Active Page Item Developer - Cookbook

By Kris Coppieters

1. Introduction

The *Active Page Item Developer Toolkit* is a system for Adobe® InDesign® CS, CS2,CS3 and CS4 which assist the development of JavaScript-based solutions within InDesign. The purpose of this manual is to show through examples how its features are used. This manual was last updated for version 1.0.47 of the *Active Page Item Developer Toolkit*.

2. Getting started

This section of the cookbook explains how to create simple Active Page Items.

The *APID ToolAssistant* plug-in is part of the *Active Page Item Developer Toolkit* and allows individual page items in an InDesign document to become active components.

Any page item can observe itself and/or one or more other page items, and can react to changes to either itself or the observed items. Active Page Items can also look out for external events and react to them.

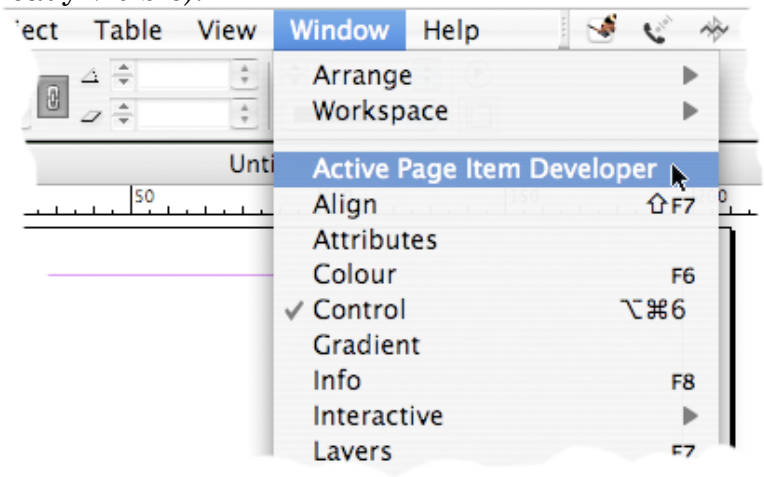
We'll start with very simple examples, and gradually build up towards more complex examples.

The scripting language used is JavaScript (ExtendScript).

2.1. Reacting to modifications

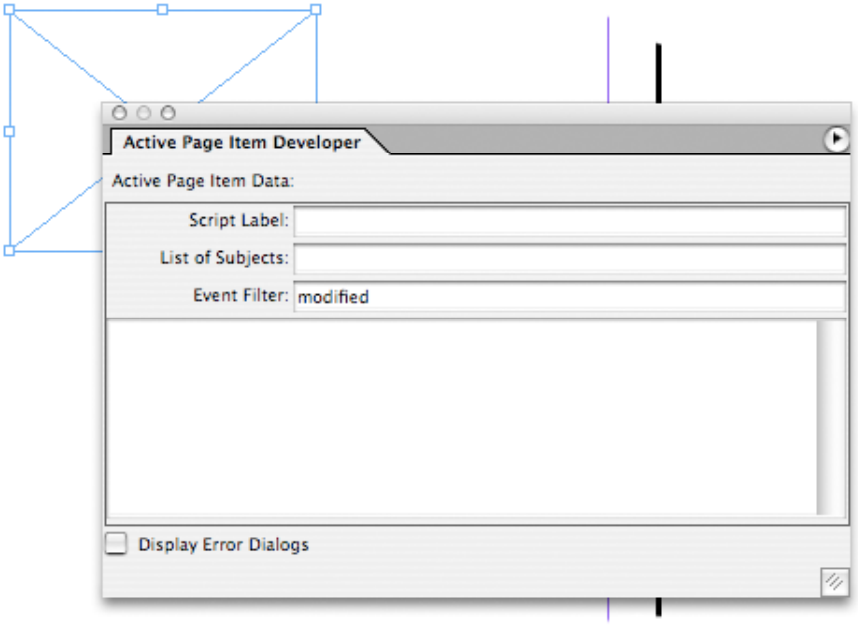
Make sure the *APID ToolAssistant* plug-in is properly installed. Create a new, empty document in InDesign CS, CS2, CS3 or CS4.

Then create a frame of some kind on the first page. Use the *Window – Active Page Item Developer* menu item to make the *Active Page Item Developer* palette visible (if it is not already visible).



.

Select the frame, and type the text `modified` in the *Event Filter* field on the *Active Page Item Developer* palette.

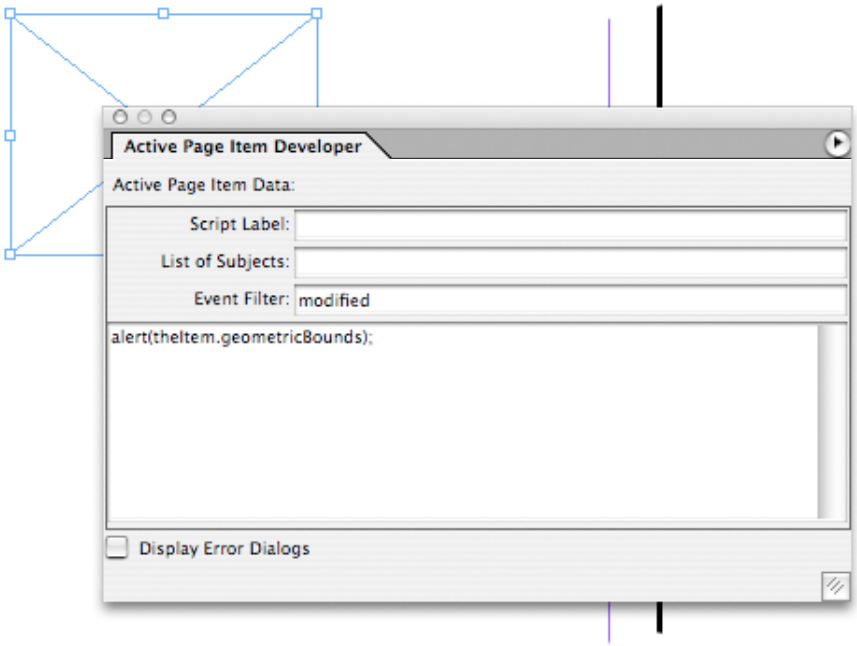


This tells the frame that it needs to keep an eye out for modifications to itself and execute the script that we'll enter in the larger scrolling text area on the *Active Page Item Developer* palette, which is known as the *Script* or *JavaScript* field.

Type the following script:

```
alert(theItem.geometricBounds);
```

into the *Script* field exactly as shown.



This one-liner will bring up a dialog with the current item's bounds.

Together, these entries mean: "Tell me when the frame is modified, and when it is, show me the geometric bounds".

Make sure you hit the *Tab* after having typed the word `modified` so InDesign is notified of the change; forgetting to hit the *Tab* key often causes InDesign to ignore whatever you typed into the palette.

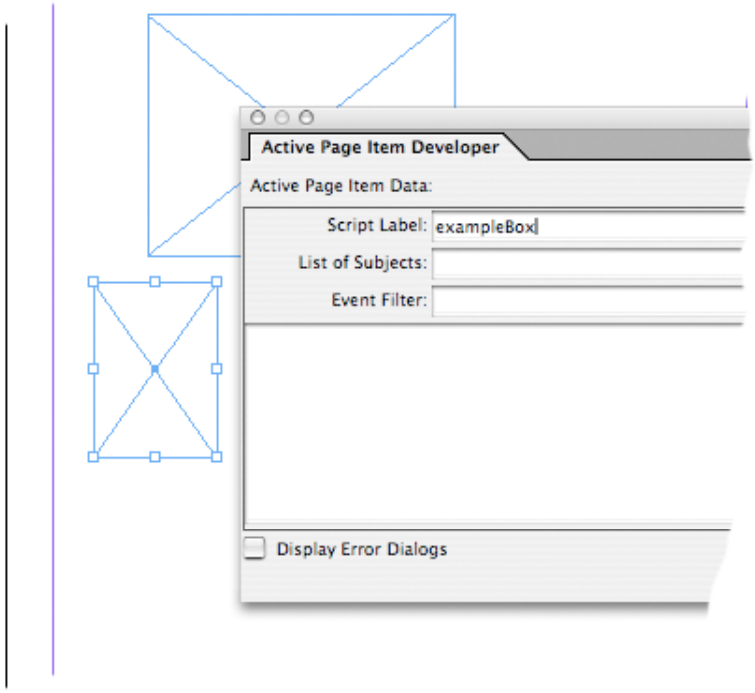
Now each time you move the frame a bit, a dialog should pop up with four floating-point numbers. Not very useful, no – it just demonstrates some of the basic principles behind the *Active Page Item Developer*.

2.2. Observing a subject

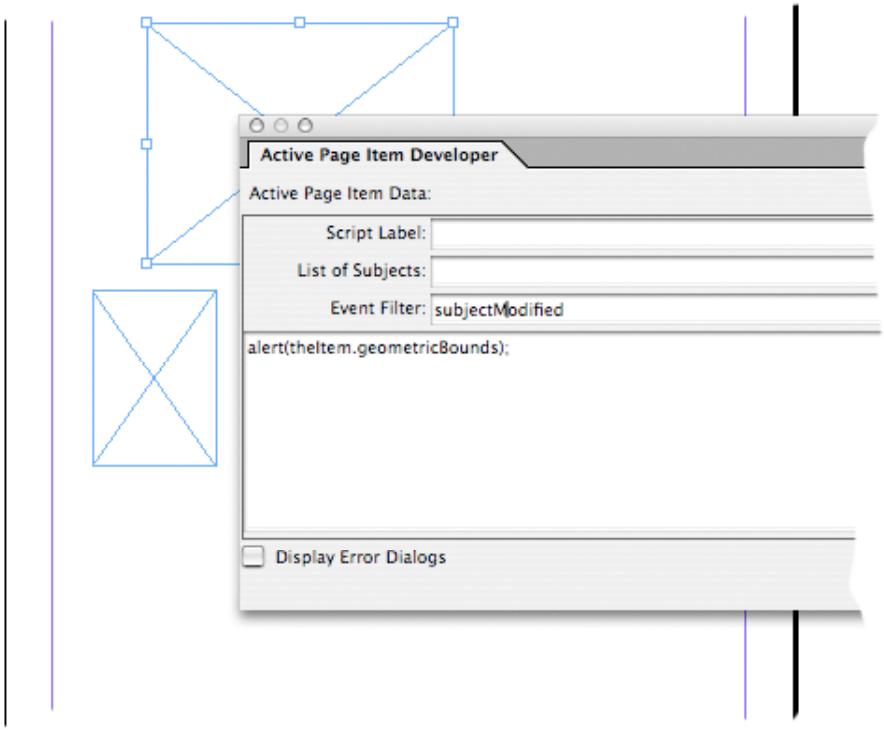
Objects can be observing one another. To help with this, we need to have a way to refer to a particular page item. The *Script Label* field on the *Active Page Items* palette allows us to assign a name to a page item.

The *Script Label* is often referred to as the *Script Tag*.

Create a second frame on the page, and while it is still selected, name it `exampleBox` in its *Script Label* field. Hit the *Tab* key afterwards.

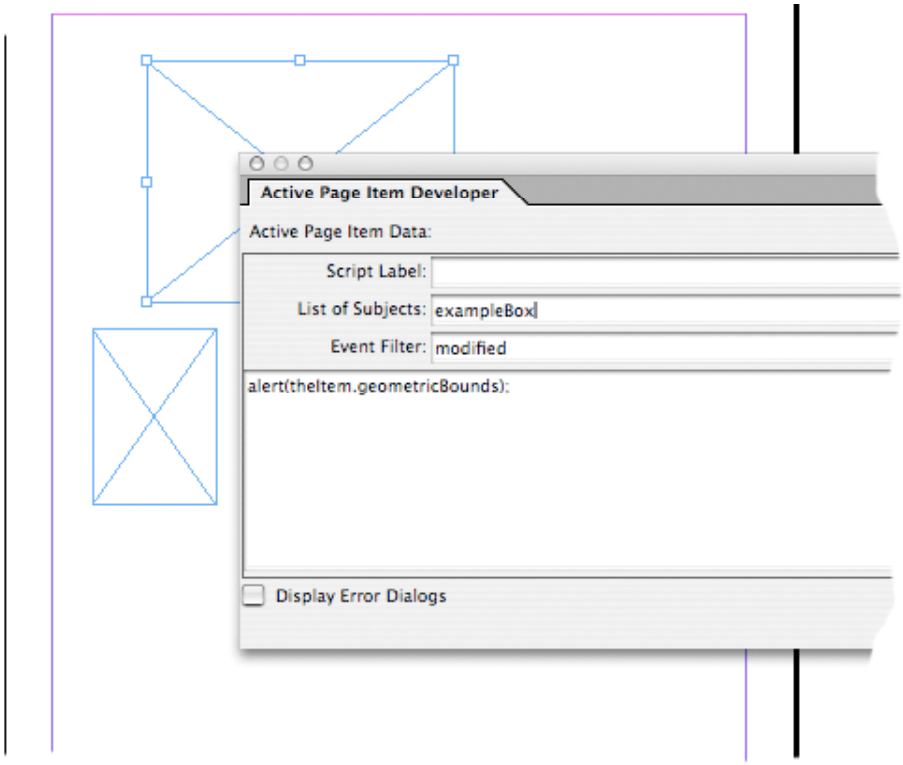


Then select the previous frame again, and change the contents of the *Event Filter* field from `modified` into `subjectModified`. Make sure to respect upper- and lowercase letters, and hit the *Tab* key afterwards.

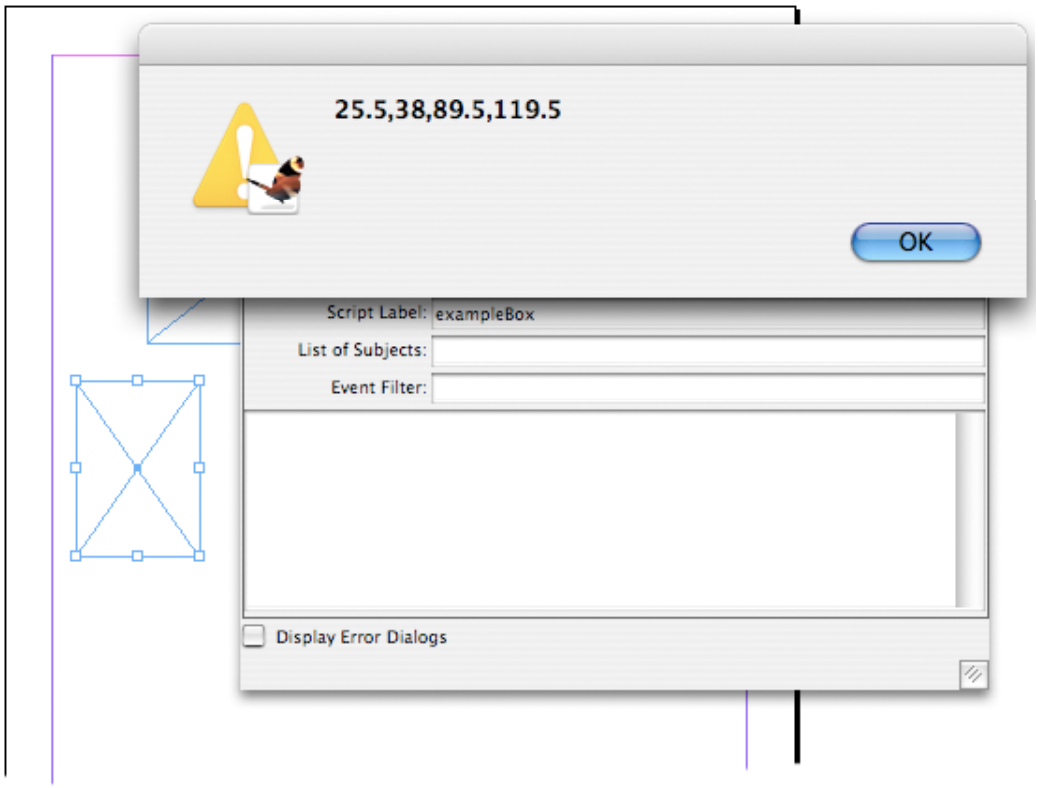


Finally, enter `exampleBox` in the *List of Subjects* field for the first frame, and hit the *Tab* key.

This configuration has the following effect: the original frame is now watching out for modifications in any of its subjects (`subjectModified`). Right now, it only has one subject – the `exampleBox`. If and when the `exampleBox` is modified, the script on the observing frame will be called.



Move the *exampleBox* around a bit. Each time you move the *exampleBox,* a dialog comes up with floating point coordinates; click *OK* to make it go away.



Take note that the displayed coordinates *don't change* on each move – that's because *theItem* in our JavaScript always refers to the page item to which the script is attached. As long as the original frame is not moved, the geometric bounds shown won't change.

Moving the *exampleBox* triggers the execution of the script, but the data presented in the dialog relates to the other frame.

2.3. Referring to a subject

If we want to show the geometric bounds of the *exampleBox* instead of the observing frame, we need to change the script a little bit: the *eventSource* attribute of *theItem* refers to the cause of the *subjectModified* event.

Change the script so it reads:

```
alert(theItem.eventSource.geometricBounds);
```

adding the property *eventSource* after *theItem*.

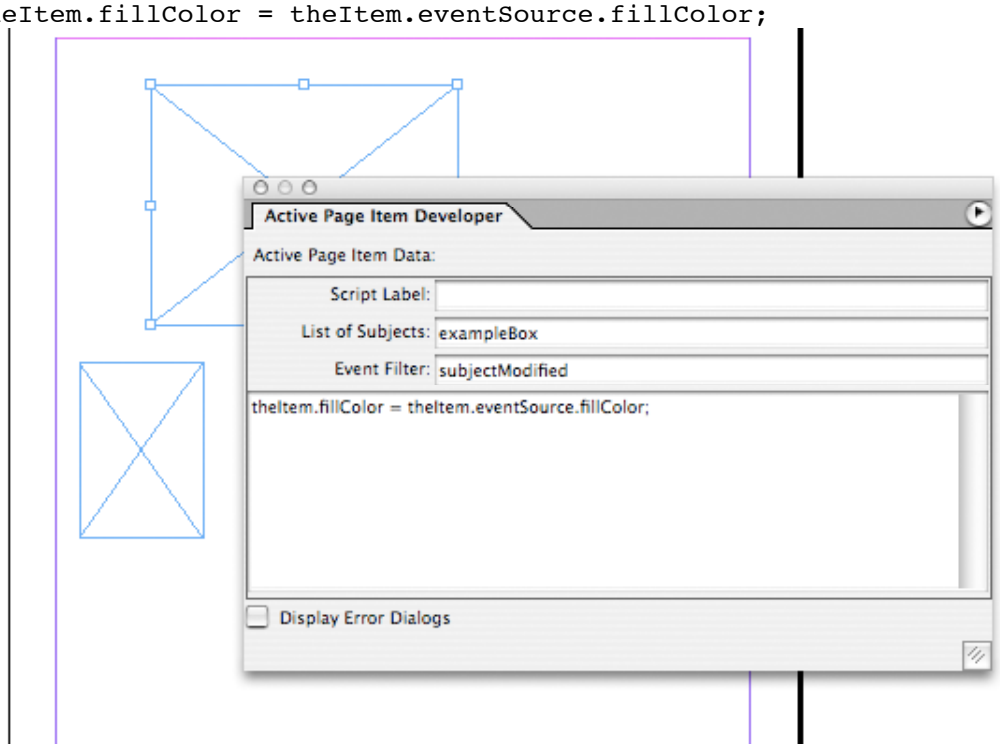
If you now move the *exampleBox* the numbers will be different each time because they reflect the bounds of the *exampleBox*.

The *eventSource* property comes in handy when a single frame is observing multiple subjects – when one of the subjects causes an event to be caught by the observer, you can use *theItem.eventSource* to find out which subject caused the event.

