Autodesk University 2012

The Answer Is Yes:
Scripting with Autodesk PLM 360 Part 2

Jared Sund - Autodesk

PL3657-P Attend this class to complete your Autodesk PLM 360 PowerTrack. This class builds on "You Can Do That?! Scripting with Autodesk® PLM 360, Part 1." We will outline JavaScript examples and pose customer-inspired examples of customization. Learn how scripting can be used for time-bound workflow, for value-based transitions, and to spawn items in other workspaces.

Learning Objectives

At the end of this class, you will be able to:

- Spawn items in other workspaces through scripting
- Configure approval board and delegations through scripting
- Script time-bound actions and transitions
- Automate process through value-based transitions in scripts

About the Speaker

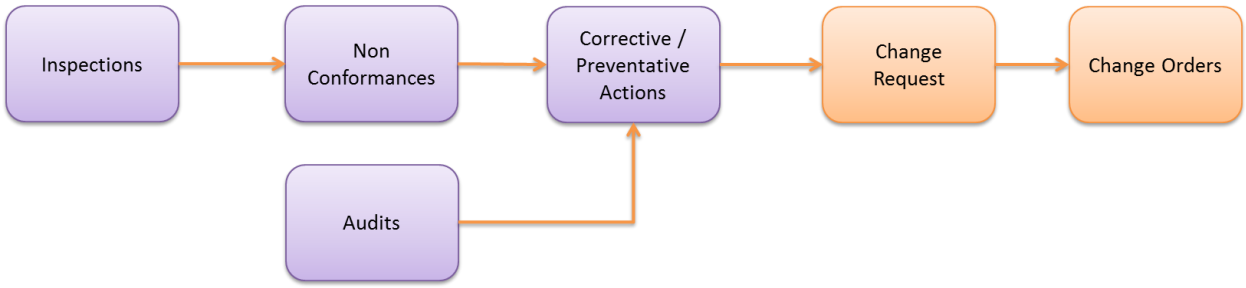
Jared is an Autodesk PLM 360 Product Manager residing in the Lake Oswego, Oregon office. Jared's main focus areas within PLM 360 include: the scripting engine, REST API, Enterprise Integrations, Administration/Configuration, along with other areas of the tool. In addition to managing aspects of the PLM 360 product, Jared is also called upon for his technical knowledge in developing different customer solutions. Jared came to Autodesk in 2011 after spending 10+ years supporting and implementing PLM, PDM, CAD, CAM, and analytical engineering tools for companies that include: Xerox Corp., L3 Communications, Raytheon, and Tektronix. He also holds a BS in Computer Science from Portland State University.

Email: jared.sund@autodesk.com

PLM 360's server-side scripting engine provides tools and functionality to take your PLM 360 solutions to the next level. While scripting is not required to build solutions in PLM 360, the functionality and tools offered here can increase the capability and automation of your solutions. In this course we'll build on the basic PLM 360 scripting knowledge gained from the Scripting 1 class by reviewing some customer inspired standard solutions in more detail.

Standard Solution Spawning Map

PLM 360's scripting engine provides tools to spawn (create) new records in any valid workspace. Spawning new items as show in the map below provides a means to extend our solution workflows into your company's larger business solutions.



Through the use of spawning, the example above (available in the PLM 360 standard solution set) creates a solution that ties our Quality systems (Inspections, Non-Conformances, Audits, and Corrective / Preventative Actions) directly to our Engineering systems (Change Request and Change Orders). As we will see in reviewing spawning solutions, we not only maintain a flow through isolated solutions, but also build in connectivity between the independent items. This process allows us to design a solution that maps and relates individual processes into a larger business solution.

There are three main areas we will focus on to enable the spawning solution from Inspections through Change Orders:

- ✓ The PLM 360 scripting function createItem('WS_WORKSPACE_ID');
- ✓ How to make our scripts manageable with Libraries
- ✓ Creating relationships via pick lists based on a list of records from a workspace

One of the key features that enables spawning behaviors is the PLM 360 scripting function, createItem. The createItem function takes a single argument (the workspace ID, where the record will be created), and returns an item object. Workspace IDs can be located in each Workspace's settings page.

```
var newNonConfItem = createItem('WS_NON_CONFORMANCE');
newNonConfItem.TITLE = 'Test Item';
newNonConfItem.DESCRIPTION = 'This item was automatically created';
```

