsOutput=isOutput, IsDirectOutput=isOutput,
Expression=value, DisplayText=paramName)
```

Remote Solve Manager

Your SDK project may have involved Remove Solve Manager (RSM) if your external application update routine was time consuming or resource intensive. RSM allows you to send jobs to a remote compute server for queuing and execution on high performance compute clusters.

You would have defined one or more data entities for background job tracking, implemented the “pending state” routines on your update command, added new queries and commands for job status pinging, reconnect requests, and failure handling, and finally provide resync code to merge the remote job results with the local project. For SDK developers, RSM was the second most complex add-in development topic, behind data transfer.

Fortunately, ACT simplifies the process at the extension task level. You first define RSM-specific information in the extension definition file as a child of the task:

```
<task name="DataSquares" caption="Data Squares" icon="dsquares_component"
version="1">

  ...

  <rsmjob name="squares" deletefiles="True" version="1">

    <inputfile id="1" name="input1.txt"/>

    <outputfile id="1" name="output1.txt"/>

    <program>

      <platform name="Win64"
path="%AWP_ROOT190%\Addins\ACT\extensions\DataSquares\ExampleAddinExternalSo
lver.exe" />

      <platform name="Linux64"
path="%AWP_ROOT190%/Addins/ACT/extensions/DataSquares/ExampleAddinExternalSo
lver.exe" />

      <argument name="" value="inputFile:1" separator=""/>

      <argument name="" value="outputFile:1" separator=""/>

    </program>

    <callbacks>

      <oncreatejobinput>createJobInput</oncreatejobinput>

      <onjobstatus>getJobStatus</onjobstatus>

      <onjobcancellation>cancelJob</onjobcancellation>

      <onjobreconnect>reconnectJob</onjobreconnect>

    </callbacks>

  </rsmjob>

</task>
```

Next, you implement the four required callbacks:

```
def createJobInput(task, inputFilePaths):

  ExtAPI.Log.WriteMessage('creating job input')

  inputFilePath = inputFilePaths[0]

  #get param values

  inputValue = task.Properties["Inputs"].Properties["Input"].Value
```

```
  #write input file
  ExtAPI.Log.WriteMessage("Writing input value ("+str(inputValue)+") to file
(" + inputFilePath + ")")
  f = open(inputFilePath, "w")

f.write('input='+inputValue.ToString(System.Globalization.NumberFormatInfo.I
nvariantInfo))
  f.close()
def reconnectJob(task, outputFilePaths):
  ExtAPI.Log.WriteMessage('reconnecting job')
  outputValue = None
  outputFilePath = outputFilePaths[0] #I know we only have one specified
based on our definition...so work off of the first entry
  f = open(outputFilePath, "r")
  currLine = f.readline()
  while currLine != "":
    valuePair = currLine.split('=')
    outputValue = System.Double.Parse(valuePair[1],
System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
  f.close()
  #set output value
  ExtAPI.Log.WriteMessage("Retrieved value (" + str(outputValue) + ") from
file (" + outputFilePath + ")")
  if outputValue == None:
    raise Exception("Error in update - no output value detected!")
  else:
    task.Properties["Outputs"].Properties["Output"].Value = outputValue
def getJobStatus(task, outputFiles):
  ExtAPI.Log.WriteMessage('checking job status')
  outputFilePath = outputFiles[0]
  finished = System.IO.File.Exists(outputFilePath)
```