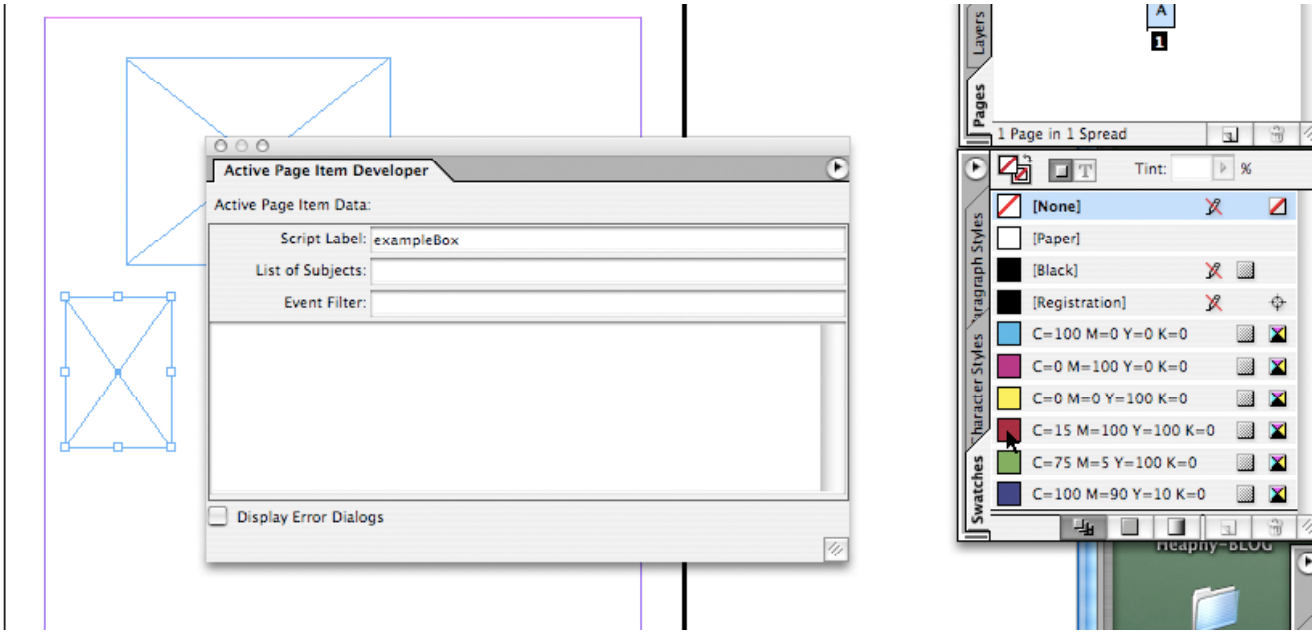
2.4. Copying the fill color

Now, we can easily change the script to copy the fill color of the *exampleBox* into the observing frame, instead of bringing up a dialog. Change the script attached to the observing frame so it reads:

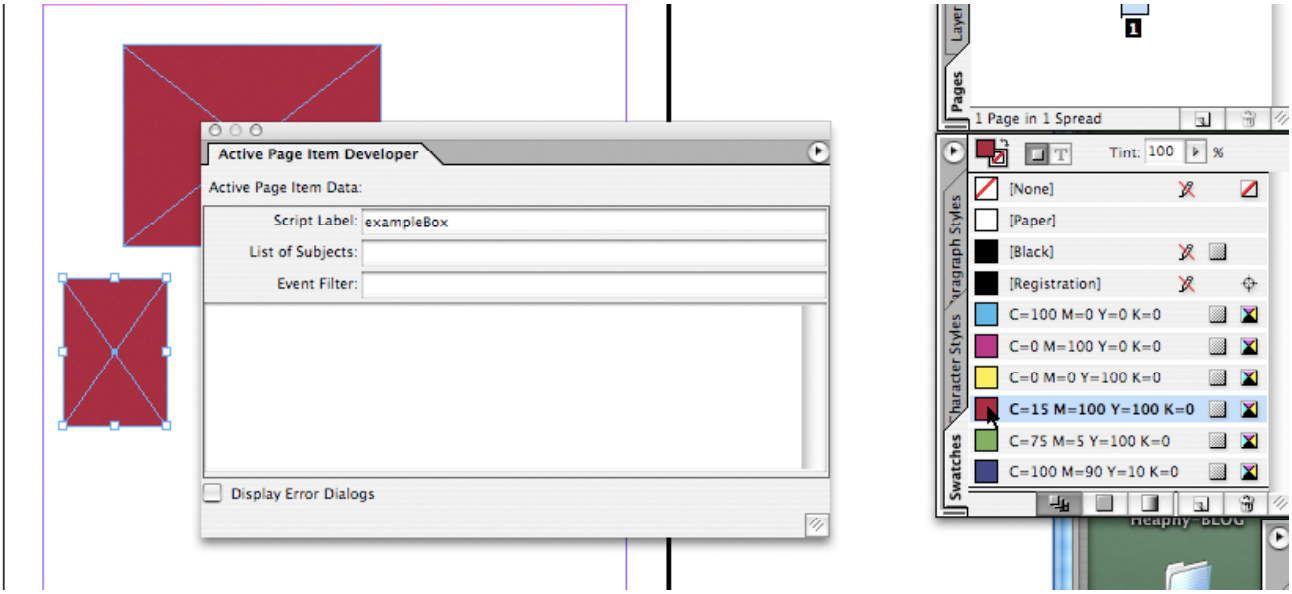
```
theItem.fillColor = theItem.eventSource.fillColor;
```



The whole setup is now configured to automatically copy the color of the *exampleBox* into the observing frame each time the *exampleBox* frame changes.

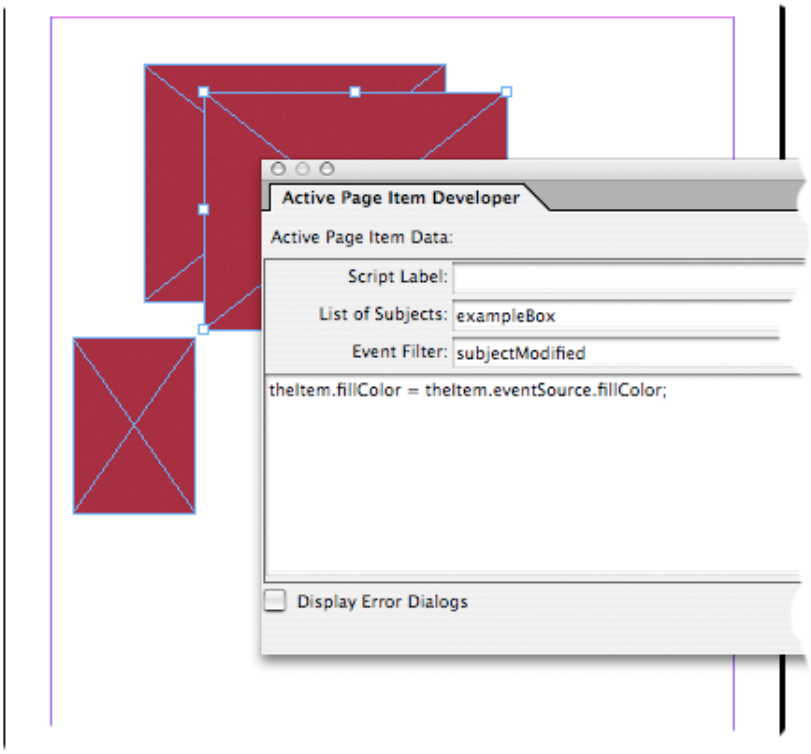


Assign a fill color to the *exampleBox*; the observer will immediately fill itself with the same fill color.

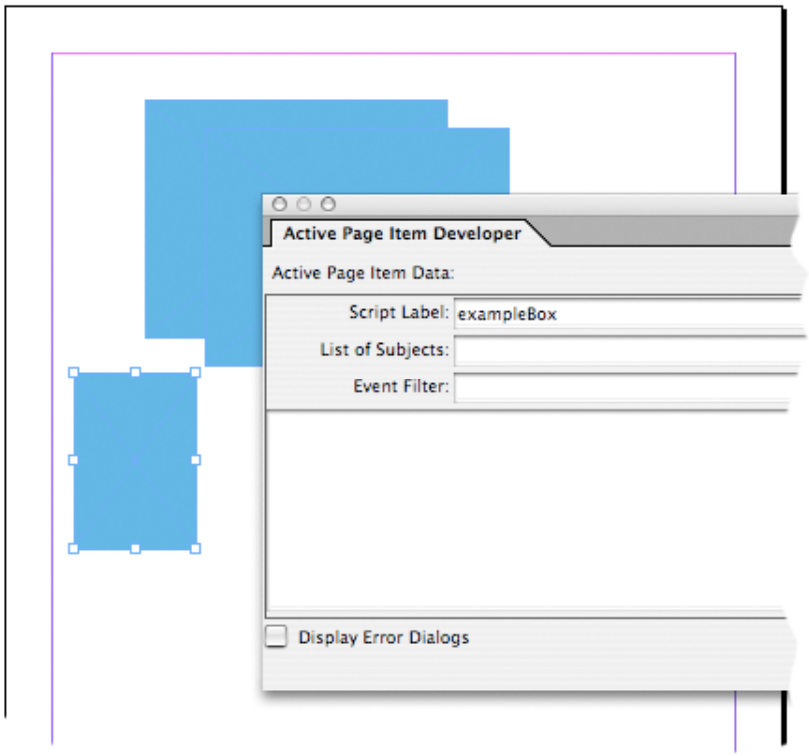


2.5. More than one frame can observe

Copy the observing frame, so there are two identical frames, with identical *Event Filter* field contents and identical JavaScripts.

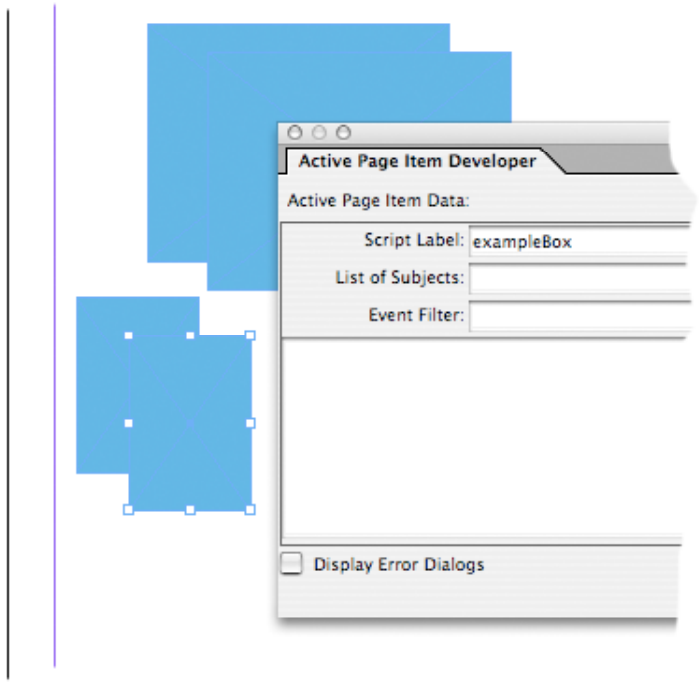


Change the `exampleBox` fill color. Note that both observers change color.

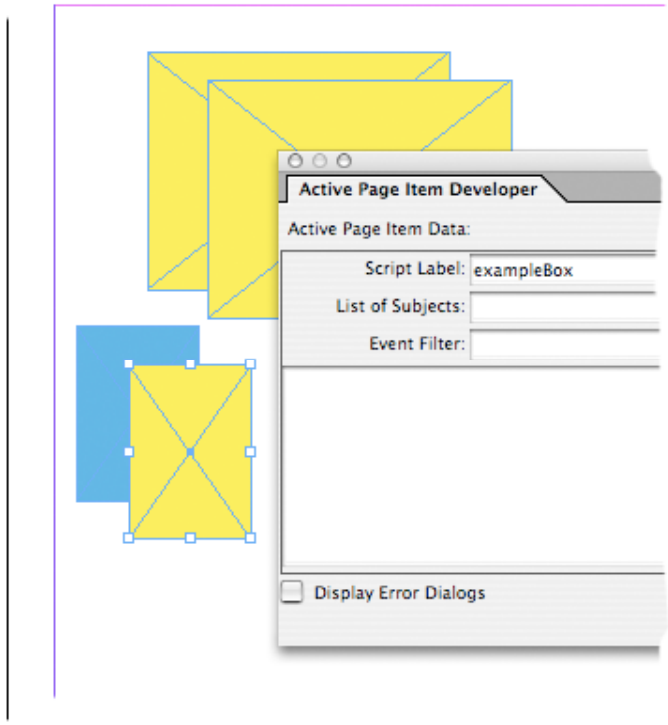


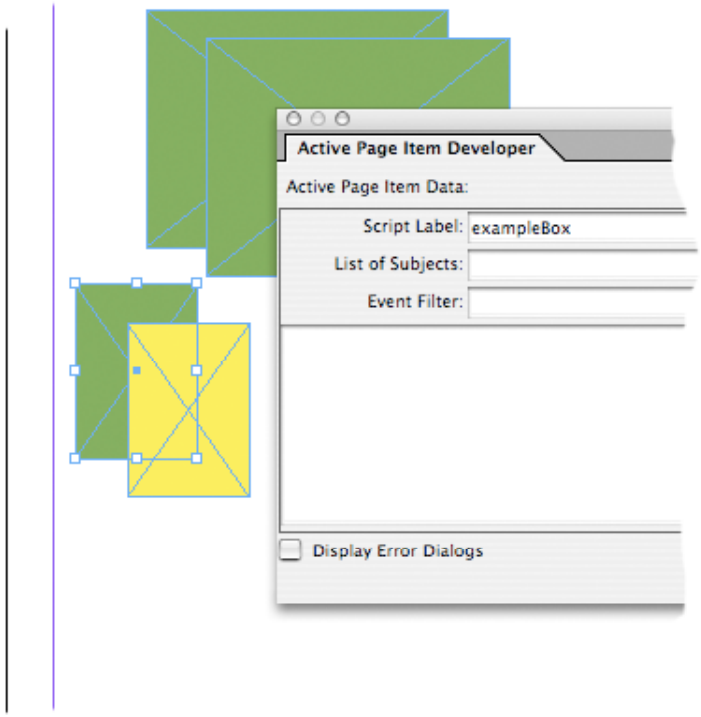
2.6. Script Labels do not need to be unique

Copy the *exampleBox*, so there are two identical frames, both carrying script label *exampleBox*.



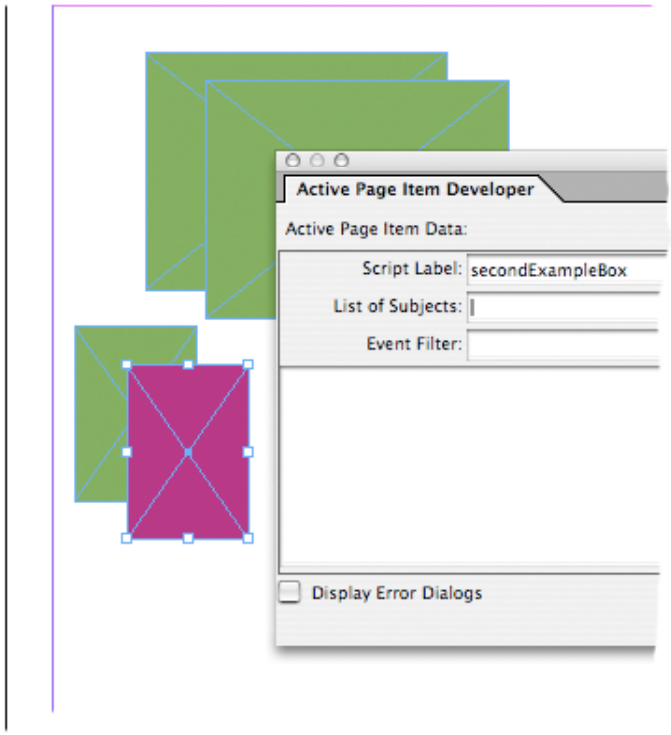
Modify the fill color of one and then of the other. Note that both observers react to changes in both copies of *exampleBox*.



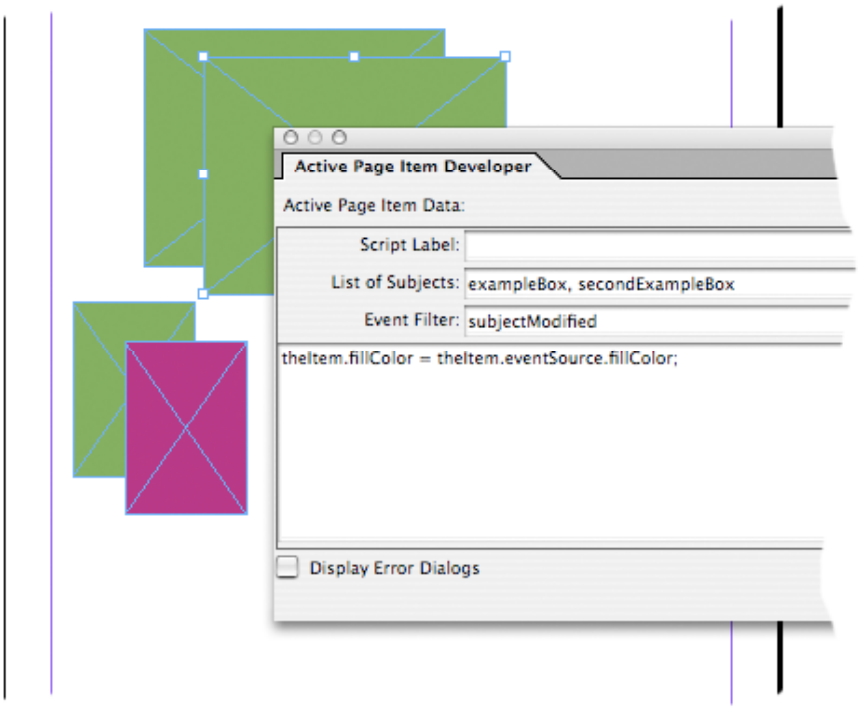


2.7. Reacting to multiple subjects

Change the script label of the second `exampleBox` and call it `secondExampleBox` (do not forget to press the *Tab* key afterwards). Change the fill color and note that the observers now ignore the color change.

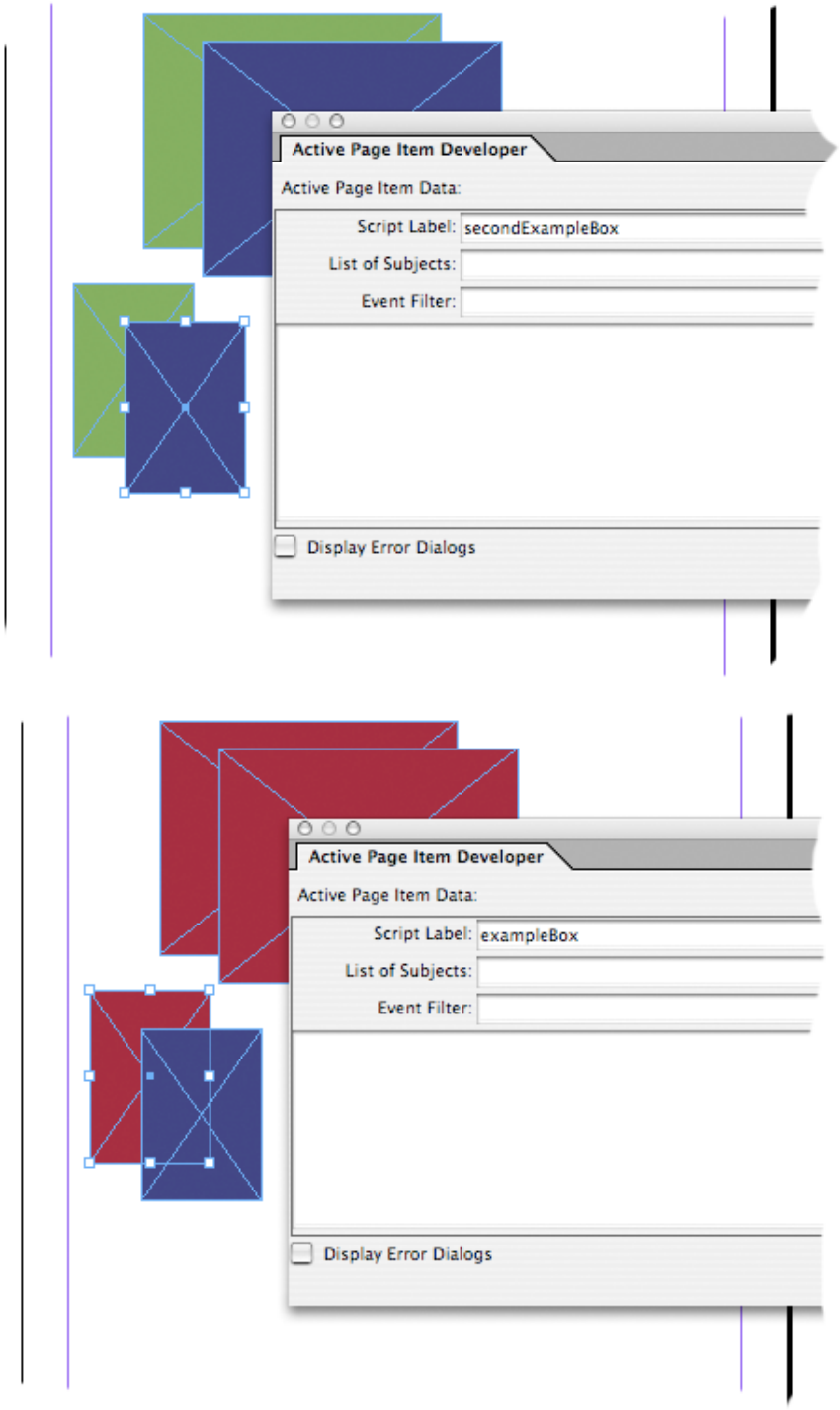


Pick one of the observers and change the *List of Subjects* field to `exampleBox, secondExampleBox` (i.e. two subjects).



Now, one of the observers only observes `exampleBox.` The other observing frame that we just modified observes both `exampleBox` as well as `secondExampleBox`.