The returned item (newNonConfItem in the example above) is a pre-defined top level object that is very similar to the standard item object supplied to all scripts. This newly spawned item is pre-populated with all the item properties available within the Workspace the item was created in. With access to a full item definition of the spawned item, we can create and populate new items from within a single script. See the link below for more details on the createItem() function:

http://wikihelp.autodesk.com/enu?adskContextId=PLM360_HELPID_DG_SCRIPTING_REF&language=enu&product=PLM_360

As discussed in the PLM 360 Scripting 1 class, Library scripts can be created to help make our scripts more manageable. The use of functions or objects in Library scripts offers better reuse of the code we write, along with a central location to store the business logic of our scripting solutions. Below are two library examples that encapsulate our "create" and "populate" new item behaviors into a function and a JavaScript object.

In each of the examples below, notice the heavy use of block comments. While it is good practice to comment your scripts, I believe that a good explanation of the details of re-usable code is a must. In addition to commenting the purpose, and required references, arguments, and return values, I also like to document an example use-case for each function or object. This is especially important when a function or object has a complex data-type as an input parameter, or as a return value. In six months you maybe in a position to re-use some of your existing code, or inherit code from someone else, and at that point you'll be very thankful that script was well documented.

createNewItem function:

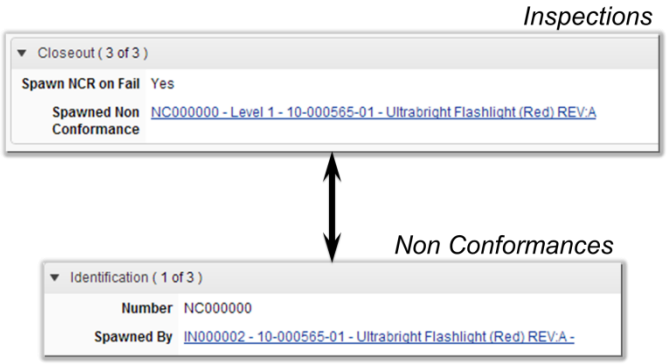
```
/*
functionality: Creates a new item with the passed properties
input   : baseItemName(string) - WorkSpace ID for the new item
        : newItemProperties(associative array) -- Key(string) = New
        Item Field ID, Value = Field's value
return  :
example :
      var newProperties = []
      newProperties['TITLE'] = "This is an example;
      newProperties['QTY'] = 5;

      var newItem = createNewItem('WS_NAME', newProperties);
*/
function createNewItem(baseItemName, newItemProperties){
      newItem = createItem(baseItemName);
      for(var propKey in newItemProperties){
            newItem[propKey] = newItemProperties[propKey];
      }
      return newItem;
}
```

spawnItem object:

```
/*
functionality: Object handler for creating new items
constructor : baseItemName(string) - WorkSpace ID for the new item
functions  : addField (string, var) - FieldID in new item(string),
value for the field
           : addFields (associative array) - Key(string) = New Item
Field ID, Value = Field's value
           : createItem() - creates and returns the requested item
example  :
        var spawnObject = new spawnItem('WS_ID');
        spawnObject.addField('TITLE', 'This is an example');
        spawnObject.addField('QTY', 5);

        var newItem = spawnObject.createItem();
*/
function spawnItem(baseItemName){
    var newItemProperties = [];

    this.addField = function(fieldID, fieldValue){
        if(fieldValue !== null){
            newItemProperties[fieldID] = fieldValue;
        }
    };

    this.addFields = function(propertiesArray){
        for(var propKey in propertiesArray){
            if(propertiesArray[propKey] !== null){
                newItem[propKey] = propertiesArray[propKey];
            }
        }
    };

    this.createItem = function(){
        var newItem = createItem(baseItemName);
        for(var propKey in newItemProperties){
            newItem[propKey] = newItemProperties[propKey];
        }
        return newItem;
    };
}
```

The last topic to discuss in our spawning discussion is how to establish bi-directional association between the spawning item and the newly spawned item. PLM 360 offers a great feature that allow us to create dynamic pick lists based on the records that exist within a workspace. Through the use of these pick lists, we can define fields within our items that can

link or create a relationship between individual records (within the same or workspace or from different workspaces). The fields driven by pick lists can be used in the scripting engine to traverse across multiple items and values can also be set from our scripts. From the spawning map listed above, we can see linked relationships between our Inspection and spawned Non-Conformance records.