```
  return finished
def cancelJob(task, inputFiles, outputFiles):
  ExtAPI.Log.WriteMessage('performing cancellation clean up')
```

Now your task is RSM-compatible. The complete IronPython script with full update, progress monitoring, and task callback routines:

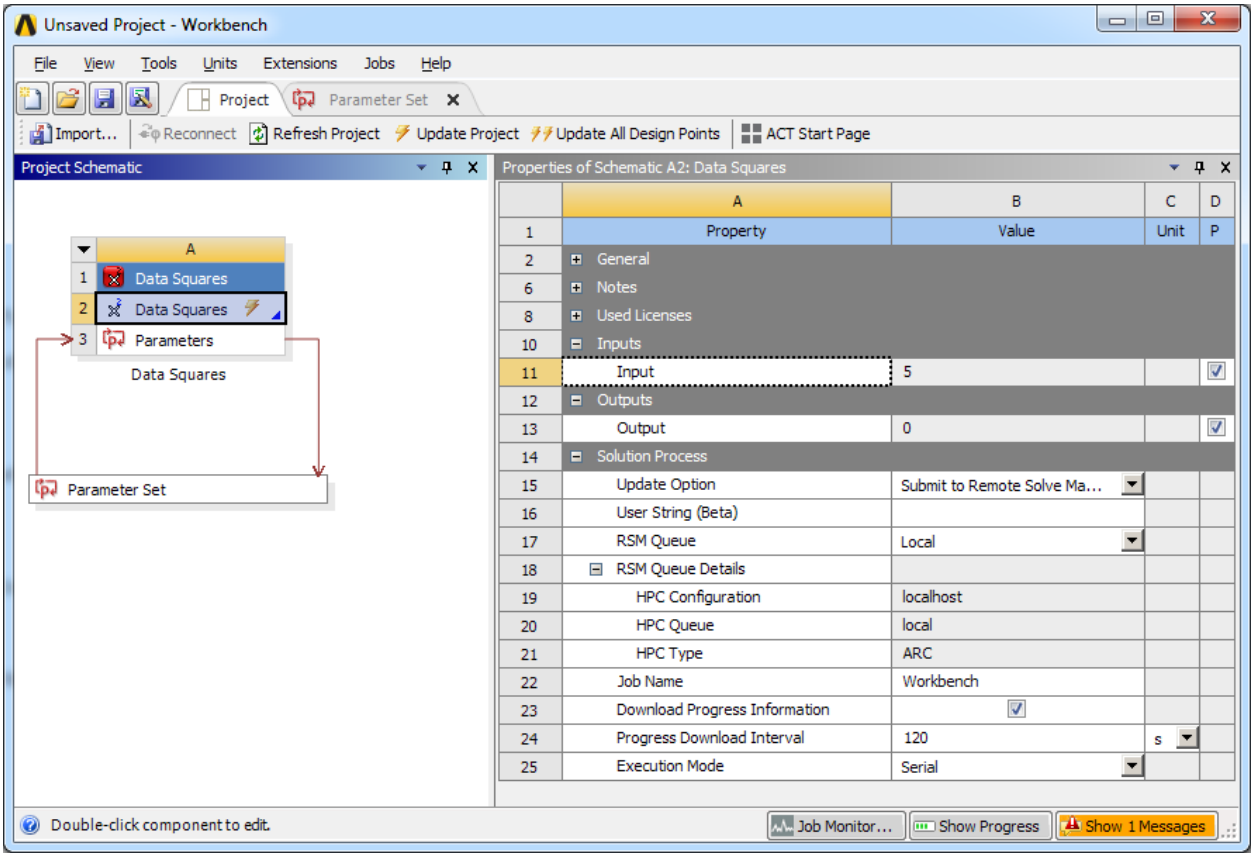
```
def update(task):
  activeDir = task.ActiveDirectory
  extensionDir = task.Extension.InstallDir
  exeName = "ExampleAddinExternalSolver.exe"
  solverPath = System.IO.Path.Combine(extensionDir, exeName)

  monitor = ExtAPI.UserInterface.ProgressMonitor
  monitor.BeginWork("Data Sqaures Solver", 3)
  monitor.WorkDetails = "Preparing solver input..."
  monitor.UpdateWork(1)

  #get param values
  inputValue = task.Properties["Inputs"].Properties["Input"].Value

  #prep i/o file paths

  inputFileName = "input.txt"
  outputFileName = "output.txt"
  dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)
  dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)

  #write input file
  f = open(dpInputFile, "w")

f.write('input='+inputValue.ToString(System.Globalization.NumberFormatInfo.I
nvariantInfo))
```

```
  f.close()


  monitor.UpdateWork(1)


  monitor.WorkDetails = "Executing Solver..."


  #run exe


  exitCode = ExtAPI.ProcessUtils.Start(solverPath, dpInputFile,
dpOutputFile)
  if exitCode != 0:
    raise Exception ('External solver failed!')
  #read output file


  monitor.UpdateWork(1)


  monitor.WorkDetails = "Retrieving results from solver..."


  #read output file


  outputValue = None
  f = open(dpOutputFile, "r")
  currLine = f.readline()
  while currLine != "":
    valuePair = currLine.split('=')
    outputValue = System.Double.Parse(valuePair[1],
System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
  f.close()