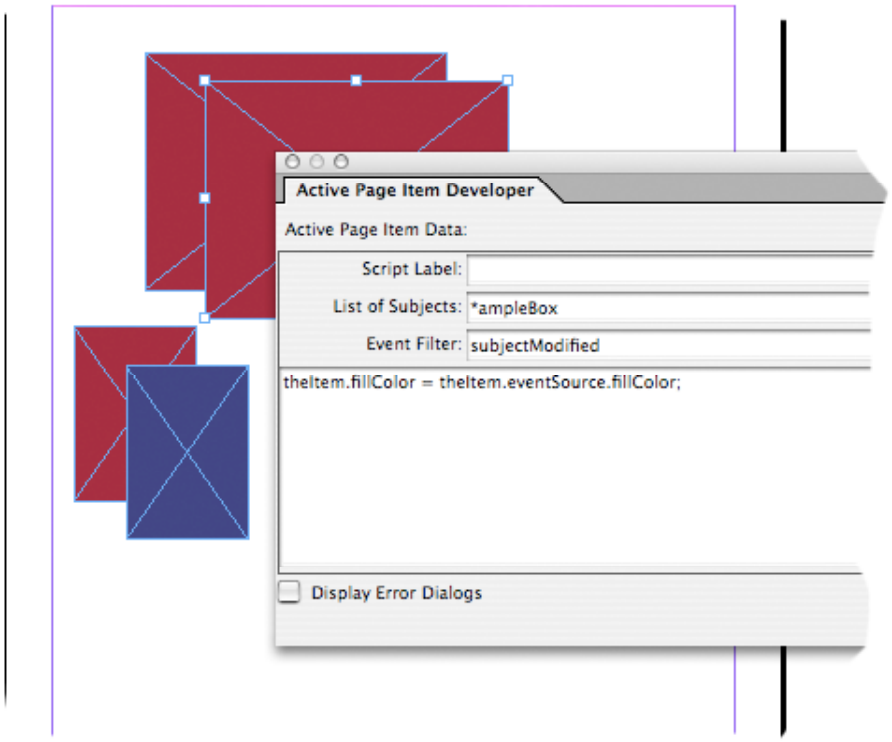
Change the color on both example frames and see what happens.



2.8. Using wildcard expressions in subject lists and event filters

The *List of Subjects* field also accepts wildcard expressions – it allows one or more wildcard expressions, separated by commas.

Change the observer's *List of Subjects* field from `exampleBox, secondExampleBox` to just `*ampleBox` (do not forget to press the *Tab* key after the change). This is a wildcard expression that matches both `exampleBox` and `secondExampleBox`. Note that the setup still works as before – the wildcard expression 'captures' both frames.



Noteworthy is that a *List of Subjects* field that contains just a * will match *any* subject – even subjects that don't have a script label assigned (in other words, it also matches subjects with an empty script label).

The *Event Filter* field uses a similar mechanism: one or more desired events or wildcard expressions for events can be listed, separated by comma's.

The following characters are used for wildcard expressions:
* means 'zero or more characters'.
? means 'exactly one character'.
! means 'not'.

For example:
?*
is a wildcard expression that matches at least one or more characters.

Consequently,
!?*
is a wildcard expression that matches 'not one or more' – in other words it only matches zero-length (i.e. empty) labels.

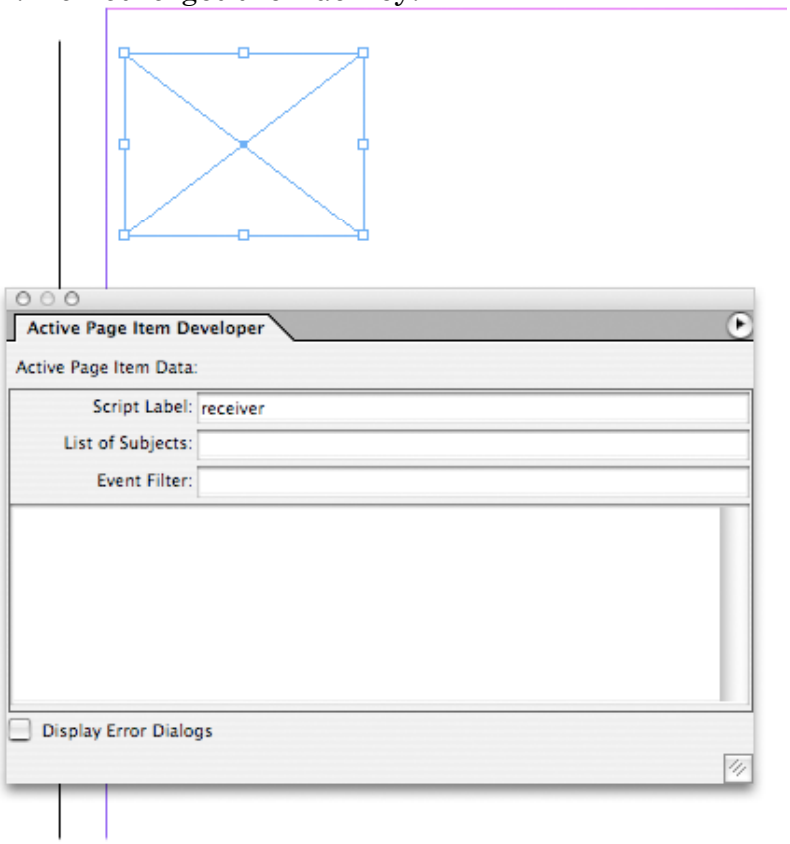
This expression !?* is a handy trick to match only empty strings.

2.9. Transmitting instead of observing

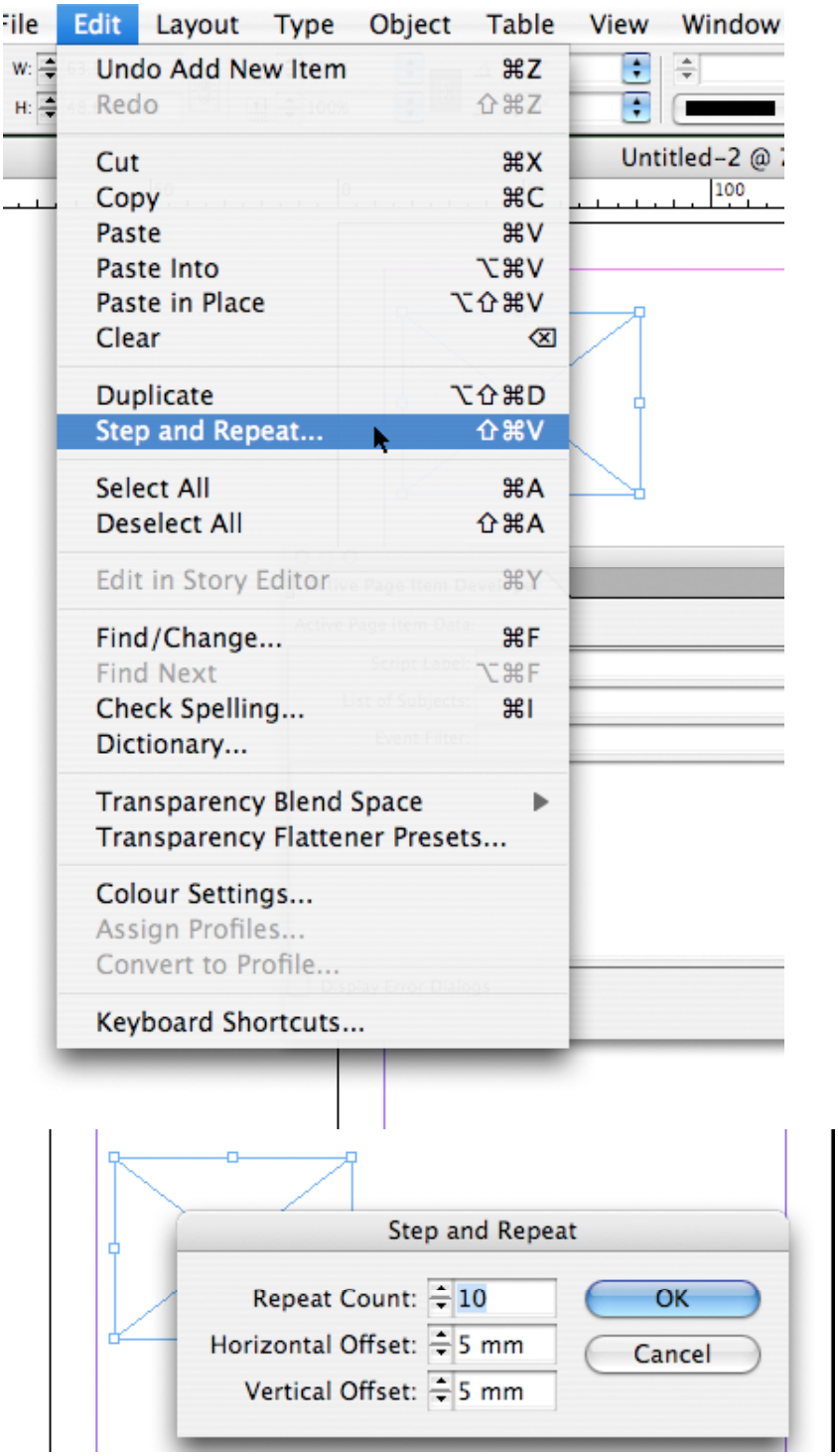
The technique demonstrated earlier is called 'observing'. There is an alternate technique which achieves similar results, and which can be a bit faster when many page items are involved.

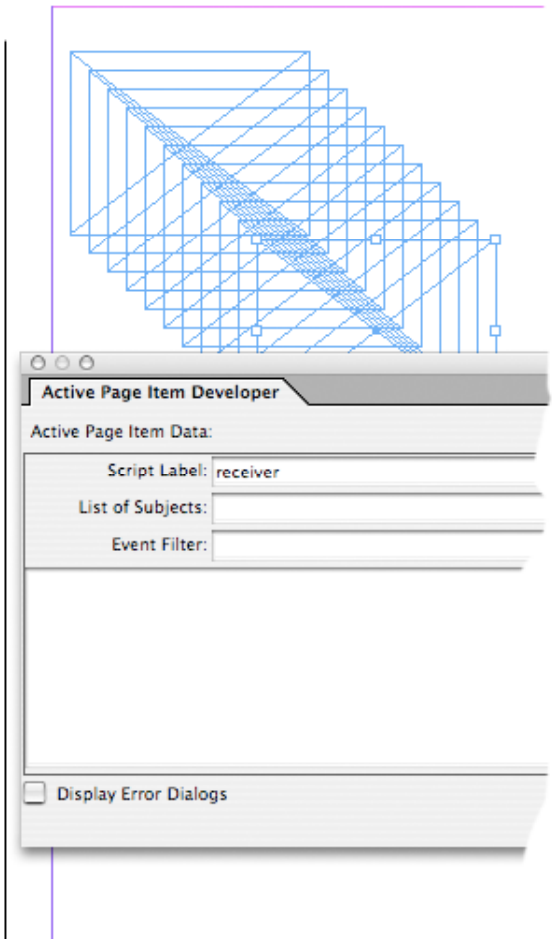
Create a new, empty document. Then create a single frame, and give it the script label

receiver. Do not forget the *Tab* key.



Use Step and Repeat to create a number of copies.

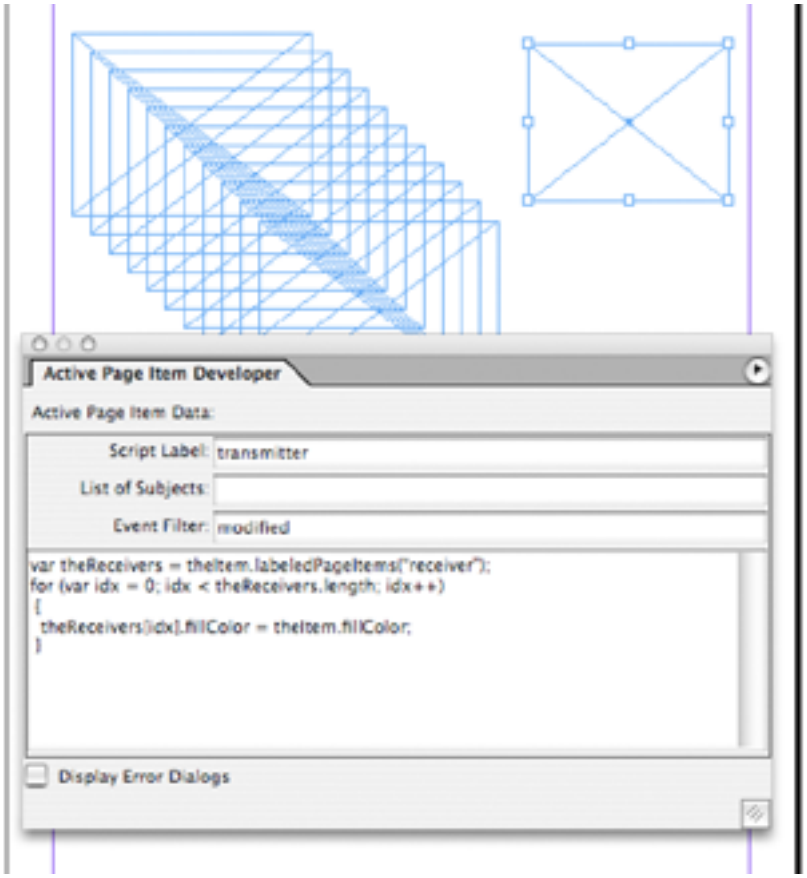

File
Edit
Layout
Type
Object
Table
View
Window
Undo Add New Item ⌘Z
Redo ⇧⌘Z
Cut ⌘X
Copy ⌘C
Paste ⌘V
Paste Into ⌥⌘V
Paste in Place ⌥⇧⌘V
Clear ⌫
Duplicate ⌥⇧⌘D
Step and Repeat... ⇧⌘V
Select All ⌘A
Deselect All ⇧⌘A
Edit in Story Editor ⌘Y
Find/Change... ⌘F
Find Next ⌥⌘F
Check Spelling... ⌘I
Dictionary...
Transparency Blend Space
Transparency Flattener Presets...
Colour Settings...
Assign Profiles...
Convert to Profile...
Keyboard Shortcuts...
Untitled-2 @
100
Step and Repeat
Repeat Count: 10
Horizontal Offset: 5 mm
Vertical Offset: 5 mm
OK
Cancel



Create a new frame, and give it the Event Filter *`modified`*. You can optionally enter *`transmitter`* in its *Script* Label field but that is not necessary for the script below to work – it just makes it easier to refer to the frame when talking about it.

Then give the *`transmitter`* frame the following JavaScript:

```
var theReceivers = theItem.labeledPageItems("receiver");
for (var idx = 0; idx < theReceivers.length; idx++)
 {
  theReceivers[idx].fillColor = theItem.fillColor;
 }
```



In the previous samples, the copying of the color was performed by the observer(s), and the example frame was a passive component as far as scripting went.

In this sample, the example frame (the transmitter) has become an active component instead, and the other frames are passive receivers.

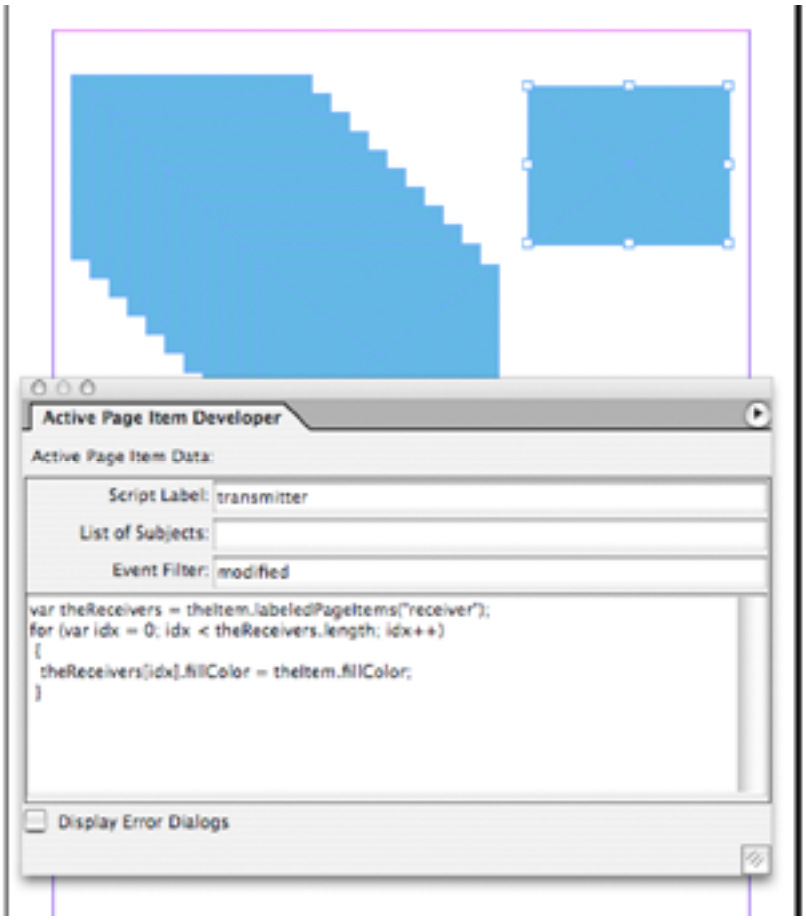
The script first asks *`theItem`* to provide an array of all frames carrying the script label *`receiver`*. This array will contain references to all the frames we created with Step and Repeat.

Then the loop copies the color of the *`transmitter`* frame into all the slaved receivers.

This approach is faster than the equivalent setup where all the *`receiver`* frames have a script

```
theItem.fillColor = theItem.eventSource.fillColor;
```

as well as a *List of Subjects* field that refers to the *`transmitter`*.



3. Editing and managing scripts

Because the palette is quite limited for editing, it is often easier to use an external editor to edit scripts.

We **strongly** recommend you use the APIDTemplate document as a starting point for developing scripted plug-ins – it automates a lot of the set-up necessary for an easy-to-maintain project. See paragraph 4.3 for more info.

3.1. Copy-Paste

One approach is to type the text into an external editor and then copy-paste the text into the palette.

3.2. External Scripts

Another approach is to use external files for the scripts, at least during development.

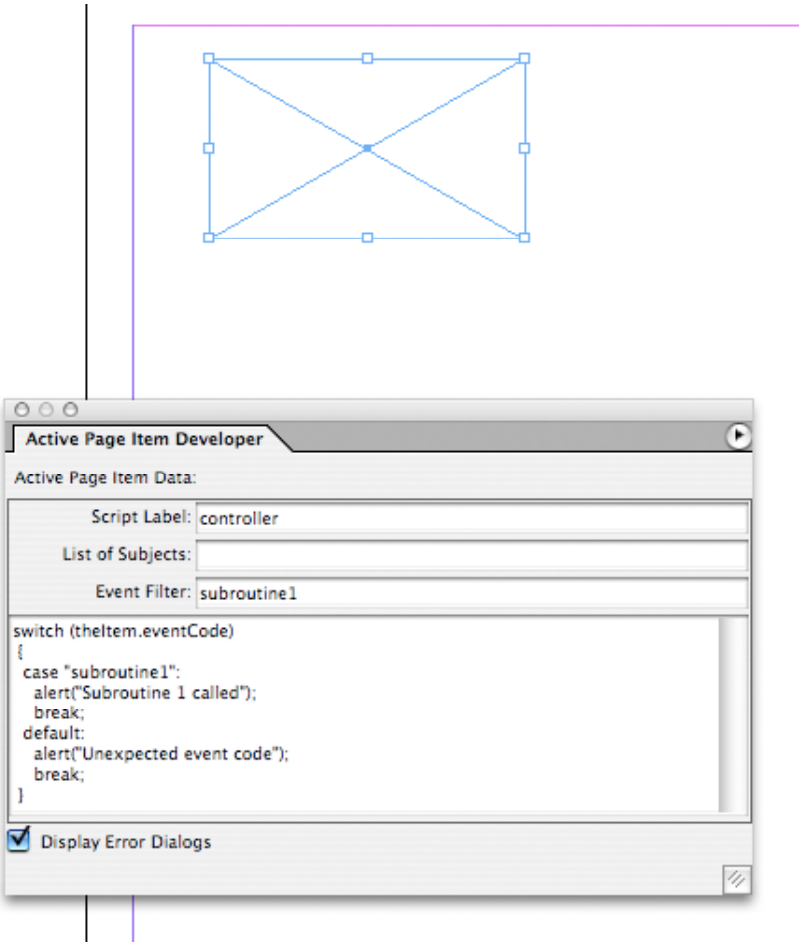
The following example calculates the location of the current document (assuming the document is not newly created and has been saved at least once), and then expects a script *testscript.js* to sit in the same folder.

```
var theDocument = GetDocument(theItem);
var theScriptFile = File(theDocument.filePath + "/testscript.js");
app.doScript(theScriptFile);

function GetDocument(thePageItem)
 {
  while (thePageItem != undefined && ! (thePageItem instanceof Document))
   {
    thePageItem = thePageItem.parent;
   }
  return thePageItem;
 }
```

3.3. One big script versus many small scripts

When developing a solution around *Active Page Item Developer* you can opt to use a host of small scripts and scatter them all over the document. This approach is usable, but as the complexity of your solution increases it will become harder and harder to maintain the collection of scripts.