When spawning the Non Conformance record from an Inspection record, we are able to populate the SPAWNED_BY (Non-Conformance) field with our current Inspection item. We can then use the return object (newly spawned Non-Conformance item) from our createNewItem function to populate the SPAWNED_NON_CONFORMANCE field in the spawning Inspection record.

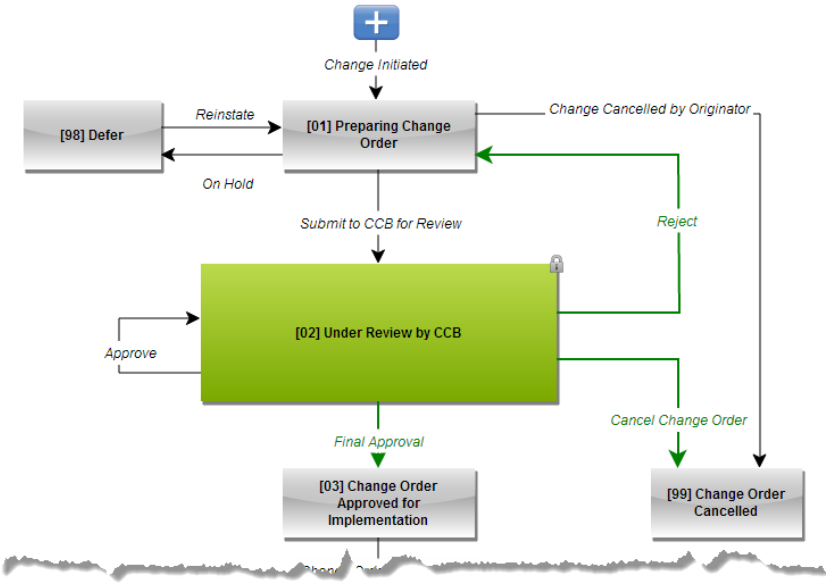
```
newProperties.SPAWNED_BY = item;

var newItem = createNewItem('WS_NON_CONFORMANCE', newProperties);

if(newItem !== null){
   item.SPAWNED_NON_CONFORMANCE = newItem;
}
```

Remember from our createNewItem function that the newProperties associative array is a list of the fieldIDs and values that will populate our newly spawned item. Since the SPAWNED_BY field is a record backed pick list, we can set our current item as value of this pick list. Additionally, the newItem returned from our function can also be used to populate the record backed pick list, SPAENED_NON_CONFORMANCE. This spawning behavior allows the creation, population and referencing between our items all in a single script or operation.

Workflow Approval Board

The Workflow Approval Board solution was developed to block or allow workflow actions based on each individual record's selected approvers. Approvers can be selected from a list of all PLM 360 users, and/or from a predefined list of approvers.

Change Order WF Approval Board



In this example we will not go through the Workflow Approval Board scripts in great detail, but rather we will discuss an engineering process you may choose to use when faced with the need to create a large or complex scripting solution.

Since PLM 360 is easy to use, modify, and very agile, there is no need to spend days, weeks, or months developing a lengthy requirements document. I find that it is much easy to just establish a short/concise list of the solutions requirements. As you can see from the requirements list below, our WF Approval Board requirements are indeed short and straight to the point.

Requirements	
1.	Different approvers for each record
2.	Different number of approvers for each record
3.	Must accommodate ad-hoc and predefined users
4.	Upon rejection, all approvers must re-approve
5.	Must be scalable to be used on multiple different workflows
6.	Must be scalable to accommodate changes in biz processes

Once we have received and validated our requirements list, as the PLM 360 admin, developer, or business administrator we should analyze the requirements and construct a short list of how a solution could be developed within PLM 360 to fit our requirements. In software development, this we would likely see this as a Waterfall method, or iteratively as a Modified Waterfall method.

Analysis	
By record approvers	Pick lists - Users, ALL by Name
Approver lists	Pick list - Approval Lists
Transition blocking	Condition Scripts
Approval history	Item.workflowActions
Manageable scripting	Library script

After we've validated our analysis against our requirements, we can start to think about how to design a solution. One common approach to developing a solution is to skip the design phase altogether and just start coding… Taking this approach is generally a successful route for small simple scripts, but spending a little time thinking about the design for a solutions like WF Approval Board will likely result in a better/quicker solution.