  monitor.UpdateWork(1)
```

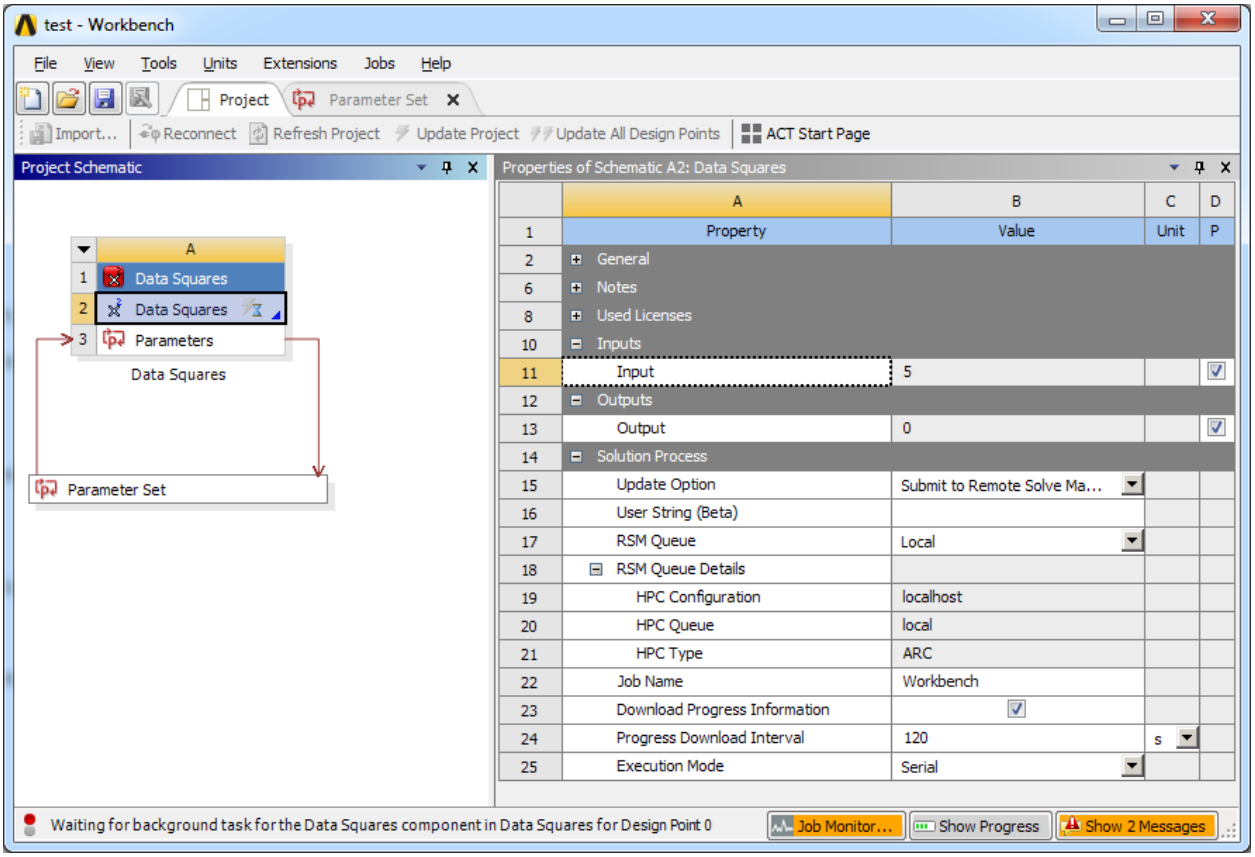
```
  #set output value


  if outputValue == None:
    raise Exception("Error in update - no output value detected!")
  else:
    task.Properties["Outputs"].Properties["Output"].Value = outputValue
  monitor.WorkDetails = "Solve completed..."
  monitor.EndWork()


def createJobInput(task, inputFilePaths):
  ExtAPI.Log.WriteMessage('creating job input')
  inputFilePath = inputFilePaths[0]
  #get param values
  inputValue = task.Properties["Inputs"].Properties["Input"].Value


  #write input file
  ExtAPI.Log.WriteMessage("Writing input value ("+str(inputValue)+") to file
(" + inputFilePath + ")")
  f = open(inputFilePath, "w")


f.write('input='+inputValue.ToString(System.Globalization.NumberFormatInfo.I
nvariantInfo))
  f.close()
def reconnectJob(task, outputFilePaths):
  ExtAPI.Log.WriteMessage('reconnecting job')
  outputValue = None
  outputFilePath = outputFilePaths[0] #I know we only have one specified
based on our definition...so work off of the first entry
  f = open(outputFilePath, "r")
  currLine = f.readline()
  while currLine != "":
    valuePair = currLine.split('=')
    outputValue = System.Double.Parse(valuePair[1],
```