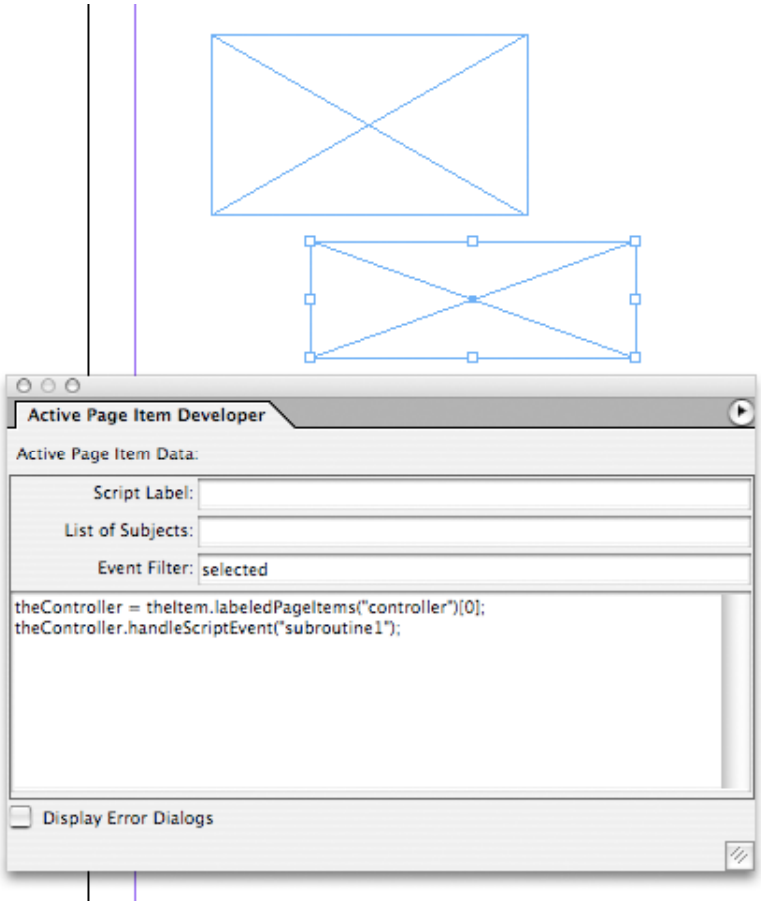
A second approach is to store (nearly) all scripting material in a single page item (often referred to as a *controller*). This controller typically observes a host of other page items, reacts to events they emit, and reacts to external events.



The controller lists in its event filter all possible event codes it needs to process, including any custom event codes that will be emitted by other page items through *handleScriptEvent* calls. In this example, we expect other page items to use the event code *subroutine1*, and the script attached to the controller is something similar to:

```
switch (theItem.eventCode)
 {
  case "subroutine1":
    alert("Subroutine 1 called");
    break;
  default:
    alert("Unexpected event code");
    break;
 }
```

Other page items can still have scripts attached to them, but these scripts are normally trivial and consist typically of just two lines – the first one to fetch a reference to the controller, the second a call to *`theController.handleScriptEvent`*, so further handling of the event is passed on to the controller, whose big script contains the necessary logic.



This example script first gets hold of the controller via the *`theItem.labeledPageItems`* method. Remember that this method returns an array – so we need to add an index *`[0]`* to actually access the single element of that array (assuming only a single page item in the whole document will carry the label *`controller`*). Then it calls *`theController.handleScriptEvent`* with an event code *`subroutine1`* to activate the relevant code in the controller script.

```
theController = theItem.labeledPageItems("controller")[0];
theController.handleScriptEvent("subroutine1");
```

4. Examples related to the additional PageItem properties

The Active Page Items plug-in adds a number of properties and events (a.k.a. methods) to page item objects.

Some of the added properties are visible through the *Active Page Item Developer* palette: the *Script Label* field shows the *`label`* property, the *List of Subjects* field shows the *`subjectScriptTagList`* property, the *Event Filter* field shows the *`eventFilter`* property, and the large text edit field shows the *`javaScript`* property.

The *Display Error Dialogs* checkbox shows the *`displayErrorDialogs`* property, and in InDesign CS2 or above, the *Use Debugger* checkbox shows the *`useDebugger`* property.

The other properties (like *`dataStore`*, *`tempDataStore`*,...) are only accessible from a JavaScript.

In this section, we'll build a few example scripts that demonstrate various techniques for using the *APID ToolAssistant* plug-in.

4.1. Adding a context menu to a page item

In this example, we'll add a context menu to a page item, which allows us to rotate the page element in increments of 45 degrees.

We'll start with a simple trial set-up first, and gradually build it up to something useful.

Adding a context menu to a single page item

Create a new page item, and use the *Active Page Item Developer* palette to attach the information shown below to it.

Set the script to

```
if (theItem.eventCode == "loadContextMenu")
{
  var theMenu = new Array();
  theMenu.push(["1/8 turn","rotateCW"]);
  theMenu.push(["-1/8 turn","rotateCCW"]);
  theItem.contextMenu = theMenu;
}
else if (theItem.eventCode == "rotateCW")
{
  theItem.absoluteRotationAngle += 45;
}
else if (theItem.eventCode == "rotateCCW")
{
  theItem.absoluteRotationAngle -= 45;
}
```

This script processes three possible event codes: *`loadContextMenu`* which will be received when the user right-clicks or control-clicks on a selection of page items. It also processes two new, non-standard event codes *`rotateCW`* and *`rotateCCW`* that are to be sent when the user selects one of the context menus.

When the page item receives a context-click, it will create an array *`theMenu`,* which contains two entries, one for each context menu item.

Each of the entries is itself an array with two strings – the first string is the menu item text, the second string is the event code that will be emitted when the context menu is selected.

In this example, the build-up of the menu is spread over a few lines to improve readability, but you might just as well have written the equivalent code:

```
…
if (theItem.eventCode == "loadContextMenu")
{
 theItem.contextMenu = [["1/8 turn","rotateCW"],["-1/8 turn","rotateCCW"]];
}
…
```

When one of the menu items is selected, an event code *`rotateCW`* or *`rotateCCW`* will be sent,

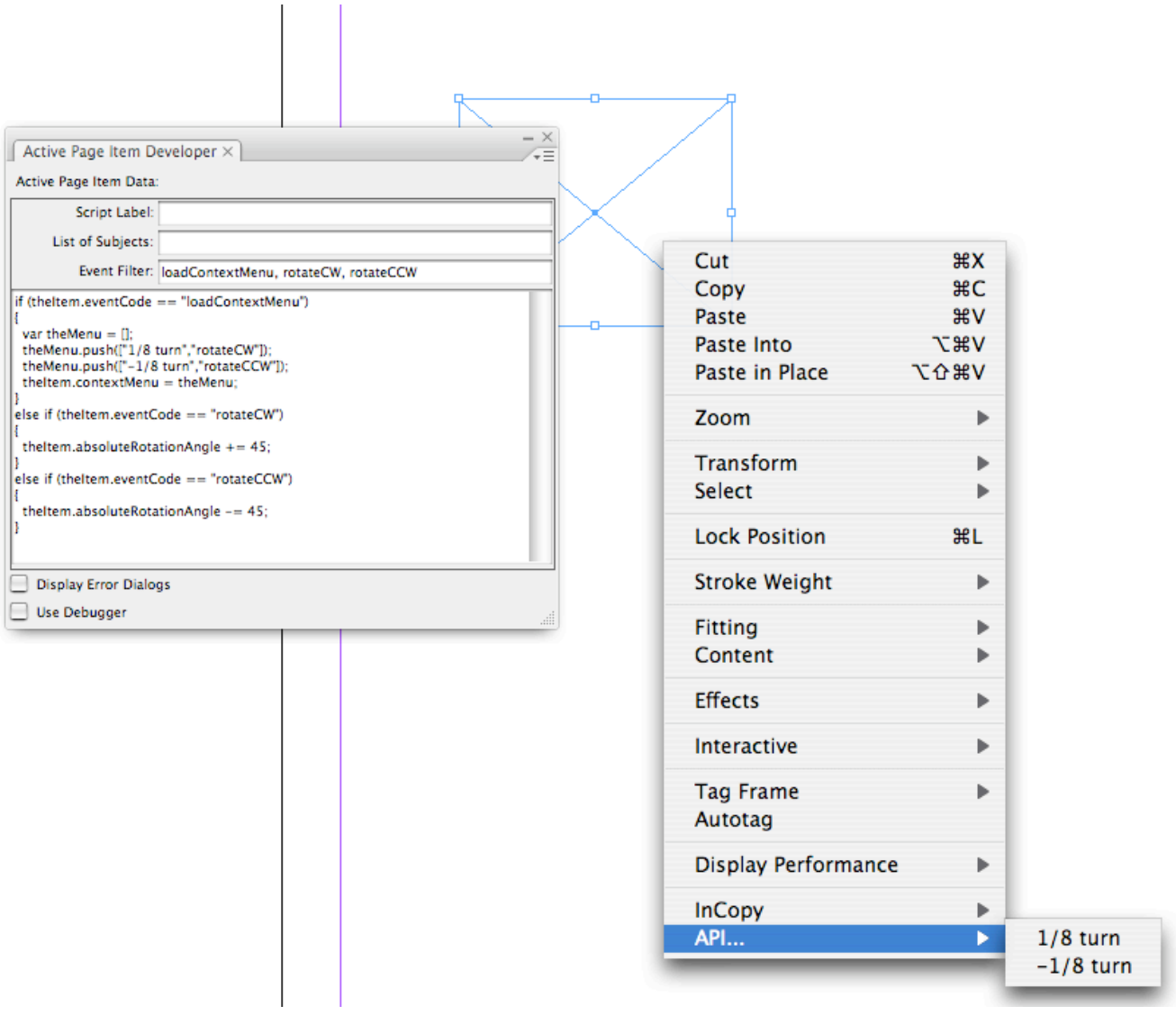
and the above script will increase or decrease the rotation of the page item by 45 degrees.

Set the event filter to
`loadContextMenu, rotateCW, rotateCCW`
(don't forget to use the *Tab* key).

Neglecting to do this would make the script seem not to work.

Select and context-click the page item to fire off a `loadContextMenu` event and execute the script.

Then bring up the context menu (control-click on Macintosh, right-click on Windows) – an *API…* menu item should be available, with two sub-menu items *1/8 turn* and *-1/8 turn*.



Adding a context menu to multiple page items

The above example works fine, but it only handles a single page item. Instead, we'd like to add the same context menu to all page items.

To achieve that goal, we need to make use of a so-called *controller*.

A controller is a page item that observes the document and provides functionality for the whole document. The advantage of using a controller is that we don't need to manually add scripts and filters to individual page items.

Delete the page item from the previous example. Create a new page item somewhere on the pasteboard, and give it the script label *rotationController*. This script label is not really important, but we add it as a reminder of the function of the page item we're about to build.

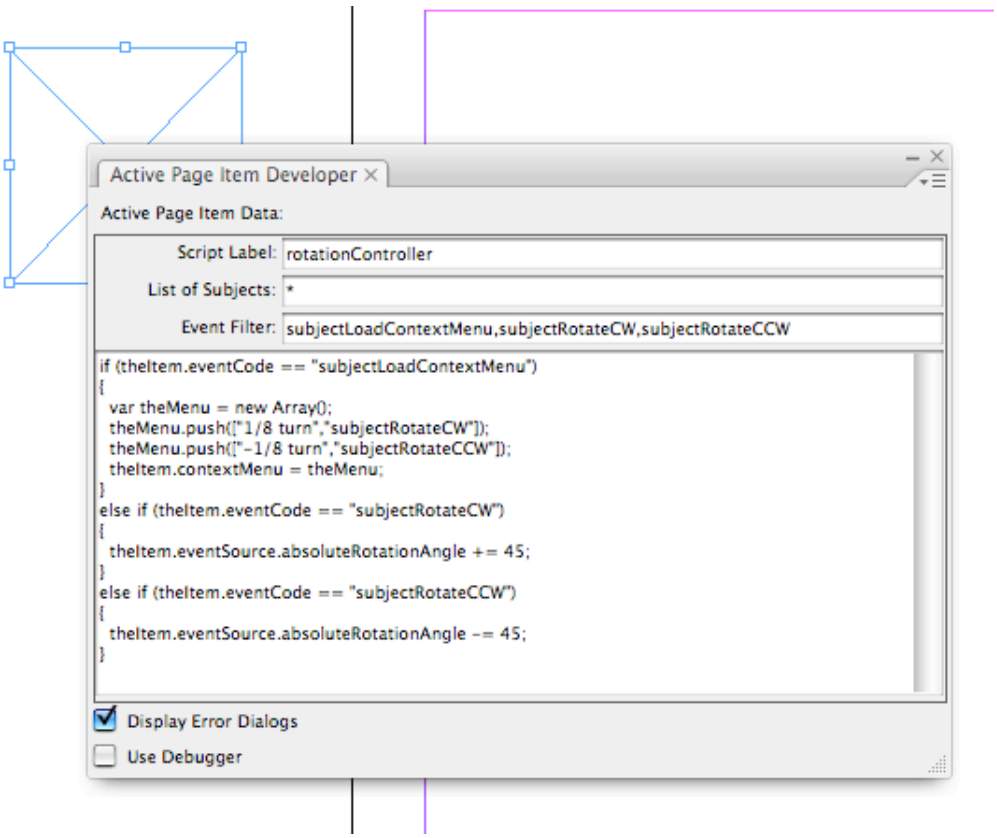
We put the controller in the pasteboard area because it will most probably not be meant to appear on the printed result.

Use the *Active Page Item Developer* palette to add the following script to the controller:

```
if (theItem.eventCode == "subjectLoadContextMenu")
{
  var theMenu = new Array();
  theMenu.push(["1/8 turn","subjectRotateCW"]);
  theMenu.push(["-1/8 turn","subjectRotateCCW"]);
  theItem.contextMenu = theMenu;
}
else if (theItem.eventCode == "subjectRotateCW")
{
  theItem.eventSource.absoluteRotationAngle += 45;
}
else if (theItem.eventCode == "subjectRotateCCW")
{
  theItem.eventSource.absoluteRotationAngle -= 45;
}
```

Set the event filter to
subjectLoadContextMenu, subjectRotateCW, subjectRotateCCW

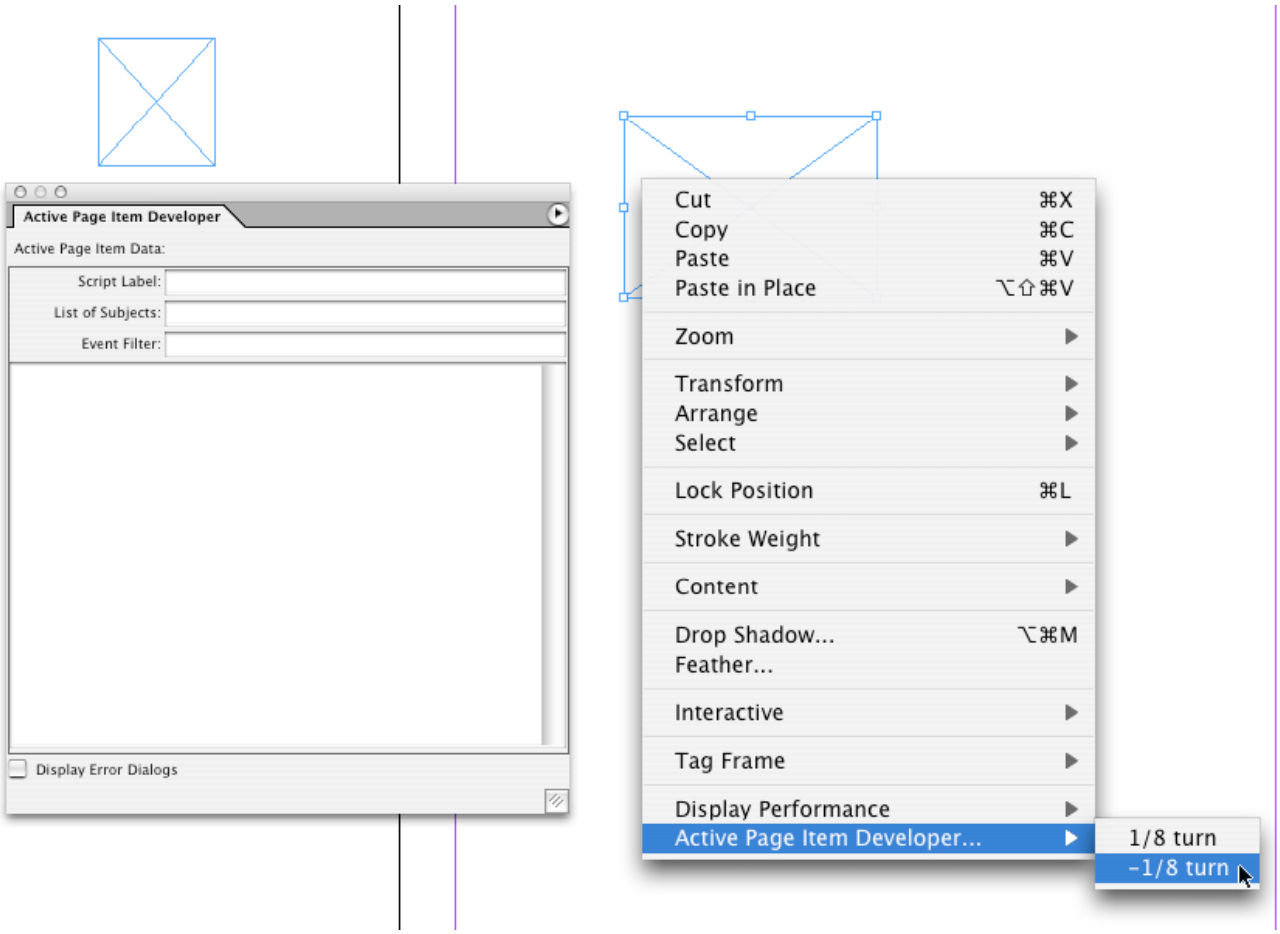
The main differences with the previous example are that the new event codes now all start with the prefix *subject…* Furthermore, the rotation now refers to *theItem.eventSource.absoluteRotationAngle* instead of *theItem.absoluteRotationAngle.*



Because the context menu event codes all start with a `subject…` prefix, they are automatically associated with all of the page items observed by the `rotationController`.

The *List of Subjects* of the `rotationController` is set to `*` (an asterisk)- meaning that our `rotationController` observes all page items in the document, even those ones without a script label.

The end result is that any page item will now have two context menu entries. Selecting one of the menu items on any page item will invoke the script of the `rotationController`, with `theItem.eventSource` set to the page item whose context menu was selected.



4.2. Persistent data

'Persistent data' means: data that remains accessible between multiple calls to a script and/or remains accessible after save-close and re-open of the document.

One issue with ExtendScript is that it is not immediately obvious how to store persistent data.

A standard, InDesign-provided way of keeping some persistent data associated with an object is to use the `extractLabel/insertLabel` methods.

A disadvantage of these two methods is that they only allow you to store and retrieve string data.

To alleviate some of the limitations, the *APID ToolAssistant* plug-in implements some alternate approaches that accomplish similar results.

First of all, the *APID ToolAssistant* plug-in adds two new properties to page items: `dataStore` and `tempDataStore`.

These are single-value containers that store simple data types like strings, integers, booleans, or arrays of these simple data types (in the current version, they cannot store objects composed of these simple data types, only arrays).

The difference between `dataStore` and `tempDataStore` is that `dataStore` is persistent when the document is save-closed and re-opened, whereas `tempDataStore` is not saved, and is

reset each time a document is re-opened.

Alternatively, there are also the two methods *getDataStore/setDataStore* which also use a string key to store and retrieve simple data types. An empty key ("") is associated with the same data storage locations as are accessed through *dataStore* and *tempDataStore.*

The following code:

```
var x = theItem.dataStore;
var y = theItem.tempDataStore;
theItem.dataStore = "xyz";
theItem.tempDataStore = 123;
```

is equivalent to:

```
var x = theItem.getDataStore("");
var y = theItem.getDataStore("",true);
theItem.setDataStore ("","xyz");
theItem.setDataStore("", 123, true);
```

and also to:

```
var x = theItem.getDataStore("",false);
var y = theItem.getDataStore("",true);
theItem.setDataStore ("","xyz",false);
theItem.setDataStore("", 123, true);
```

Automatically positioning elements

The following example uses the persistent data to distinguish between different reasons for the *modified* event code. At the same time, it also shows the use of the *labeledPageItems* method.

A *modified* event code can be the result of a user action, or it can be the result of a scripted action.

Sometimes it is important to distinguish between the two – scripts can get into endless loops if you are not careful.

For example, you could write a script that reacts to a *modified* event by repositioning the page element that caused the event. This repositioning causes another *modified* event to be fired, causing another repositioning, etc… ad infinitum.

By carefully using the persistent data stores such loops can be avoided.