The forward thinking required to design a solution is sometimes difficult, most certainly when you may be new to toolset like PLM 360. I suggest breaking your design into two areas: what do we need to know or what questions should our solution answer, and what code can we re-use? Some solutions may need more topics covered in the design, but this generally a good starting point.

For the WF Approval Board, our general requirements questions could be:

- How many approvals are needed?
- Is the current user an approver for this record?
- Has this approver already approved?
- Does this record require more approvals?

Also, what code could be reused?

- getApprovers()
    - getUnique()
    - inArray()
- requiresMoreApprovals()
- userAlreadyApproved()

Remember, PLM 360 is agile and easy to use, so our design (like our requirements and analysis) does not need to be overly complex. Just from the simple design we have laid out, we are ready to start building our solution.

```
//condition return value
var userCanApprove = false;

//start by getting the first and last name of the usr and check with
the multi-select
var usr = Security.loadUser(userID);
```

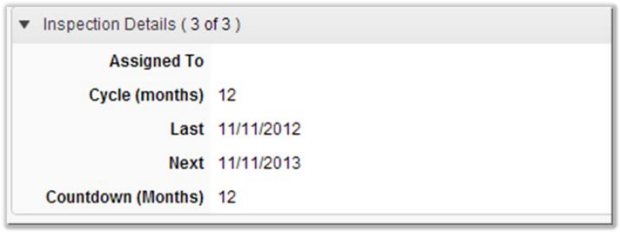
```
var usrName = usr.lastName + ", " + usr.firstName;
var wfSteps = item.workflowActions;

//generate a list of approvers from APPROVAL_LISTS and
APPROVALS_REQUIRED
var approvers = getApprovers();

//main program starts here
if(approvers.length > 1){ //more than one approver is needed
  if(inArray(approvers,usrName) === true){ //current user is in list of
approvers
    if(userAlreadyApproved(wfSteps,userID) === false){ //current user
hasn't already approved
     if(requiresMoreApprovals(wfSteps, approvers.length,false)===true){
     //still needs more approvals
       userCanApprove = true; //ok for the current user to approve
     }

    }

  }
}

returnValue(userCanApprove);
```

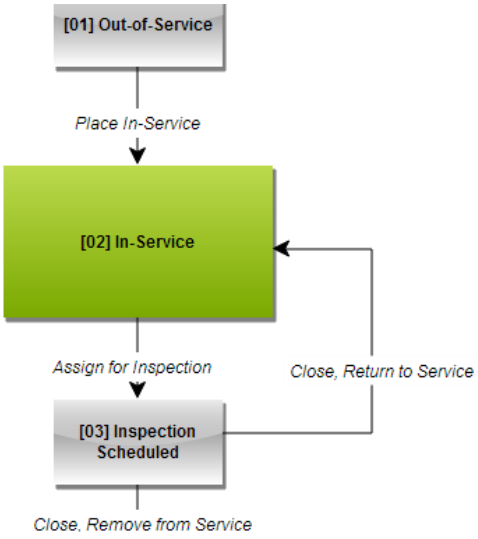
From the sample condition script above, you will see how our design of questions and functions (re-usable code) has helped us to create a solution that meets our requirements. Also, note that adding new requirements (maybe modifying our existing questions or adding new ones), should easily trickle back through our analysis, design, and then directly into our code.

Even complex solutions developed within PLM 360 should be easy to create and manage, just like you've seen here with WF Approval Board.

Temporal (Time) Based Transitions

With the use of condition scripts, WF Actions (transitions) can be blocked based on difference or the delta between a set dates and the date a WF Action is requested. In our example, we will look at how we block equipment maintenance inspection tasks for some period of time. Each Inspection record contains a field to determine the inspection cycle time in months, and values to indicate when the next and last inspection occurred. Our script will be designed to block the "Assign for Inspection" transition, keeping the equipment in service until the next inspection cycle comes due.





To create this solution, we will require two scripts: an action script to create the next inspection date based on the current date and the cycle period, and a condition script that will block the "Assign for Inspection" transition until the equipment maintenance record is within one month of the next inspection cycle.