```
System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
  f.close()
  #set output value
  ExtAPI.Log.WriteMessage("Retrieved value (" + str(outputValue) + ") from
file (" + outputFilePath + ")")
  if outputValue == None:
    raise Exception("Error in update - no output value detected!")
  else:
    task.Properties["Outputs"].Properties["Output"].Value = outputValue
def getJobStatus(task, outputFiles):
  ExtAPI.Log.WriteMessage('checking job status')
  outputFilePath = outputFiles[0]
  finished = System.IO.File.Exists(outputFilePath)
  return finished
def cancelJob(task, inputFiles, outputFiles):
  ExtAPI.Log.WriteMessage('performing cancellation clean up')
def status(task):
  return None
def reset(task):
  task.Properties["Inputs"].Properties["Input"].Value = 0
  task.Properties["Outputs"].Properties["Output"].Value = 0
```

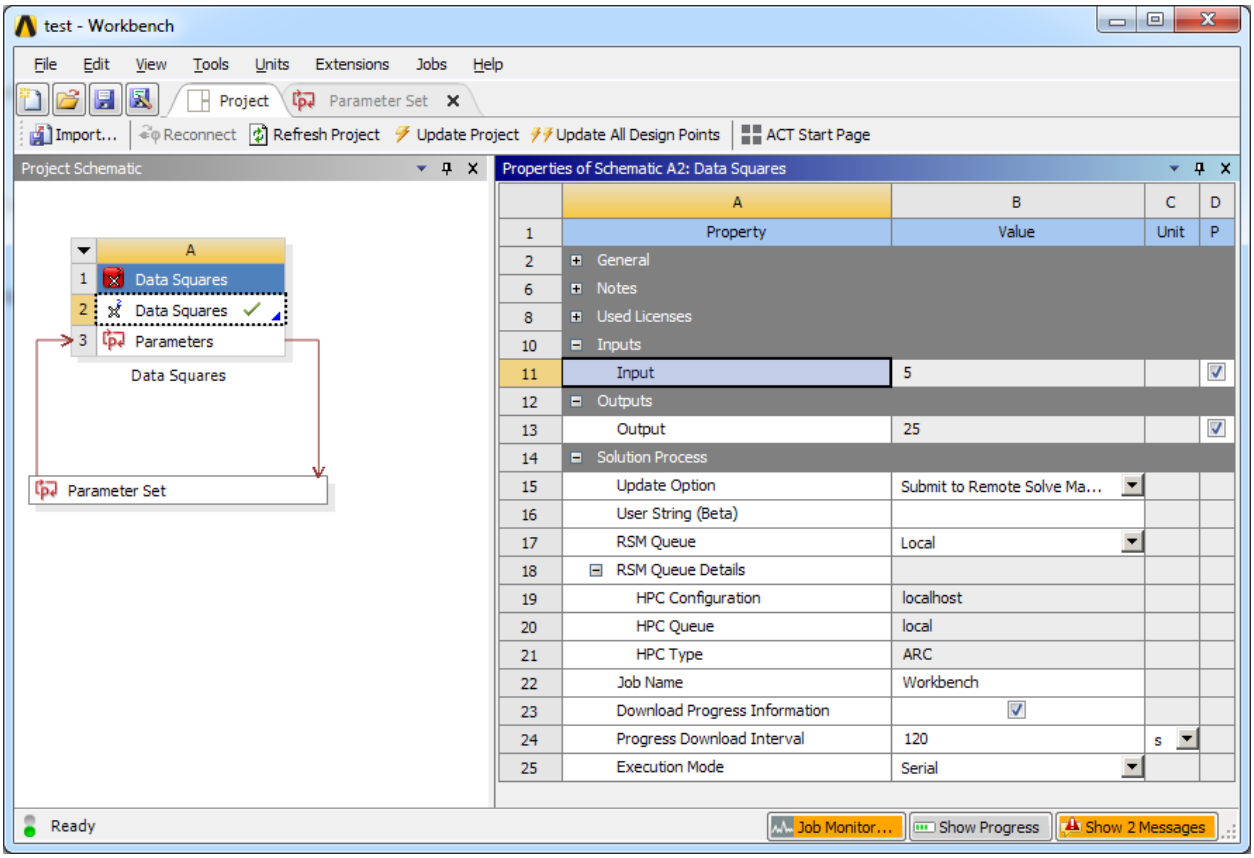
After selecting the task within the **Project Schematic**, you will see **Solution Process** options in the **Properties** view. For **Update Option**, selecting **Submit to Remote Solve Manager** will display RSM management properties:



On task update, you will see your task transition to the pending state while it waits for the job to remotely execute:



When your task has processed the remote results, it will transition its state to “Up-to-Date” and the new data will appear on the output property:



Summary

You should view ACT’s extension concept as a hook into the general processing within Workbench. This generic hook allows you to take off-the-shelf ANSYS Workbench and customize it for your own unique simulation needs.

For legacy External Connection Add-in projects, your migration should be straightforward and natural.

SDK projects typically exercised more nuanced behavior. Understanding the hooks exposed by ACT and re-using your existing code will lead to a smoother transition.

Visit the ACT Resources page on the ANSYS Customer Portal for detailed ACT documentation and Workflow templates, some of which are referenced within this migration guide. Leveraging the Workflow-specific resources will help to ease your transition from our legacy customization products to ANSYS ACT.