Create a new document and create a new page item in the document. Give it the script label *autoLiner*, and the event filter *created, modified*. Attach the following script to the page item:

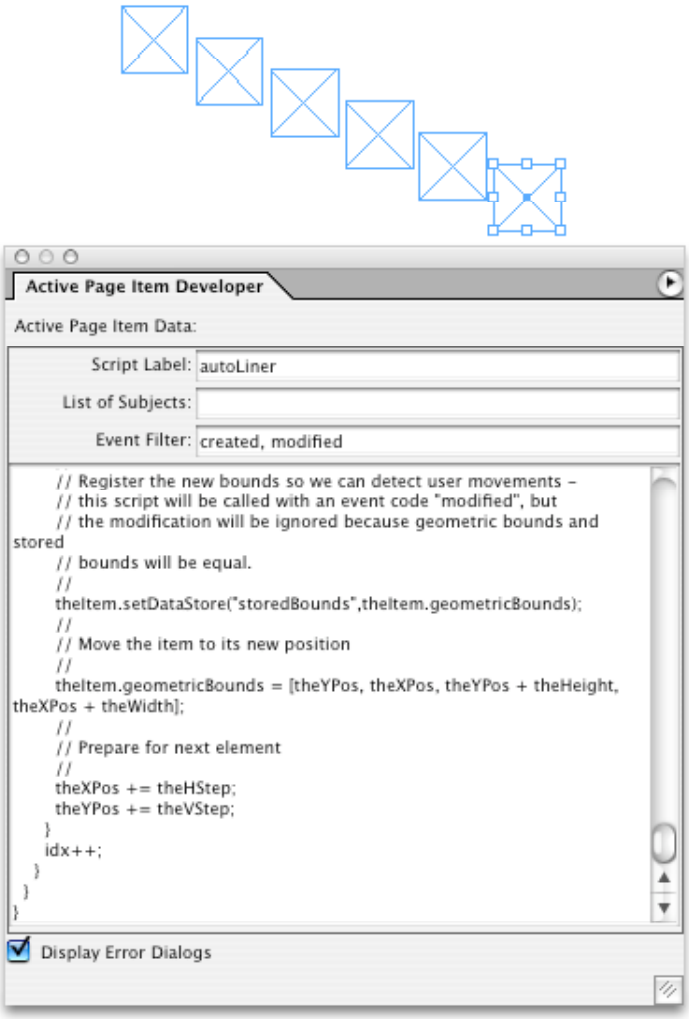
```
switch (theItem.eventCode)
{
  case "created":
    //
    // This page item was just copy-pasted - move it where it should be.
    //
    RepositionElements(theItem,true);
    break;
  default:
    //
    // This page item was modified. Check its storedBounds to find out
    // whether it was
    // - not moved
    // - moved by the user
    // - moved by the script
    // Only when it was moved by the user do we reposition elements
    //
    if (IsMovedByUser(theItem))
    {
      RepositionElements(theItem,false);
    }
    break;
}

function IsMovedByUser(theItem)
{
  //
  // Compare the stored geometric bounds with the real
  // geometric bounds. If they are different, the element
  // has been moved by the user. If the stored bounds
  // are undefined, then we've never been called before on this
  // element, so we also consider this element 'moved'.
  //
  var theStoredBounds = theItem.getDataStore("storedBounds");
  var isMoved = false;
  if (theStoredBounds == undefined)
  {
    isMoved = true;
  }
  else
  {
    isMoved = false;
    var idx = 0;
    //
    // Loop through top, left, bottom, right until difference found
    // or until all four coordinates checked.
    //
    while (idx < 4 && ! isMoved)
    {
      isMoved = theItem.geometricBounds[idx] != theStoredBounds[idx];
      idx++;
    }
  }

  return isMoved;
}
```

```
function RepositionElements(theItem,isNewlyCreated)
{
  //
  // Fetch an array with all elements that share the same label
  // as the current item (theItem will also be in this array).
  //
  var relatedElements = theItem.labeledPageItems(theItem.label);
  //
  // We need at least 3 elements before we can do anything useful
  //
  if (relatedElements.length >= 3)
  {
    //
    // Find the two leftmost, topmost items that are not the newly created
    // item itself
    //
    if (isNewlyCreated)
    {
      var topLeftMost = Find2TopLeftMostElements(relatedElements,theItem);
    }
    else
    {
      var topLeftMost = Find2TopLeftMostElements(relatedElements);
    }
    //
    // Iterate through all page items with the same label and align them in a
    // single file with the same space between them.
    //
    RedistributeElements(topLeftMost[0],topLeftMost[1],relatedElements);
  }
}
```

```
//
// For 2nd parameter pass "undefined" or omit it when called without a newly
// created item
//
function Find2TopLeftMostElements(theElementArray,theNewlyCreatedItem)
{
  var theFirst = undefined;
  var theSecond = undefined;
  var idx = 0;
  while (idx < theElementArray.length)
  {
    //
    // Ignore the newly created item (if any)
    //
    theItem = theElementArray[idx];
    if (theItem != theNewlyCreatedItem)
    {
      //
      // Keep track of the two leftmost, topmost items
      //
      if (theFirst == undefined)
      {
        theFirst = theItem;
      }
      else if
      (
        // Is the new item more to the left than the first element
        // or is the new item just as much to the left but above the first?
        theFirst.geometricBounds[1] > theItem.geometricBounds[1]
       ||
         (
           theFirst.geometricBounds[1] == theItem.geometricBounds[1]
          &&
           theFirst.geometricBounds[0] >= theItem.geometricBounds[0]
         )
      )
      {
        //
        // The previous "first" can now be our new second - we
        // have a better "first".
        //
        theSecond = theFirst;
        theFirst = theItem;
      }
      else if (theSecond == undefined)
      {
        theSecond = theItem;
      }
      else if
      (
        // Is the new item more to the left than the second element
        // or is the new item just as much to the left but above the second?
        theSecond.geometricBounds[1] > theItem.geometricBounds[1]
       ||
         (
           theSecond.geometricBounds[1] == theItem.geometricBounds[1]
          &&
           theSecond.geometricBounds[0] >= theItem.geometricBounds[0]
         )
      )
      {
        theSecond = theItem;
      }
    }
    idx++;
  }

 //
 // Return 2-element array
 //
 return [ theFirst, theSecond];
}
```

```
function RedistributeElements(theFirst,theSecond,theElementArray)
{
  //
  // Calculate how much distance to leave between consecutive
  // elements by comparing the first two
  //
  var theHStep = theSecond.geometricBounds[1] - theFirst.geometricBounds[1];
  var theVStep = theSecond.geometricBounds[0] - theFirst.geometricBounds[0];

  var theXPos = theFirst.geometricBounds[1];
  var theYPos = theFirst.geometricBounds[0];
  var idx = 0;
  while (idx < theElementArray.length)
  {
    var theItem = theElementArray[idx];
    //
    // Calculate the height for this element (elements might have
    // different heights and/or widths, so we recalculate it for
    // each element)
    //
    var theHeight = theItem.geometricBounds[2] - theItem.geometricBounds[0];
    var theWidth = theItem.geometricBounds[3] - theItem.geometricBounds[1];
    //
    // Check whether we'll really move the element (allow for round-off
    // errors up to 0.001)
    //
    if
    (
      Math.abs(theItem.geometricBounds[0] - theYPos) > 0.001
     ||
      Math.abs(theItem.geometricBounds[1] - theXPos) > 0.001
    )
    {
      var theRight = theXPos + theWidth;
      var theBottom = theYPos + theHeight;
      var theNewBounds = [theYPos, theXPos, theBottom, theRight];
      //
      // Register the new bounds so we can detect user movements -
      // this script will be called with an event code "modified", but
      // the modification will be ignored because geometric bounds and stored
      // bounds will be equal.
      //
      theItem.setDataStore("storedBounds",theNewBounds);
      //
      // Move the item to its new position
      //
      theItem.geometricBounds = theNewBounds;
    }
    //
    // Prepare for next element
    //
    theXPos += theHStep;
    theYPos += theVStep;
    idx++;
  }
}
```

Now use copy-paste to create a copy of the page item, and move this second copy somewhere close to the first. Then paste a third copy. The script will kick into action and move the newly created element into the neighborhood of the first two. Paste a few more times.



Use the mouse to drag the first or second element to another position. As soon as you let go of the mouse button, the remainder of the elements will reposition itself.

The script handles two event codes: *`created`* and *`modified`*. When the *`created`* event code is received, we know that the element must be considered 'moved' – our script has not 'seen' this element before, so we immediately call the routine that repositions the page elements.

If the event code is *`modified`*, we first check whether the element is moved or not, and whether it was moved by the user.

To differentiate between user movement and script-driven movement, we use a persistent data entry called `storedBounds`, which contains a copy of the last known `geometricBounds` of the page item - `geometricBounds` is an array of 4 numerical values – top, left, bottom, right.

When the script moves elements around, it takes care to make sure `storedBounds` is equal to the new `geometricBounds` prior to the move – so if the script is activated because of a script-driven movement, the script will know to ignore the movement.

On the other hand, when the user moves an element, the `storedBounds` will be different from the `geometricBounds` and the script will reposition the elements as needed.

The function `IsMovedByUser()` implements this 'user-movement' detection.

The function `RepositionElements()` uses `Find2TopLeftMostElements()` to find the two top-most left-most elements, and passes them to `RedistributeElements()` to use as examples for calculating by what distances elements should be spaced horizontally and vertically.

`RepositionElements()` also is an example of how the `labeledPageItems` method can be used to get an array of all elements that carry a particular label.

`Find2TopLeftMostElements()` is called with 1 or 2 parameters – in the case of the `created` event code, we pass the newly created item in the second parameter. In the case of the `modified` event code, we pass no second parameter, leaving the `theNewlyCreatedItem` parameter undefined.

Play around with sets of three, four, five copies and try to understand the interactions – for example, grab the third element, drag it to the left of the first two. Think about what happened. It is not obvious to see, but some interesting reshuffling has taken place…

Remark: keep in mind that, as we're copy-pasting elements, each element has a *copy* of the above script attached to it. If you want to modify the script, you should do that 'outside' of InDesign, in a text editor.

To use the modified script, you should first delete all but one of the elements you have been playing with and then paste the updated script on top of the script of the single remaining element. Failing to do this would leave you with multiple versions of the script floating around in various elements, which would cause strange results.

Using a controller

A better approach would be to move the script into a controller which resides somewhere on the pasteboard.

Delete all but one of the intelligent page items you created while playing with the previous example.

Move the single remaining page item to the pasteboard area.

We'll now modify this page item and its script to become a controller.

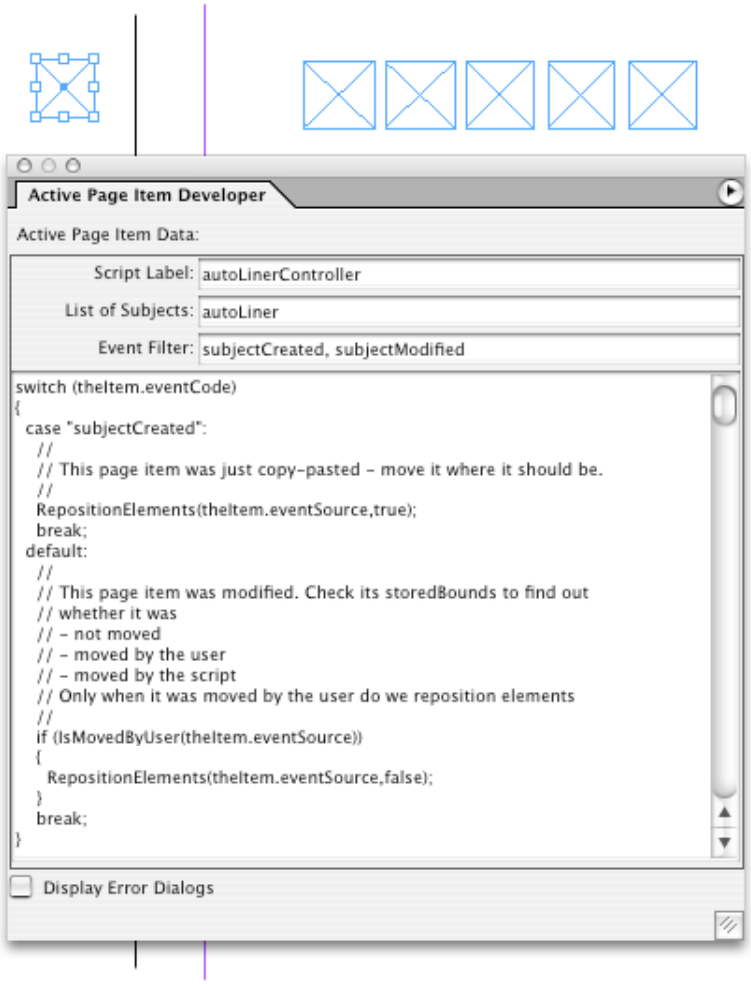
We relabel the page item from `autoLiner` to `autoLinerController.`

We also change the *List of Subjects* to `autoLiner`.

Then we change the *Event Filter* to `subjectCreated, subjectModified`

We modify the start of the script so it reacts to these event codes instead of to *created* and *modified*.

Furthermore, the script now passes *theItem.eventSource* to the functions instead of *theItem* itself – *theItem* refers to the controller, *theItem.eventSource* refers to the newly created or modified page element.



The modified part of the script now looks like this:

```
switch (theItem.eventCode)
{
  case "subjectCreated":
    //
    // This page item was just copy-pasted - move it where it should be.
    //
    RepositionElements(theItem.eventSource,true);
    break;
  default:
    //
    // This page item was modified. Check its storedBounds to find out
    // whether it was
    // - not moved
    // - moved by the user
    // - moved by the script
    // Only when it was moved by the user do we reposition elements
    //
    if (IsMovedByUser(theItem.eventSource))
    {
      RepositionElements(theItem.eventSource,false);
    }
    break;
}

… Remainder of script is identical …
```

All occurrences of *theItem* have been replaced with *theItem.eventSource*, and the *created* event code has been replaced with *subjectCreated.*

To test the script you should create a new page item, and change its label to *autoLiner*. That makes it a subject of our controller.

Then copy-paste the newly created page item, position the duplicate close to the original, and paste again – the second duplicate will automatically 'line up' with the first two.

The example works pretty much the same as the previous example – but with this set-up we only have a single copy of the JavaScript residing in the controller instead of having multiple copies scattered all around the document.

Controlling multiple groups of 'auto-liners'

Suppose you wanted to create a similar group of 'auto liners', say, on a second page – in that case, you could use a different label, say *autoLiner2* so there is no conflict with our first group which uses the *autoLiner* label.

Then you could simply add the new label to the *List of Subjects* for the controller – the same controller is able to control many groups.

That approach is not perfect yet – each time we want to add a new group – *autoLiner3*, *autoLiner4*,… we have to adjust the *List of Subjects* of the controller.

This problem is easy to solve – we simply set the *List of Subjects* to the wildcard expression *autoLiner** instead of just *autoLiner*.

Now this single controller can observe and manage multiple groups of page items –

simply create a page item with a label that starts with the word `autoLiner…` and the controller will start managing the positions of groups of copy-pasted versions of this page item.

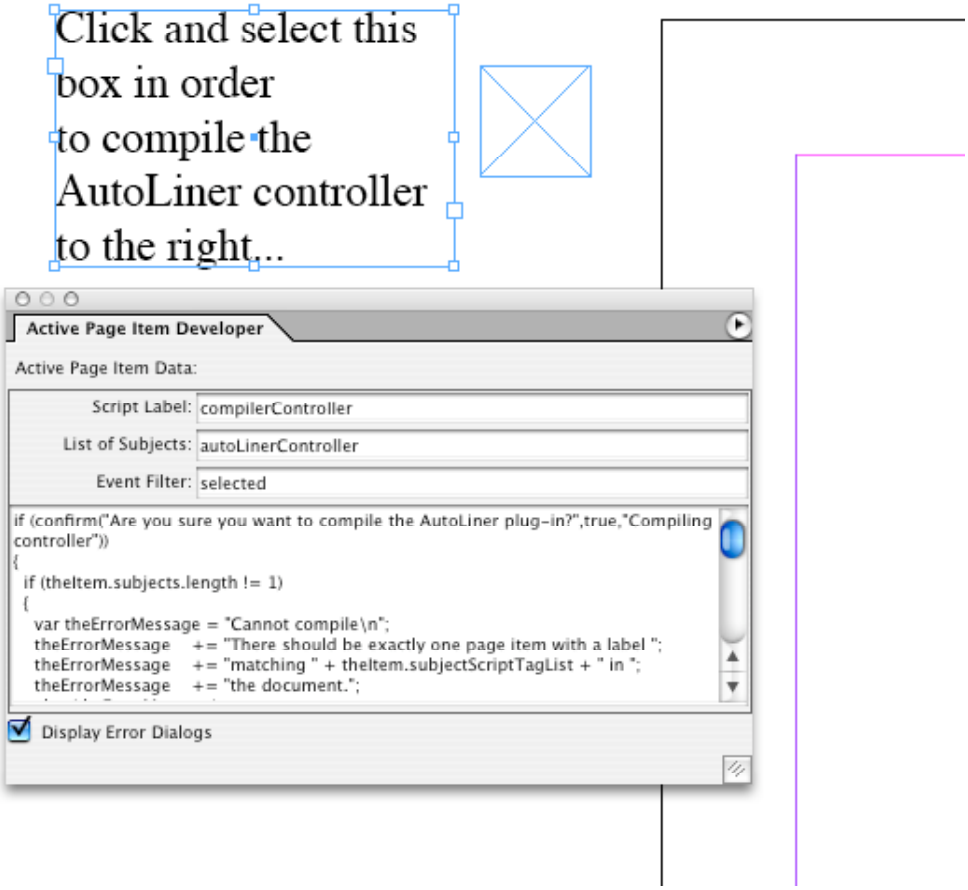
Converting the controller to a scripted plug-in

An additional benefit of the controller-based approach is that it becomes easy to convert the controller to a plug-in.

To do the conversion, we will use a second controller – create a new page item on the pasteboard and label it `compilerController`