First, let us examine the Action Script (GenSetNextLastDates) that is used to calculate and store the record's inspection dates.

```
//returns the formatted date, plus some number of months
function getDateAdjusted(addMonths){
    var currentTime = new Date();
    currentTime.setMonth(currentTime.getMonth() + addMonths);
    return (currentTime);
}

//gets the next cycle time in base 10 value
var monthsTillNext = parseInt(item.CYCLE_MONTHS,10);

item.LAST = getDateAdjusted(0);
item.NEXT = getDateAdjusted(monthsTillNext);

//clear the assigned to user
if(item.ASSIGNED_TO !== null)
{
    item.ASSIGNED_TO = null;
}
```

As you can see from the action script, we have a function to calculate our next inspection date. This function could be put into our generic dates library set of functions, as it maybe something we'll need to use later in another script. Other than that, the script simply sets the last and next

inspection dates. We've also taken the liberty of clearing out the ASSIGNED_TO field, to ensure the proper inspector is chosen for each new inspection cycle. Now let us take a look at the condition script.

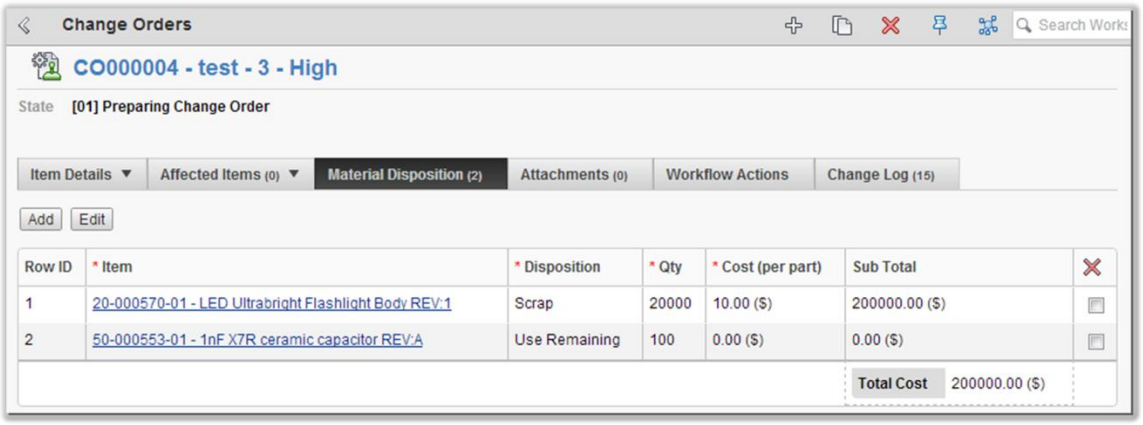
```
/*
Setup:      Requires computed field item.COUNTDOWN_MONTHS
Purpose:    To only allow withdraw if Countdown period
              is less than or equal to a month
Method:     test COUNTDOWN_MONTHS value
Result:     user has transition if less than one month to countdown
*/

//the calibration period in months
var calibrationPeriod = 1;

//Set the initial value
var withinCalPeriod = false;

if(item.COUNTDOWN_MONTHS !== null &&
   item.COUNTDOWN_MONTHS <= calibrationPeriod){
        withinCalPeriod = true;
}

returnValue(withinCalPeriod);
```

Again, a very simple script that tests the COUNTDOWN_MONTHS (computed field, to calculate the months difference between now and the next inspection cycle) against our calibration period (1 month). The condition script is normally blocking, but if our test is true the transition will be allowed.

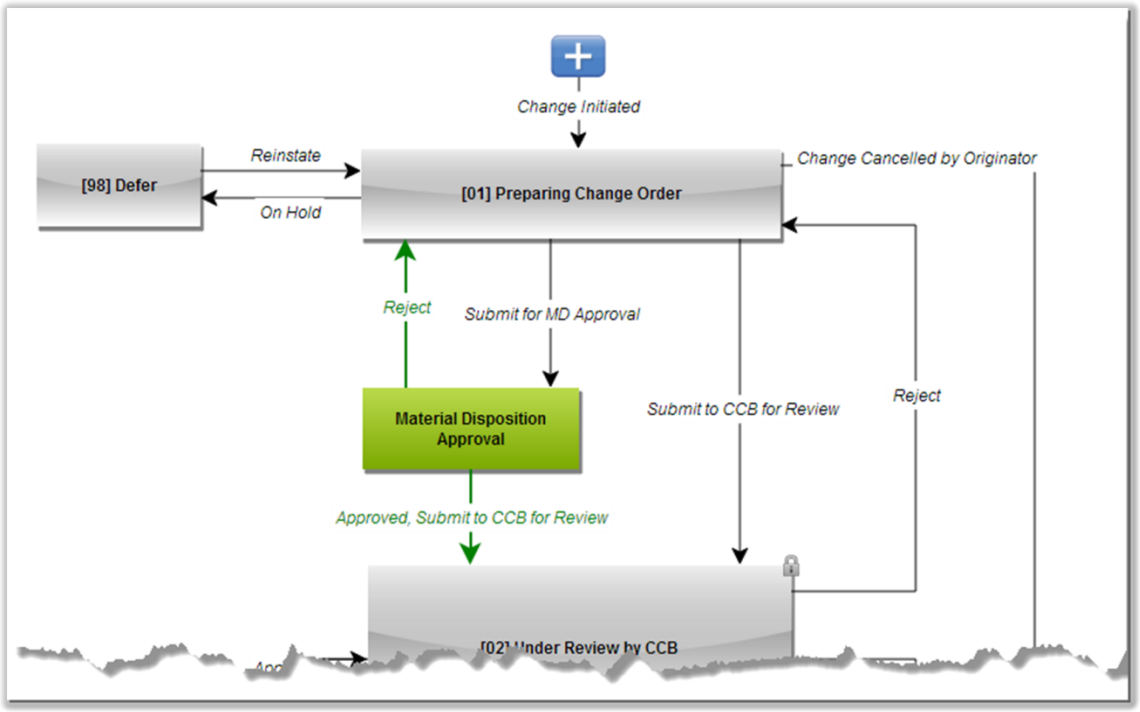
This is a very simple example of temporal based transition blocking using condition scripts. The use of a condition script to block our inspections until within a month of the inspection cycle performs two main tasks for us: maintains our business process, and secondly keeps this task from our Outstanding Work list until it is time to perform it.

Value Based Transitions

Like the temporal based transition blocking condition script, we can also create condition scripts that block based on values. In this example, I have added a Material Disposition (MD) form (using the grid tab) to our existing Change Orders. The MD offers a location within our Change Order to disposition materials affected by this change while calculating the total scrap and re-work costs associated with our Change Order.



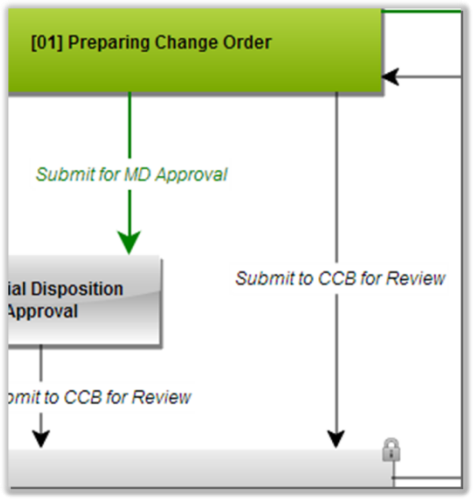
In addition to adding Material Disposition to our change order, we've also added some changes to our Change Order workflow to accommodate MD approvals.



In this example, our business requires any Change Order with material costs greater than $1,000 to go through a Material Disposition Approval, and two different sets of MD approvers: a set for MD costs less than $100,000 and another set for Change Orders with greater than or equal to $100,000 in MD costs.

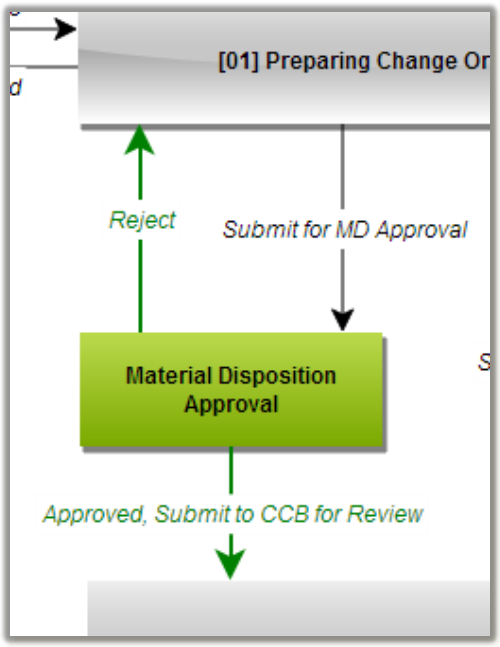
For our first condition, we need to calculate the total MD costs for our Change Order to determine which transition the Change Order can take. If the total cost is greater than $1,000,