Set the *List of Subjects* to `autoLinerController.`

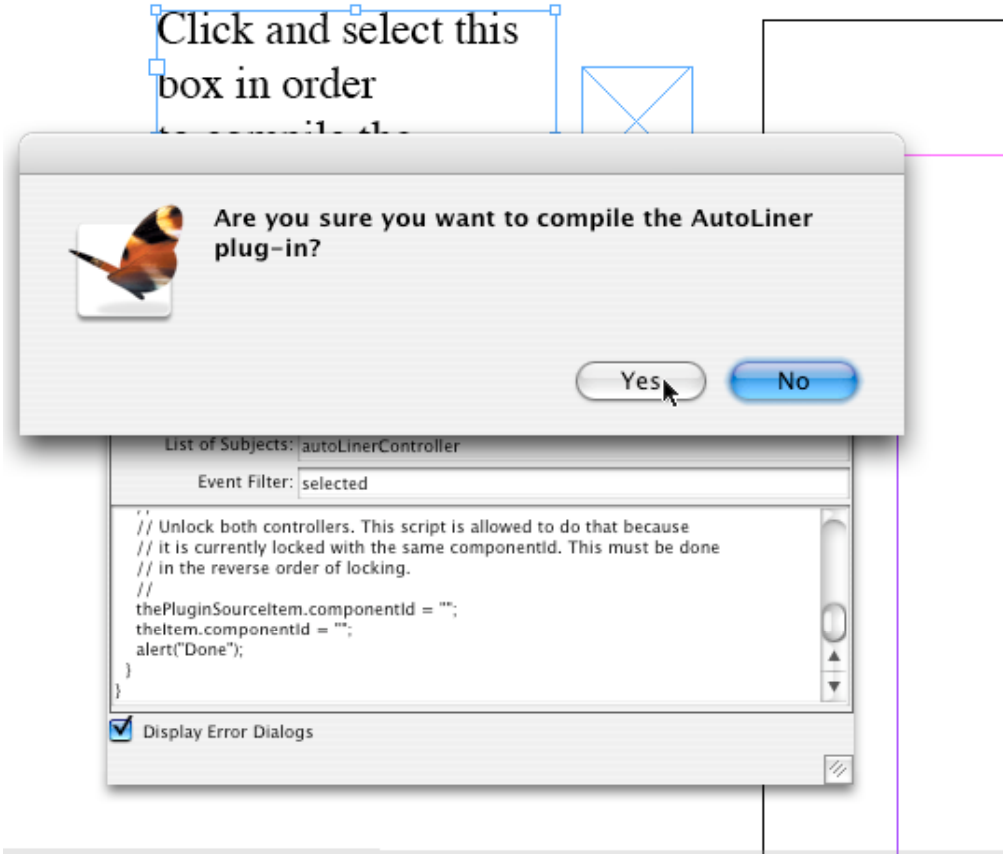
Set the *Event Filter* to `selected.`



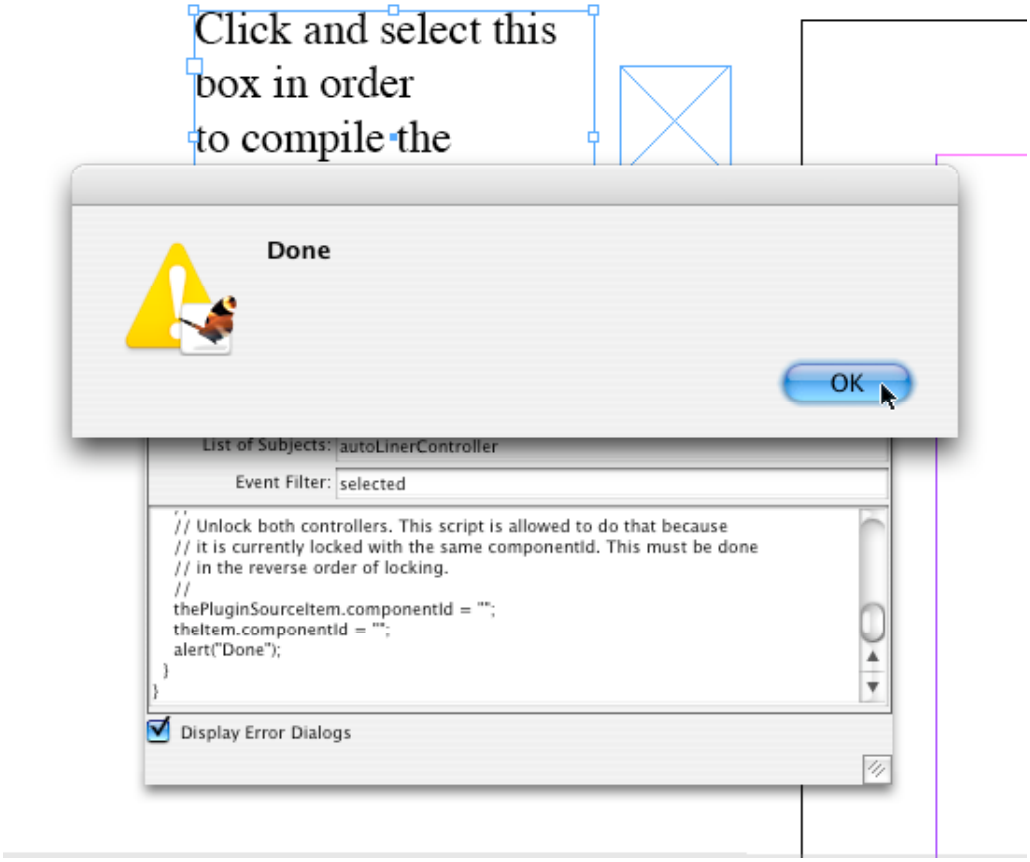
Attach the following script to the *compilerController:*

```
if (confirm("Are you sure you want to compile the AutoLiner plug-in?",true,"Compiling controller"))
{
  if (theItem.subjects.length != 1)
  {
    var theErrorMessage = "Cannot compile\n";
    theErrorMessage    += "There should be exactly one page item with a label ";
    theErrorMessage    += "matching " + theItem.subjectScriptTagList + " in ";
    theErrorMessage    += "the document.";
    alert(theErrorMessage);
  }
  else
  {
    //
    // Our single subject is the autoLinerController
    //
    var thePluginSourceItem = theItem.subjects[0];
    //
    // This componentID embeds the following attributes:
    //
    // name of plugin:      AutoLiner
    // password:            ThePassword123
    // copyright:           (c) 2006 Rorohiko Ltd.
    // licensing:           free
    // min. plugin version: 1.0.6
    // end-date:            -1,-1,-1 (no timeout)
    // num actual days:     0 (not set)
    // num demo days:       0 (not set)
    //
    var theComponentName = "AutoLiner;ThePassword123;(c) 2006 Rorohiko Ltd.;Free";
    var theComponentId = theComponentName + ",1.0.6,-1,-1,-1,0,0";

    //
    // "Lock" both controllers. Because this currently executing script
    // still needs to be able to access the other controller we need to
    // lock it too, with the same componentId.
    //
    // If we did not lock theItem too, it would loose access to
    // thePluginSourceItem as soon as we locked that. By setting the
    // componentId in the proper order we make sure theItem can continue
    // to access thePluginSourceItem's APID ToolAssistant data.
    //
    theItem.componentId = theComponentId;             // Lock compilerController
    thePluginSourceItem.componentId = theComponentId; // Lock autoLinerController

    //
    // Save the locked autoLinerController as a plug-in
    //
    thePluginSourceItem.saveToPlugin("autoLiner");

    //
    // Unlock both controllers. This script is allowed to do that because
    // it is currently locked with the same componentId. This must be done
    // in the reverse order of locking.
    //
    thePluginSourceItem.componentId = "";
    theItem.componentId = "";
    alert("Done");
  }
}
```

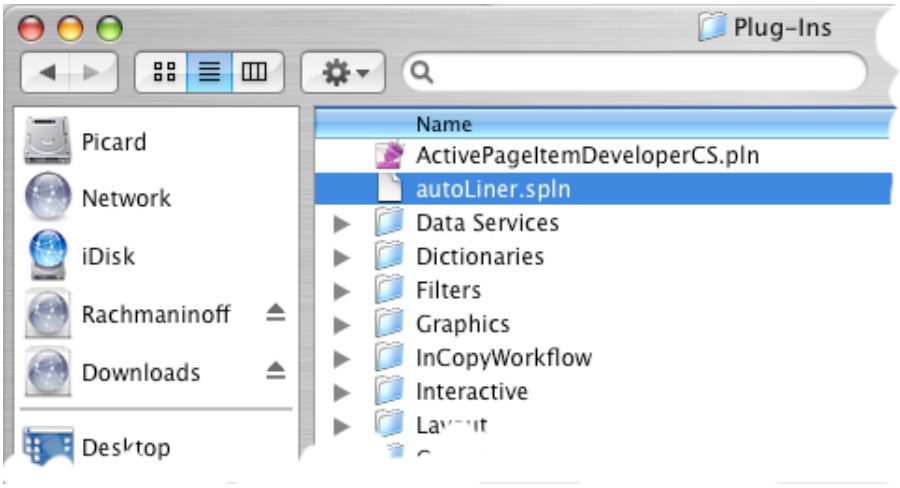
Deselect and then select the `compilerController`. A dialog should present itself. Click *Yes*. This dialog is useful for those cases where you simply want to re-select this page item *without* causing a compilation.



A second dialog shows. Click *OK*.



Go have a look in the InDesign *Plug-Ins* folder – there should be a new file called *autoLiner.spln* – this is your scripted plug-in file.



Now we can go through the same motions as before, but this time without having to attach any scripts to page items.

Create a new document, and create a new page item. Set the label of this page item to `autoLinerTest` (or anything else that starts with `autoLiner…`)

Copy-paste the new page item. Put the copy close to its original. Paste again. The pasted

item should align itself with the first two.

Note that this last document has no scripting attached to any page item – all we did was 'mark' some page items as 'special' by giving them a script label that started with `autoLiner…` The rest of the functionality is contained in the scripted plugin *autoLiner.spln* which resides in your InDesign *Plug-Ins* folder.

4.3. Using the APIDTemplate

The *APIDTemplate* document automates a lot of the tedious work of properly setting up a new APID project.

In this example, we'll use *APIDTemplate* to create a new project that adds a menu item for to the menu bar that allows us to apply a 'yellow marker' to a text selection.

Scripted plug-ins are private to documents

To understand the following example better, we need to first stress an important detail.

Each time a document is loaded, the APID ToolAssistant will create a new and **separate** instance of the scripted plug-in, specifically for this document.

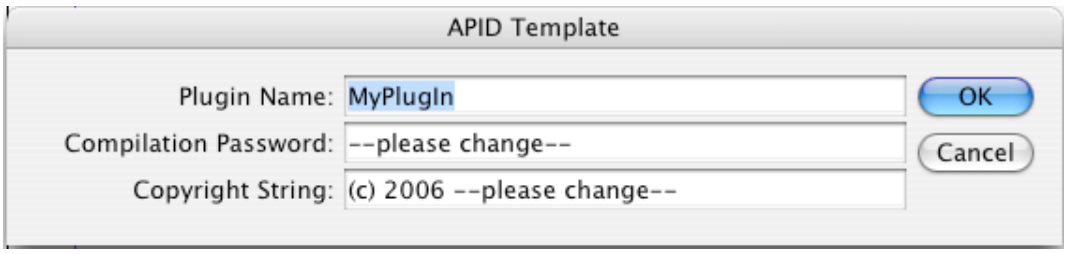
In other words: each open document has its own private copy of any scripted plug-ins that are residing in the plug-ins folder.

And if no documents are open, no instances of the scripted plug-in are available – that is why the menu items created by scripted plug-ins are disabled when no document is open.

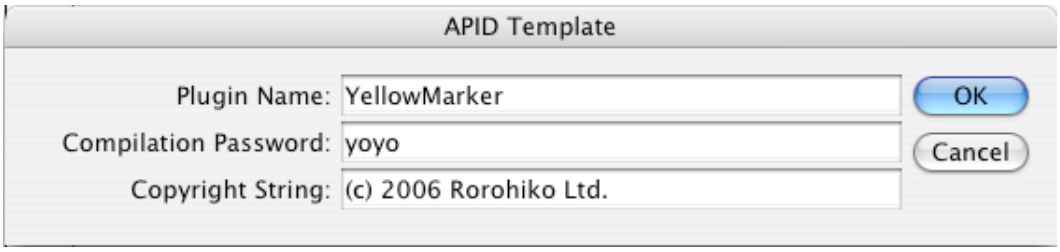
Creating the YellowMarker project

First, launch InDesign and open the document *APIDTemplate.indd*. In this example, we'll use InDesign on Mac OS X for generating the screenshots – it's all very similar on Windows and on CS, CS2, CS3 or CS4.

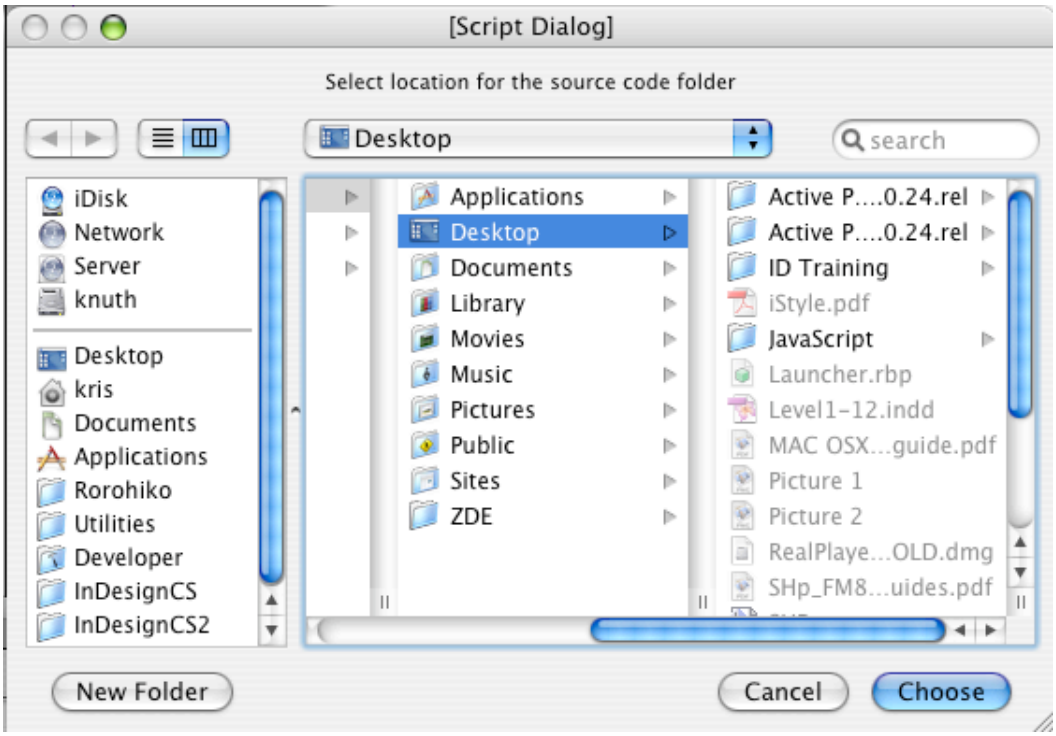
If all is well, a dialog should show up:



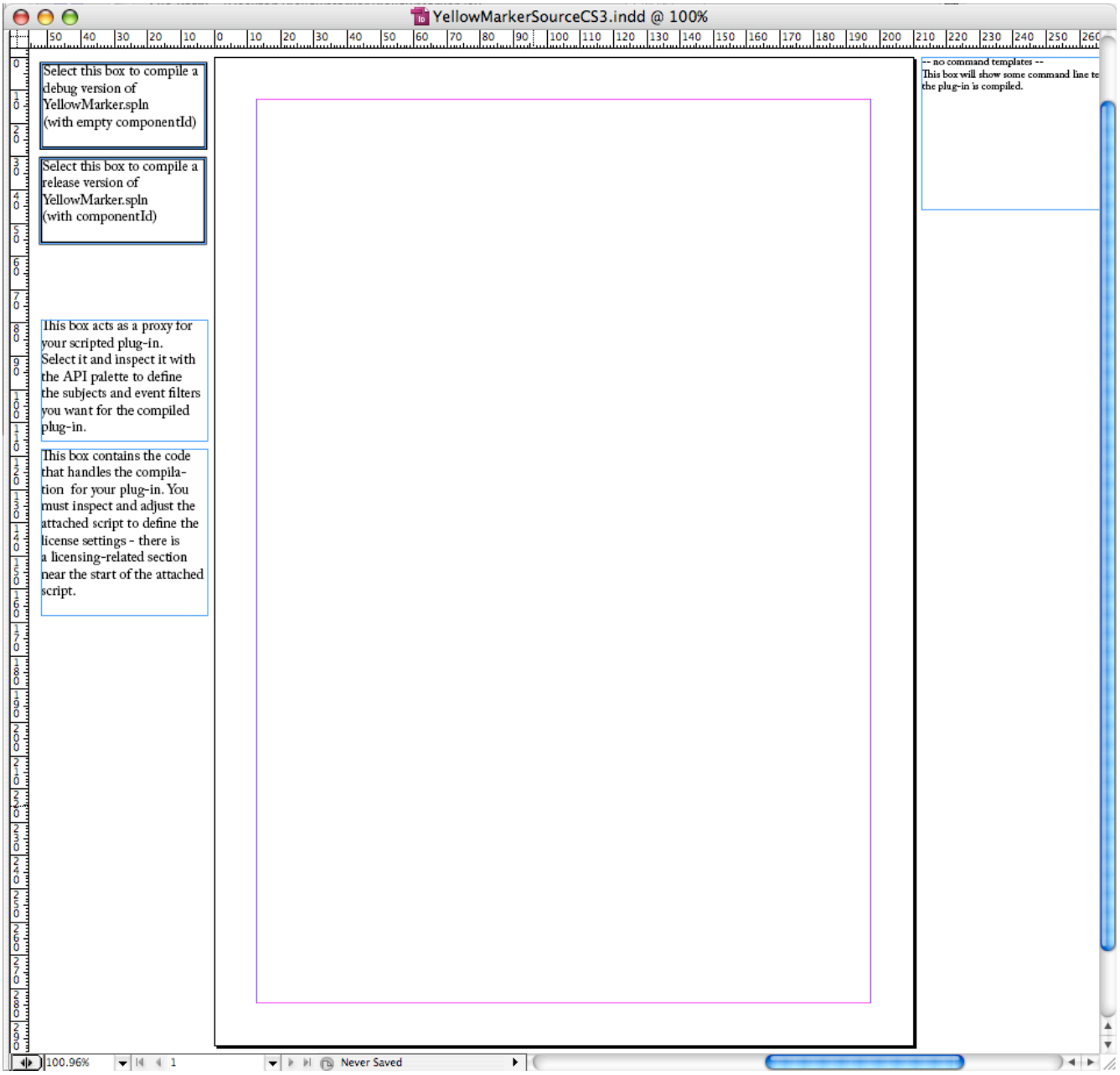
Fill this dialog with the required values – choose a plug-in name, choose a random password, and change the copyright string to suit:



Click OK. You now have to select a location to save the project folder. I've selected my Desktop:



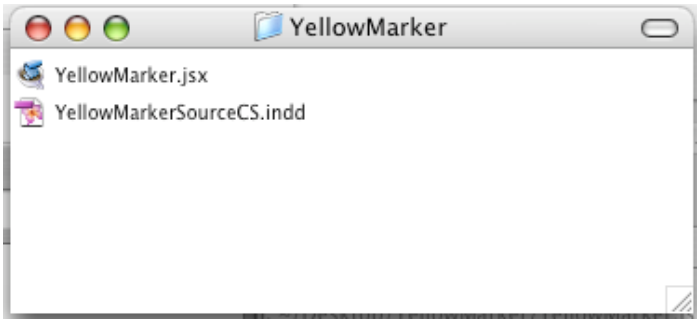
Click Choose. Finally, you are presented with a 'source' document – this document can be used to develop/debug/test your source code, and when you're ready, you can compile your source code into a scripted plug-in:



The above screen shot is made with InDesign CS3 – other versions would create similar documents, but with the name like *YellowMarkerSourceCS.indd* or *YellowMarkerSourceCS2.indd* or similar.

In what follows, we'll refer to *YellowMarkerSourceCSx.indd* in what follows to signify any one of the possible file names.

The contents of the source folder on our desktop look similar to this:



There are two files (as well as possibly an InDesign lock file – ignore it). The file YellowMarker.jsx is used to store our source code. The file *YellowMarkerSourceCS.indd* (or *YellowMarkerSourceCS2.indd* or *YellowMarkerSourceCS3.indd* or similar) is used to run/debug/test our source code.

Adding the source code

Let's first put our source code into the *YellowMarker.jsx* file. You can use any text editor – just make sure you save the file as ASCII text (some editors save RTF or Unicode text without telling you – that won't work!)

I've put the following source code into *YellowMarker.jsx* – it's not a complicated script, and I won't go into too much detail about it – study it at your leisure.

Do not forget to save. Forgetting to save is the biggest gotcha in this setup – the *YellowMarkerSourceCSx.indd* document will run whatever was last saved. If you forget to save, you'll be running an older version, and experience some frustration as a result.

```
//
// YellowMarker.jsx
// This script should be in the same folder as the
// document YellowMarkerSourceCS.indd
// It is automatically called by the
// YellowMarkerController page item.
//

// ******* Main program

if (theItem.eventCode == "menuYellowMarker")
{
  var markerColor = GetMarkerColor();
  Mark(app.selection,markerColor);
}
else if (theItem.eventCode == "docLoaded" || theItem.eventCode == "selected")
{
  app.registerMenuItem("menuYellowMarker","Apply Yellow Marker");
}

// ******* Subroutines

function Mark(theSelection, markerColor)
{
  if (theSelection instanceof Array)
  {
    for (var idx = 0; idx < theSelection.length; idx++)
    {
      Mark(theSelection[idx],markerColor);
    }
  }
else if
  (
    theSelection instanceof Text
   ||
    theSelection instanceof Word
   ||
    theSelection instanceof Paragraph
   ||
    theSelection instanceof TextColumn
  )
  {
    theSelection.underline = true;
    theSelection.underlineWeight = theSelection.pointSize;
    theSelection.underlineOffset = -theSelection.descent;
    theSelection.underlineColor = markerColor;
  }
}

function GetMarkerColor()
{
  var markerColor = app.activeDocument.swatches.item("MarkerYellow");
  var theErr;
  try
  {
   var name = markerColor.name;
  }
  catch (theErr)
  {
   markerColor = app.activeDocument.colors.add();
   markerColor.name = "MarkerYellow";
```

```
    markerColor.model = ColorModel.process;
    markerColor.space = ColorSpace.cmyk;
    markerColor.colorValue = [0, 0, 100, 0];
   }
   return markerColor;
}
```

Check the main program and note it expects one of three event codes: *menuYellowMarker*, *docLoaded*, and *selected*.

docLoaded

First, let's examine the use of the *docLoaded* event code. This code will be received by our scripted plug-in each time a new document is loaded.

What we need to do is to register the new menu item we want to add to the menu bar – that is achieved by calling the *registerMenuItem* method. This method accepts two parameters: a user-defined event code (which **must** start with the prefix *menu…*) , and a text string to show on the menu bar.

Each time the user selects the menu item, an event code *menuYellowMarker* will be sent to all page items which are waiting for this event code by means of their event filter expression. In this case, we'll only have our single scripted plug-in watch out for this event code.

Interesting detail: each time you open a new document, a new instance of the scripted plug-in is created, and its *docLoaded* handler called. That means that each time we open a new document, the menu item is (re)-registered.

After launching InDesign, but before the first document is opened, the *Apply Yellow Marker* menu item is not visible in the *API* menu – it only appears after the first document was created or opened.

The menu item remains there, and will be disabled when no documents are open.

menuYellowMarker

That brings us to processing of the second event code – *menuYellowMarker*: this is a user-defined event code, which we'll process by applying a yellow marker (by means of an oversized yellow underline) to any selected text.

selected

Finally, there is the event code *selected*. Its only reason for existence is to help us debug and test the plug-in.

If we only processed the *docLoaded* event code, we would have to save/close/re-open our source document *YellowMarkerSourceCSx.indd* every time we make a small change and we want to re-run our script.

Instead, we will process the *selected* event in the same way as the *docLoaded* event.

That way we can simply deselect (if necessary) and reselect the proxy page item that serves as a placeholder for the scripted plug-in in the source document. So instead of

having to close and re-open we can simply click the placeholder frame and have our script (re-)execute.

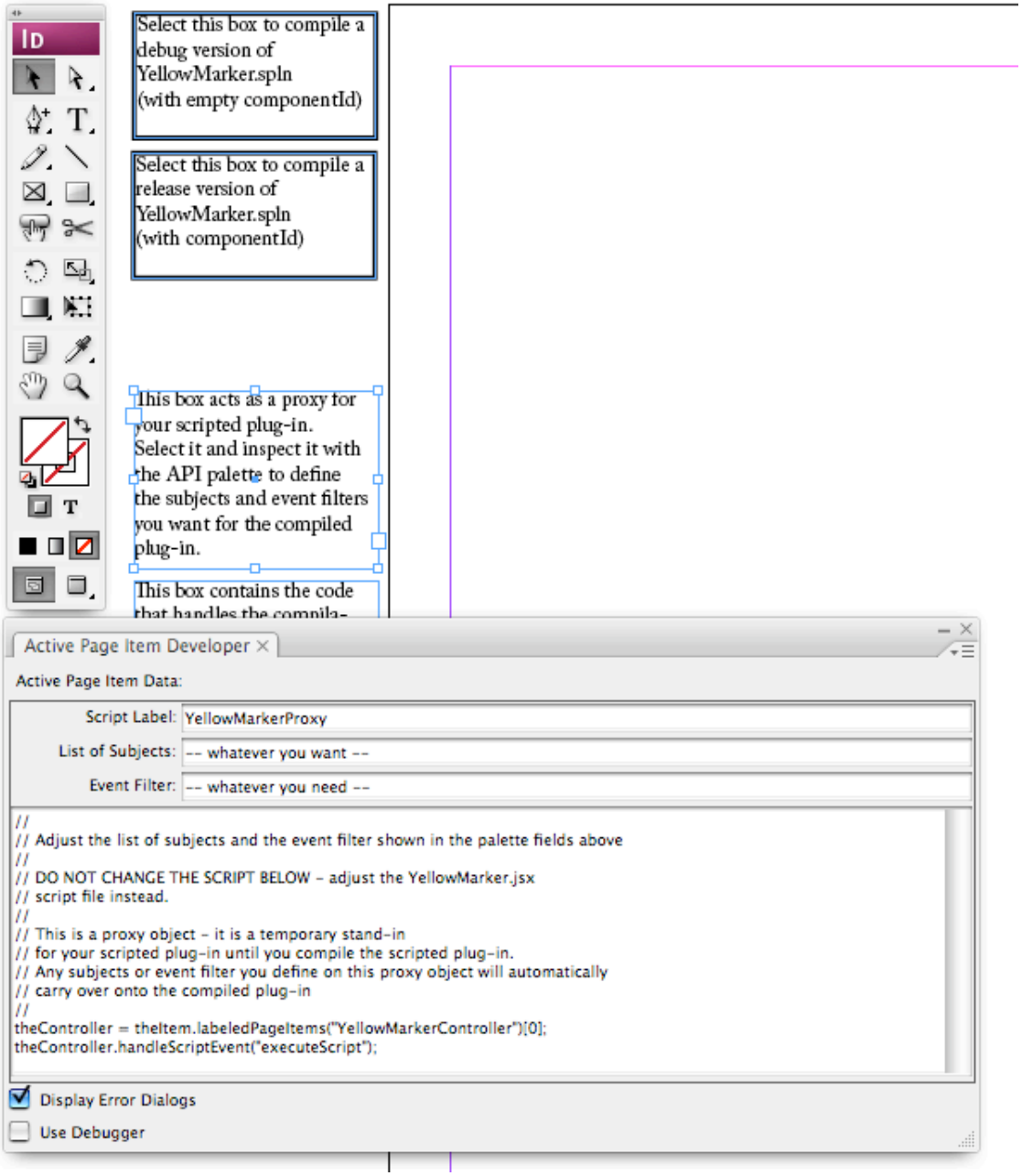
Testing the source code

After we've saved the source code from the text editor, we switch to InDesign.

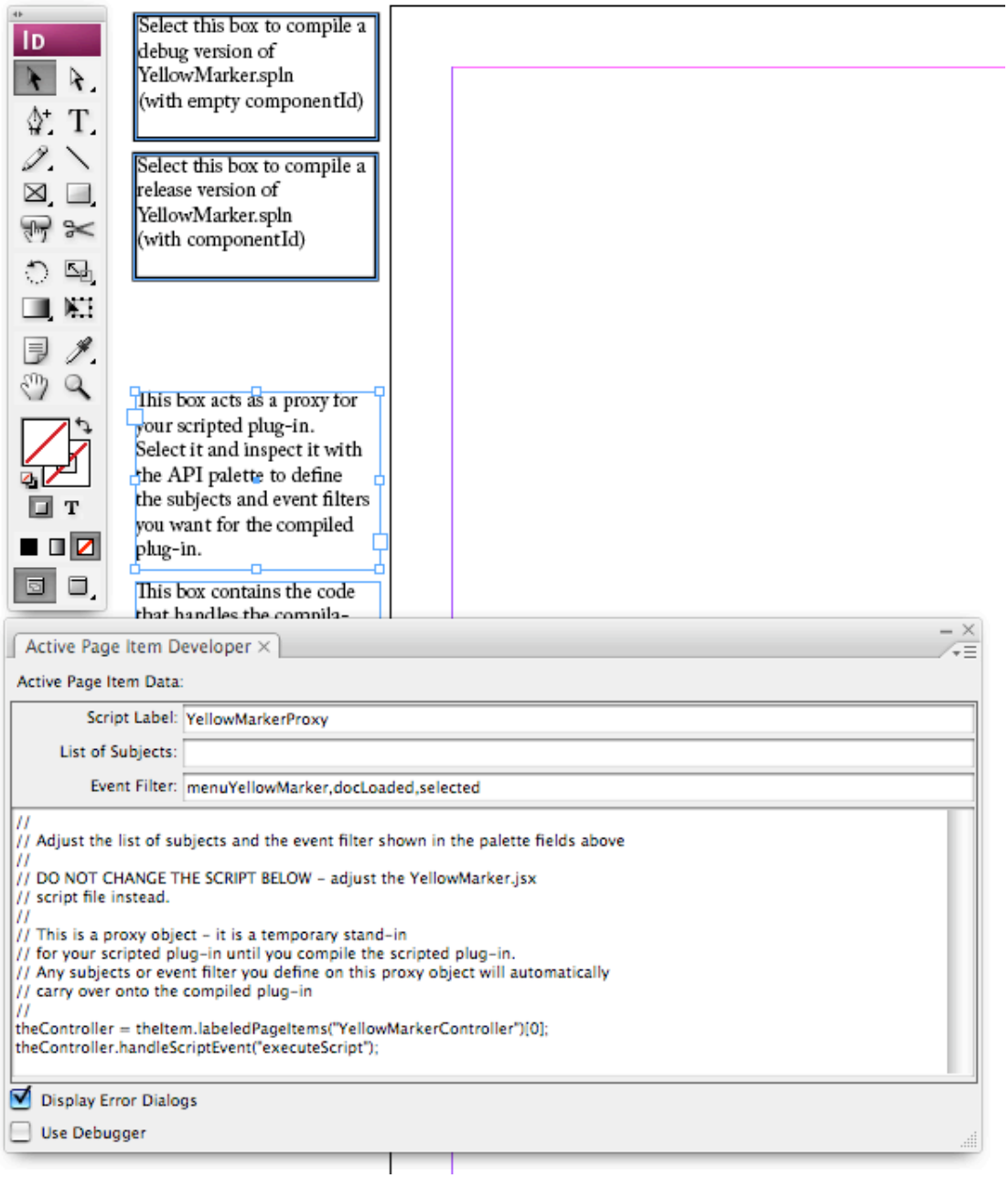
First, we need to set up the proper parameters for the scripted plug-in.

This is achieved by manipulating the placeholder page item (also known as 'the proxy') with the Active Page Item palette.

Select the proxy item, and make the Active Page Item palette visible.



Empty the list of subjects, and set the event filter so it captures our three event codes – *menuYellowMarker, docLoaded,* and *selected*.

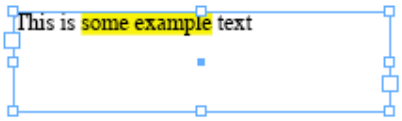


To run our source code, we can now simply select this proxy page item which represents our scripted plug-in for the time being.

Because the proxy item is currently selected, we need to deselect it first, and then re-select – clicking an already selected item has no effect; the *selected* event is only fired if the object changes from unselected to being selected.

Now check the menu bar – the *API* menu item should contain a new menu item:

Create a text frame in *YellowMarkerSourceCS.indd* and type some text in it. Select some of the words, and then select the new menu item.



Enabling and disabling the menu item

The new menu item still has a small problem: it remains enabled at all times, even when there is no text currently selected.

We can easily solve that by adding some extra source code to handle the predefined *enableMenus* event.

In this case, we only have to worry about a single menu item, but when you write a more complex scripted plug-in which manages multiple menu items, you will probably also need to take into account the *eventData* property. If the *eventCode* is *enableMenus* the corresponding *eventData* will contain the menu item's event code.

By checking `theItem.eventData` you can know which menu item needs to be tested for enable/disable.

We now modify the *YellowMarker.jsx* so it also processes the *enableMenus* event.

We also must add the *enableMenus* event to the event filter of the proxy object.

The source code is extended a little bit:

```
. . . <snip> . . .
// ******* Main program

if (theItem.eventCode == "enableMenus")
{
  //
  // Return desired menu state via our tempDataStore property
  //
  theItem.tempDataStore = HaveSomeTextSelected(app.selection);
}
else if (theItem.eventCode == "menuYellowMarker")
{
  var markerColor = GetMarkerColor();
  Mark(app.selection,markerColor);
}
. . . <snip> . . .
// ******* Subroutines

function HaveSomeTextSelected(theSelection)
{
  var textSelected = false;
  if (theSelection instanceof Array)
  {
    var idx = 0;
    while (! textSelected && idx < theSelection.length)
    {
      textSelected = HaveSomeTextSelected(theSelection[idx]);
      idx++;
    }
  }
else if
  (
    theSelection instanceof Text
   ||
    theSelection instanceof Word
   ||
    theSelection instanceof Paragraph
   ||
    theSelection instanceof TextColumn
  )
  {
    textSelected = true;
  }

  return textSelected;
}
. . . <snip> . . .
```

Don’t forget to save the modified script from your text editor!

To handle menu enabling or disabling we must handle the *enableMenus* event code, and return a boolean *true* or *false* value via the property *tempDataStore* of our item (no, using *return* does not work).

After this modification, we should see proper disabling/enabling of the menu item.

Be careful with the handling of *enableMenus* – your code should be as fast and efficient as possible, and should not take more than a fraction of a second to execute.

Compiling the scripted plug-in

After verifying everything works as expected, we are ready to compile the scripted plug-in.

First, inspect the source code attached to the compilation controller page item – it starts with a list of customizable parameters. Customize this section as desired. Shown below is an older version – as APIDTemplate evolves over time this code tends to change. For example, we could change areas marked in cyan from this:

```
//
// **** CONFIGURATION ****
//
kScriptName = "YellowMarker"
kCompiledPassword = "yoyo";
kCopyright = "(c) 2007 Rorohiko Ltd.";
kSPLNVersion = "1.0";

// Compact scripted plug-in code. Use with care - if it breaks your plug-in,
// then keep this set to false.
kCompressWhenCompile = (theItem.eventCode != "buttonPushDebug");

//
// The code below defaults the min version number to the version
// you are currently using. Can be changed to an explicit version
// number if desired (e.g. kMinAPIVersion = "1.0.23"; )
//
kMinAPIVersion = theItem.getDataStore("$VERSION$").split(".",3).join(".");

// ***** LICENSING INFO BEGIN *****

//
// If you want to use the licensing system, you must adjust the constants below
// kIsFree must be set to false, and the other constants adjusted
//
// There are three demo timeout modes - you can use one, two or all three at the
// same time. You MUST select at least one mode when setting kIsFree to false.
// When using multiple modes concurrently: the first mode that times out causes the
// plug-in to lapse.
//
// Mode 1:  interval. From the day of the first use, the demo can be used for
// kMinDemoDays consecutive days.
//
// Mode 2: actual use days. Days of actual use are counted. These days are
// not necessarily consecutive - so 5 non-consecutive days can for example
// spread over 5 weeks or more.
//
// Mode 3: hard cut-off date. The demo can be used up to a hard-coded
// cut off date.
//
//

kIsFree = true;

//
// Length of demo interval from first day of use. Can be -1
//
kMinDemoDays = 30;

//
// Minimum of (non-consecutive) actual use days - e.g. if the user starts the demo
once
// on a particular date, then does not use it for six months, and then tries
// again, he will still be allowed to use it for another 19 days.
//
kMinActualUseDemoDays = 20;

//
// "Hard" cutoff date for demo mode - unlicensed plug-in will stop
// working after this date, independent of minDemoDays and minActualUseDemoDays
// -1,-1,-1 if no cutoff needed
```

```
//
kDemoEndYear = -1;
kDemoEndMonth = -1;
kDemoEndDay = -1;

//
// If kLicenseURL is empty, the "Get License..." button will be disabled.
//
// ^1 will be replaced by the InDesign serial number (first 20 characters
// of the 24-character activation code).
// ^2 will be replaced by a URL encoded form of the plug-in name
// ^3 will be replaced by a unique system identifier (format xx:xx:xx:xx:xx:xx)
// ^4 will be replaced by a 'R', 'D', or 'E' to indicate the current API license
// already installed
// 'R' means no license installed - only APIR is licensed on this computer
// (because it is free)
// 'D' means APID is already licensed on this computer
// 'E' means an APIE license is installed (which encompasses APID as well)
...
```

kLicenseURL = "";

```
//
// if kLicenseMessage is empty (""), the "beg window" on startup will be disabled.
//
// In the two messages below:
// ^1 will be replaced by the plugin name.
// ^2 will be replaced by the number of demo days remaining
//
kLicenseMessage =
   "You are currently using a demo version of ^1," +
   " which will disable itself in ^2 days.";

//
// if kTimedOutMessage is empty (""), the "beg window" on time out will be disabled.
//
kTimedOutMessage =
   "A lapsed demo version of ^1 is installed.";

// ***** LICENSING INFO END *****
```

into the areas marked in yellow, like this:

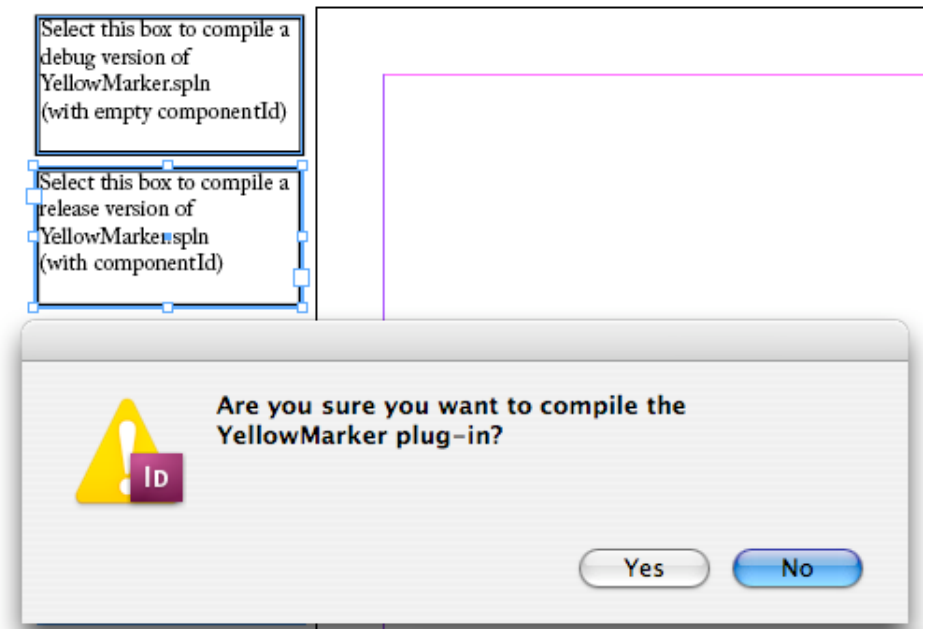
```
//
// **** CONFIGURATION ****
//
kScriptName = "YellowMarker"
kCompiledPassword = "yoyo";
kCopyright = "(c) 2007 Rorohiko Ltd.";
kSPLNVersion = "1.0";

// Compact scripted plug-in code. Use with care - if it breaks your plug-in,
// then keep this set to false.
kCompressWhenCompile = (theItem.eventCode != "buttonPushDebug");

//
// The code below defaults the min version number to the version
// you are currently using. Can be changed to an explicit version
// number if desired (e.g. kMinAPIVersion = "1.0.23"; )
//
kMinAPIVersion = theItem.getDataStore("$VERSION$").split(".",3).join(".");

// ***** LICENSING INFO BEGIN *****

//
// If you want to use the licensing system, you must adjust the constants below
// kIsFree must be set to false, and the other constants adjusted
//
// There are three demo timeout modes - you can use one, two or all three at the
// same time. You MUST select at least one mode when setting kIsFree to false.
// When using multiple modes concurrently: the first mode that times out causes the
// plug-in to lapse.
//
// Mode 1:  interval. From the day of the first use, the demo can be used for
// kMinDemoDays consecutive days.
//
// Mode 2: actual use days. Days of actual use are counted. These days are
// not necessarily consecutive - so 5 non-consecutive days can for example
// spread over 5 weeks or more.
//
// Mode 3: hard cut-off date. The demo can be used up to a hard-coded
// cut off date.
//
//
kIsFree = false;

//
// Length of demo interval from first day of use. Can be -1
//
kMinDemoDays = 10;

//
// Minimum of (non-consecutive) actual use days - e.g. if the user starts the demo
once
// on a particular date, then does not use it for six months, and then tries
// again, he will still be allowed to use it for another 19 days.
//
kMinActualUseDemoDays = 10;

//
// "Hard" cutoff date for demo mode - unlicensed plug-in will stop
// working after this date, independent of minDemoDays and minActualUseDemoDays
// -1,-1,-1 if no cutoff needed
//
```

```
kDemoEndYear = -1;
kDemoEndMonth = -1;
kDemoEndDay = -1;

//
// If kLicenseURL is empty, the "Get License..." button will be disabled.
//
// ^1 will be replaced by the InDesign serial number (first 20 characters
// of the 24-character activation code).
// ^2 will be replaced by a URL encoded form of the plug-in name
// ^3 will be replaced by a unique system identifier (format xx:xx:xx:xx:xx:xx)
// ^4 will be replaced by a 'R', 'D', or 'E' to indicate the current API license
// already installed
// 'R' means no license installed - only APIR is licensed on this computer
// (because it is free)
// 'D' means APID is already licensed on this computer
// 'E' means an APIE license is installed (which encompasses APID as well)
...
kLicenseURL = "http://www.rorohiko.com/buyourstuff?ID=^1&PI=^2&SI=^3&LL=^4";

//
// if kLicenseMessage is empty (""), the "beg window" on startup will be disabled.
//
// In the two messages below:
// ^1 will be replaced by the plugin name.
// ^2 will be replaced by the number of demo days remaining
//
kLicenseMessage =
    "You are currently using a demo version of ^1," +
    " which will disable itself in ^2 days.";

//
// if kTimedOutMessage is empty (""), the "beg window" on time out will be
disabled.
//
kTimedOutMessage =
    "A lapsed demo version of ^1 is installed.";

// ***** LICENSING INFO END *****
```

This sets our compiled plug-in up to be compressed, non-free, with a demo period of 10 days.

If the user wants to purchase the plug-in he'll be redirected to the URL http://www.rorohiko.com/buyourstuff, and pass the InDesign serial number, the plug-in name, the system id and the current API license level as URL parameters called 'ID', 'PI', 'SI', and 'LL'.

Switch back to InDesign and select the release compilation 'button':

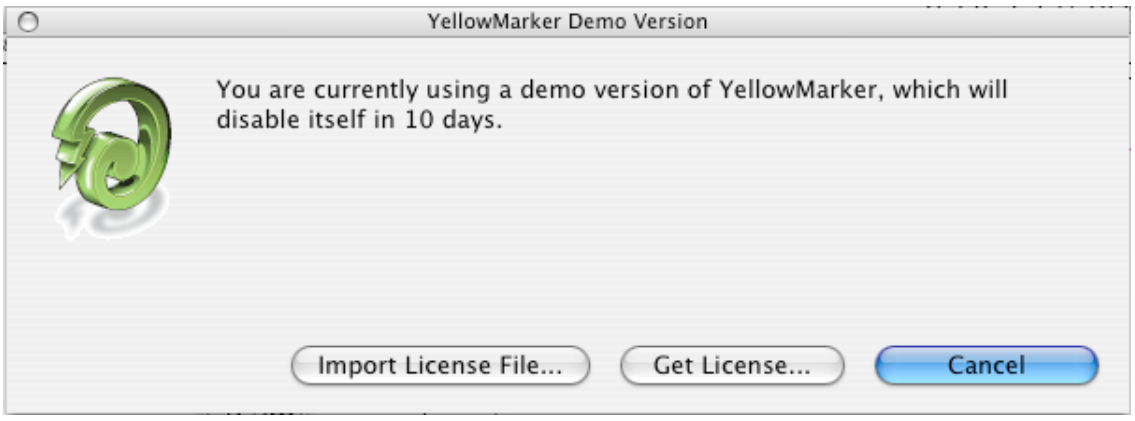


Click *Yes* to compile the plug-in. From this moment on, you have to be careful when manipulating the *YellowMarkerSourceCSx.indd* document.