then we need to block the submission to CCB and allow the transition to MD Approval. If the opposite is true, then we need to reverse the transition blocking.




```
/*
Purpose:    Determines whether or not material disposition is required
Method:     Utilize the MatDisp class requiresApproval function
Result:     True for MD approval when total costs is greater than n
*/
var returnVar = false;

if(item.descriptor.ownerID === userID){

    var mdObject = new MatDisp(item.grid);

    if(transID === 44 &&
          mdObject.requiresApproval === false){
        returnVar = true;
    }
    else if(transID === 194 &&
                mdObject.requiresApproval === true){
        returnVar = true;
    }
}

returnValue(returnVar);
```

Notice the use of the MatDisp object, we'll cover that after reviewing the two condition scripts. From the script above, you can see how we have abstracted the value of the material disposition and the total cost calculation into our object. With the simple requiresApproval function, we can keep this script very clean and easy to manage. Now let us have a look at the condition to determine whether the current user is a MD approver for this Change Order.




```
/*
Purpose:    Allows approvers based on MD costs
Method:     Utilize the MatDisp class isApprover function
Result:     True only for approvers of value based groups
*/
var mdObject = new MatDisp(item.grid);
returnValue(mdObject.isApprover(userID));
```

If you remember, we have two different sets of MD approvers based on the overall cost of the MD. Again by moving much of the business logic into our MatDisp object, we made this condition script very simple. We simply return true or false based on whether the current user (userID) is an approver or not.

Now for the meat of our business logic, the MatDisp object. In this library script object, we are able to create a single code base to house values, and logic needed to maintain our MD solution. In this library object, we can easily change threshold values, and specific groups of approvers. If a new requirement added a third set of approvers (for MD costs between $50,000 and $100,000) to our MD solution, we could easily make the change here in our object.

```
MatDisp = function(passedGrid){
    var _grid = passedGrid;
    var _maxCostWOApproval = 1000.00;
    var _costThreshHold = 100000.00;

    var rollupChangeCosts = function(){
        var totalCostOfChange = 0.00;
        for(var rowIndex in _grid){
          totalCostOfChange +=
           _grid[rowIndex].QTY * _grid[rowIndex].COST;
         }
        return totalCostOfChange;
    };

    var _totalCostOfChange = rollupChangeCosts();

    this.__defineGetter__("requiresApproval", function(){
       return (_totalCostOfChange > _maxCostWOApproval);}
     );

    this.isApprover = function(passedUserID){

        if(_totalCostOfChange < _costThreshHold &&
Security.inGroup(passedUserID, 'Material Disposition Approvers - Low')
=== true){
            return true;
        }
        else if(_totalCostOfChange >= _costThreshHold &&
Security.inGroup(passedUserID, 'Material Disposition Approvers - High')
=== true){
            return true;
        }
        return false;
    };
};
```

Integration

Make sure you attend this class, as I'll demonstrate new PLM 360 scripting integration technology! For those that cannot attend, don't worry I'll provide updates to this document following the class.

This concludes our PLM 360 scripting 1 and 2 classes. You now have a good understand of PLM 360's server-side scripting engine, and few different solution examples to help you modify existing scripts and add automation and advanced functionally to your PLM 360 solutions.

In the PLM 360 script 1 class we examined:

- ✓ An overview of PLM 360's server-side scripting with JavaScript
- ✓ PLM 360 scripting types and events
- ✓ PLM 360 scripting item object and functions
- ✓ Condition, Validation, Action, and Library "Hello World" scripts
- ✓ Standard Script Review:
    - ✓ Advanced Sequence Generator (ASG)
    - ✓ InspectionPassedFailed Condition Script's access to the grid

In this PLM 360 Script 2 class we expand on the following advanced script topics:

- ✓ Spawning items from Scripts
- ✓ Workflow Approval Board
- ✓ Temporal and Value based Transitions
- ✓ Integration