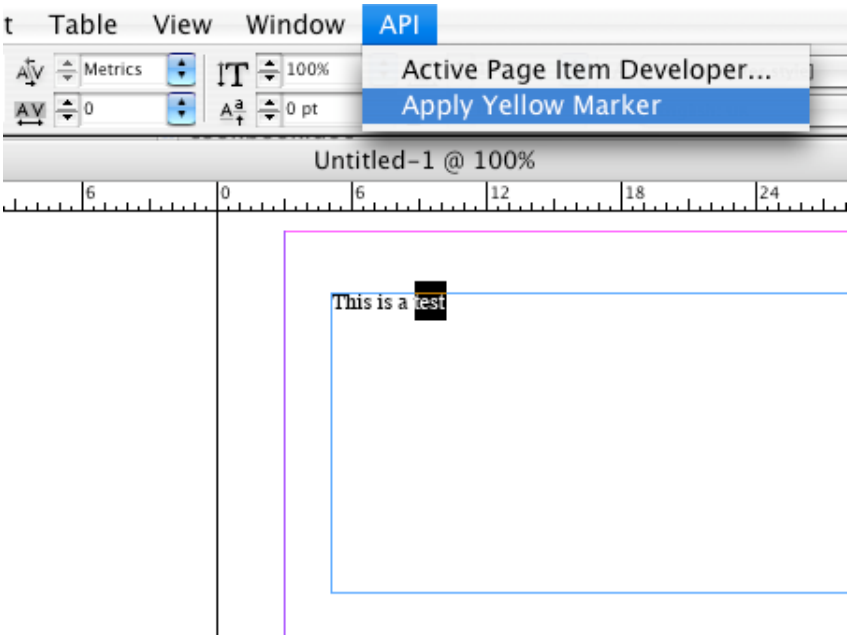
Because of the compilation, there can now be two duplicates of our YellowMarker plug-in active concurrently. One is contained in the proxy object, the other originates from the newly compiled scripted plug-in.

When you open the *YellowMarkerSourceCSx.indd* at a later point in time with the *YellowMarker.spln* installed, you will have **two** copies of the YellowMarker fighting for attention.

Close the *YellowMarkerSourceCSx.indd* and then open or create another document and verify the scripted plug-in is active. First you'll be presented with the 'begging' dialog which contains the data that was specified in the licensing area of the the compilation controller page item script:



Click past the dialog, and test the yellow marker function:



Deployment

Your plug-in is now ready for deployment: you can put the *YellowMarker.spln* file from your plug-ins folder on your web site.

Your customers will be able to try your scripted plug-in out for 10 days, after which the plug-in becomes inactive.

Your customers will have to install a copy of the *APID ToolAssistant*, a demo of which can be downloaded from our web site:
http://www.rorohiko.com/apidtoolassistant.html

The *APID ToolAssistant* we have available for download on our web site is a demo version, allowing your customer to try things out for about a month.

For continued use your customer will have to purchase a license for the *APID ToolAssistant* from us (Rorohiko Ltd.) – directly or indirectly – one license for each installed copy of InDesign used by the customer.

If the customer has, for example, both InDesign CS and CS2 installed, and wants to use both, two licenses are needed. If the user has a laptop and a desktop with the same serial number, two licenses to *APID ToolAssistant* will be needed.

Even if the customer uses multiple scripted plug-ins, the customer needs to purchase the *APID ToolAssistant* only once per installed copy of InDesign – i.e. multiple separate plug-ins that were all created using *APID Toolkit* can all be used with the customers' single installed and licensed copy of *APID ToolAssistant*.

Independently from the license for *APID ToolAssistant*, your customer will need to get a license for *YellowMarker.spln* from you.

It is up to you to decide how to handle this deployment – for example, you could pre-purchase single copies of *APID ToolAssistant* from us and bundle them with your plug-in – simply adding the cost of *APID ToolAssistant* to your selling price, or alternatively, you

could direct your customer to our web site to purchase *APID ToolAssistant* directly from us. A third option is available for APID 1.0.44 and higher: for approved component ids you can provide your customer with a combined license file that includes both the license for APID and the license for your tool.

You're also allowed to bundle the demo version of *APID ToolAssistant* with your demo version of your plug-in.

When a customer needs a license of your plug-in, you will need to use the *APIDLicenseGenerator* command line tool to generate a license file, or for approved components, you'll have to contact Rorohiko to get a proper license file.

This license file is generated based on the serial number of your customers' copy of InDesign, combined with the plug-in name (*YellowMarker)*, and the system id.

The *APIDTemplate* used to compile your plug-in has a text frame on the right of page 1 which shows examples of the command lines to use after compiling your plug-in.

The license-generation process can be automated fairly easily, if so desired, via a web site – the 'Get License…' button on the beg-screens can be configured fire off a URL that contains the needed information for automatically generating a license file on the web server.

4.4. Opening up your plug-in to external scripts

At some point in time, you might be interested in allowing some access to the functions of your scripted plug-in to other script developers.

By default, their access is limited: compiling the plug-in pretty much closes it off for any direct scripted access.

For example, it is not possible to simply send events to a compiled plug-in using the *handleScriptEvent* method – these calls are ignored for objects with compiled JavaScripts attached.

However, you can selectively open up access by defining some user-defined event codes that start with the prefix *external…*

This prefix gets special treatment, and any events with this prefix that are sent to a compiled component will be allowed through.

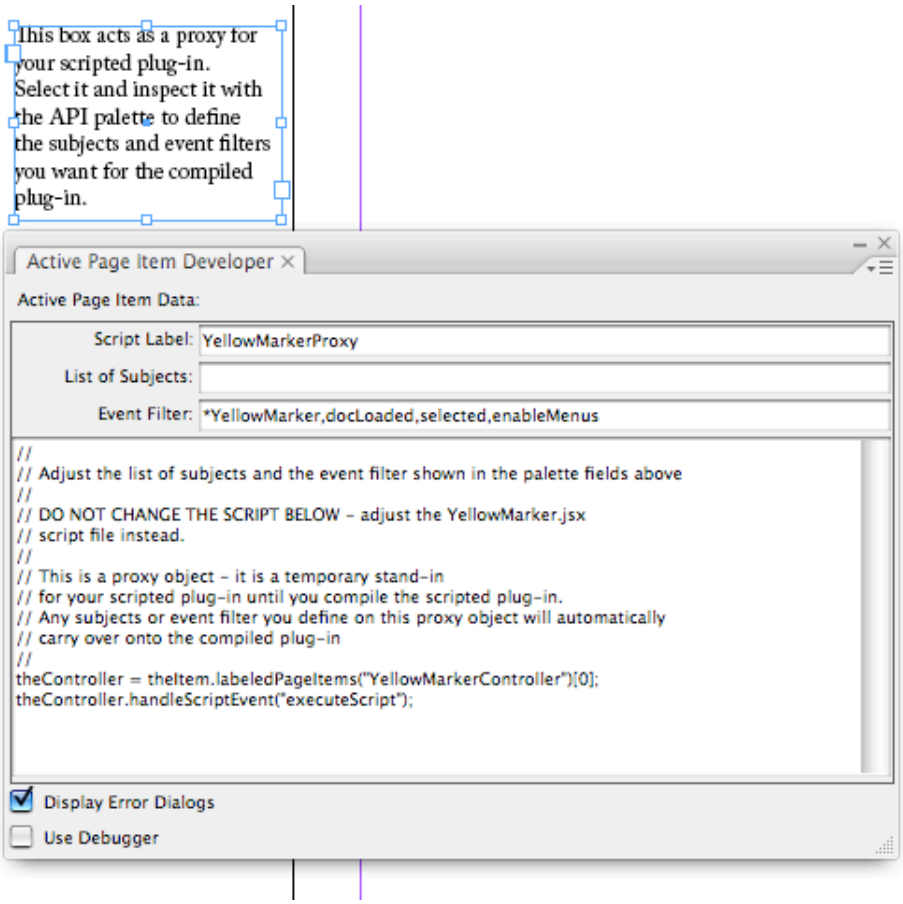
As an example, we'll extend the *YellowMarker.spln* example from section 4.3 so it accepts events from 'outside' and then we'll show how to use some example AppleScript and JavaScript to automate the *YellowMarker.spln*.

The change to the YellowMarker source code is minor: we simply add a new user-defined event *externalYellowMarker* which performs the exact same function as *menuYellowMarker*.

Because this new event starts with *external…* we will be able to send it to the scripted plug-in from the outside. First, add the new event code to the *YellowMarker.jsx* source file:

```
// ******* Main program

if (theItem.eventCode == "enableMenus")
{
  //
  // Return desired menu state via our tempDataStore property
  //
  theItem.tempDataStore = HaveSomeTextSelected(app.selection);
}
else if (theItem.eventCode == "menuYellowMarker" ||
  theItem.eventCode == "externalYellowMarker")
{
```

We also need to change our proxy object in the *YellowMarkerSourceCSx.indd*. We could simply add *externalYellowMarker* to the list of events in the event filter; in this example, we've chosen to instead change the string *menuYellowMarker* into **YellowMarker* which is a wild-card expression that will match any event that ends in …*YellowMarker*.



Recompile the plug-in, and close *YellowMarkerSourceCSx.indd*.

If you are using a Macintosh to work through this cookbook, you can fire up the *Script Editor* (in *Applications/AppleScript*) and type the following script (modify the application name accordingly if you are using CS, CS2 or CS4 instead of CS3):

```
tell application "Adobe InDesign CS3"
  set theDocument to active document
  tell theDocument
    set yellowMarkerPluginList to loaded scripted plugins label "yellowmarker"
    set yellowMarkerPlugin to item 1 of yellowMarkerPluginList
    tell yellowMarkerPlugin
      handle script event code "externalYellowMarker"
    end tell
  end tell

end tell
```

Open a new document, create a text frame and type some text in it. Select some of the text. Switch back to the Script Editor and run the script. The selected text in InDesign should become yellow-marked.

Note that the scripted plug-in label is in lowercase: when a scripted plug-in is loaded by a document, the label used will be its file name converted to lower case.

The same can be achieved with a JavaScript. Create a JavaScript source file *TestYellow.js* which contains:

```
var theDoc = app.activeDocument;
var yellowMarkerPluginList = theDoc.loadedScriptedPlugins("yellowmarker");
var yellowMarkerPlugin = yellowMarkerPluginList[0];
yellowMarkerPlugin.handleScriptEvent("externalYellowMarker");
```

Go to your InDesign application folder, and navigate into the *Presets* subfolder and then into the *Scripts* folder. Save the *TestYellow.js* file there.

Open a new document, create a text frame and type some text in it. Select some of the text. Bring up the Scripts Palette and double-click the *TestYellow.js* entry.

Again, note the label of the scripted plug-in: it's all lowercase. It is the same as the filename of the scripted plug-in, with the *.spln* extension stripped off and converted to lower case.

The same results can also be achieved with VBScript.

4.5. Hybrid JavaScript/C++

This is an advanced subject – if you don't have any experience with the InDesign SDK, you'll have to accept that some of the things discussed here won't make much sense. But if you don't know what this is about, don't worry – you probably don't need it.

Where speed is an issue, one can easily mix and match JavaScript with some C++ code.

The easiest way to access C++ code from any APID-based ExtendScripts is by adding a new interface to some of the page item boss class(es), or to the ScriptedPlugin boss class.

The interface you'll be adding corresponds to the following abstract C++ class (stored in the file *IActivePageItemExtension.h*):

```
#ifndef __IActivePageItemExtension_h__
#define __IActivePageItemExtension_h__

#include "ScriptData.h"
#include "ActivePageItemExtensionID.h"

///
/// Interface to a C++ extension aggregated onto a page item
///

class IActivePageItemExtension : public IPMUnknown
 {
public:
  enum    {kDefaultIID = IID_IACTIVEPAGEITEMEXTENSION};

  virtual ErrorCode
    CallExtension(
              ScriptData& returnData,
        const ScriptData& par1 = ScriptData(),
        const ScriptData& par2 = ScriptData(),
        const ScriptData& par3 = ScriptData(),
        const ScriptData& par4 = ScriptData(),
        const ScriptData& par5 = ScriptData()) = 0;
 };

#endif // __IActivePageItemExtension_h__
```

This interface is a one-to-one mapping onto the *callExtension* JavaScript method which is provided by the APID plug-in.

Attached to this interface you also need to add a corresponding implementation which will contains the C++ code you'll be calling from JavaScript.

In the *APID Toolkit* example files, you'll find two example projects with associated CodeWarrior or Xcode (Mac) and Visual Studio (Windows) project files which can serve as a starter for your own projects.

An important detail: you might need to aggregate the *IActivePageItemExtension* interface on *kPageItemBoss* as well as on *kActivePageItemScriptedPluginBoss*.

Before your scripted plug-in is compiled, you could be using a proxy object, which is a placeholder plage item, hence represented by a *kPageItemBoss*.

After compiling (depending on how you've set things up), your proxy object would possibly become a scripted plugin object instead, which is represented by a *kActivePageItemScriptedPluginBoss* instead.

However, it is also perfectly possible that you only need the interface on *kPageItemBoss* – you might decide to call the *callExtension* method only on page items.

5. How-to examples

5.1. Locking InDesign during idle processing

Sometimes, you have a situation where you'd like to do some processing during idle time. Yet, at the same time, you'd prefer the user to be locked out of the user-interface until the processing is finished.

Active Page Items has some functionality which is accessible via the 'callExtension' event of the Application object.

The three relevant features are: *SetApplicationModalLock* (opcode 10001), and *ClearApplicationModalLock* (opcode 10002), and *IsApplicationModalLocked* (opcode 10003).

Calling *app.callExtension(0x90b6c,10001)* will lock InDesign into a modal mode; the call *app.callExtension(0x90b6c,10002)* will release the modal lock.

Calling *app.callExtension(0x90b6C,10003)* returns a Boolean that tells you whether InDesign is currently modally locked.

Attach the following event handler script to any page item, and set the page item's event filter to *selected*.

```
if (theItem.eventFilter == "selected")
{
  theItem.eventFilter = "selected,idle";
  theItem.dataStore = 0;
  app.callExtension(0x90b6C,10001);
}
else
{
  if (theItem.dataStore > 10)
   {
     theItem.eventFilter = "selected";
     app.callExtension(0x90b6C,10002);
   }
  else
   {
    theItem.dataStore = theItem.dataStore + 1;
   }
}
```

When the page item is clicked, its handler will first change the event filter to be *selected,idle*. Then, each time the idle event is received, it will add one to *theItem.dataStore*, until, after about 11 seconds, *theItem.dataStore* reaches 11, at which time the event filter is set back to *selected*.

During this 11-second idle time, the application will be locked in modal mode – making sure the user cannot change anything via the user interface.

This locking mechanism is especially useful when calling out to external applications: while the external application is processing, we often don't want the user to 'fiddle around' with InDesign.

By using these functions we can be sure that InDesign is locked against user interactions while the external application executes.

5.2. Concurrently capturing page items with multiple controllers

A simple trick to include or exclude certain page items from processing by a particular controller is to use the page item's script label and change it on-the-fly to some particular string which is matched or not matched by an observer's subject list.

If you want to avoid having to re-type the code below, look in the example documents that accompany the APID download for *autoSquareAutoShadeCS.indd*.

A script-label activated controlled

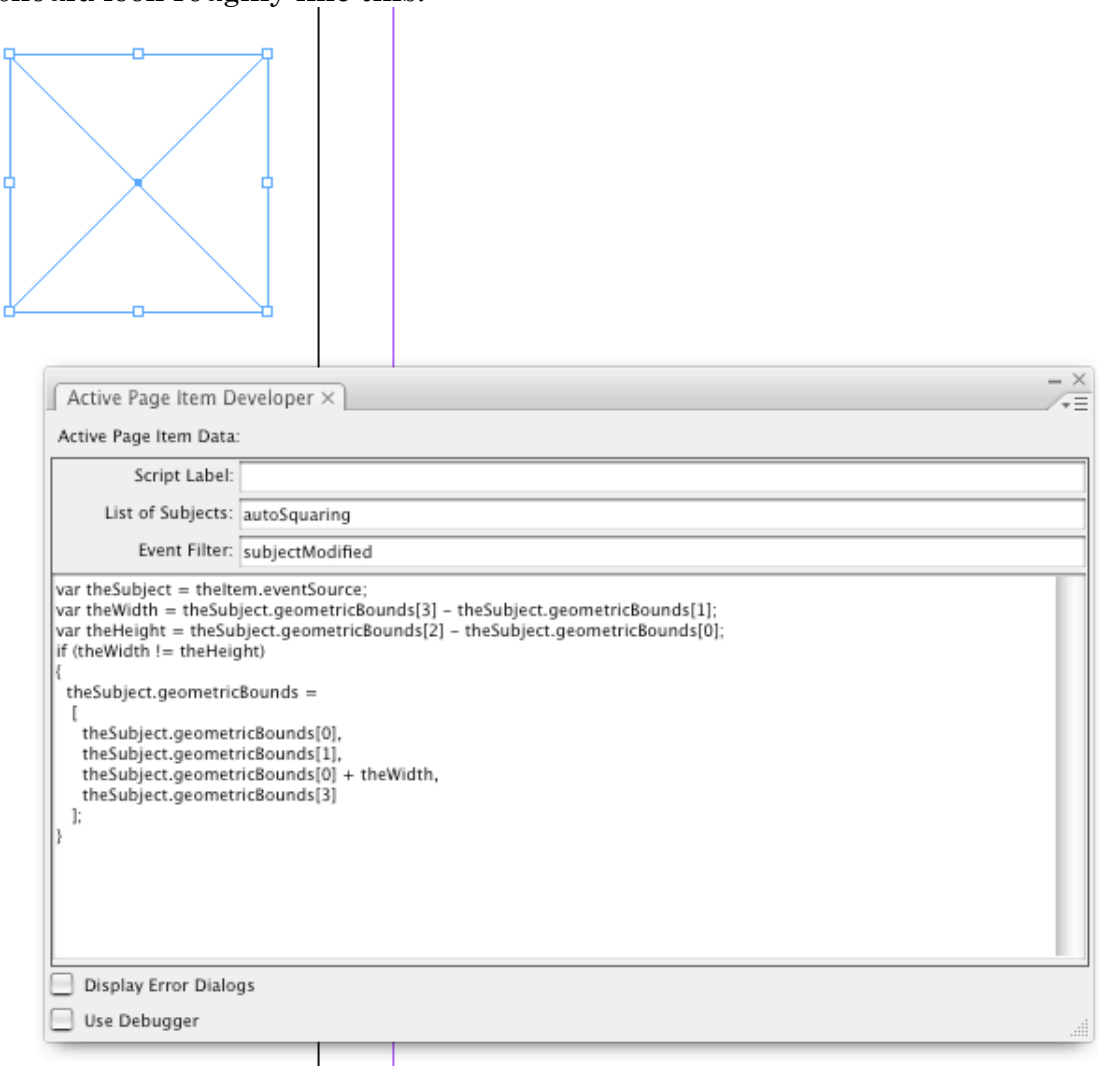
An example: create a new document, and create a frame on the pasteboard – this frame will become our new controller.

Bring up the *Active Page Item Developer* palette, select the frame, and set the *List of Subjects* to `autoSquare`, and the *Event Filter* to `subjectModified`.

Insert the following script:

```
var theSubject = theItem.eventSource;
var theHeight = theSubject.geometricBounds[2] - theSubject.geometricBounds[0];
var theWidth = theSubject.geometricBounds[3] – theSubject.geometricBounds[1];
if (theWidth != theHeight)
{
  theSubject.geometricBounds =
   [
     theSubject.geometricBounds[0],
     theSubject.geometricBounds[1],
     theSubject.geometricBounds[0] + theWidth,
     theSubject.geometricBounds[3]
   ];
}
```

It should look roughly like this:



Now create a frame somewhere on the page, and set its script label to *autoSquaring*. Try resizing this frame. Because of its label, it is a subject of the controller, and the controller will force the frame to become and remain a square.

Make a few copies of the frame, all carrying the same script label – all these frames exhibit the same 'stay square' behavior.

To stop the behavior for a particular frame, you can clear the script label.

This is a good trick, but it falls apart when there is more than one controller and more than one behavior.

Adding a second script-label activated controller

To demonstrate the issue, create a second frame on the pasteboard, and select it. Set the *List of Subjects* to *autoShade*, and the *Event Filter* to *subjectModified*.

Insert the following script:

```
//
// Maximum drop shadow distance
//
const kDropShadowMaxPoints = 20;

// do {} while (false); construct: if precondition fails,
// bail out with "break;"
do
{
  theSubject = theItem.eventSource;

  thePage = GetParentPage(theSubject);

  // Bail out if no parent page
  if (thePage == null)
    break;

  theDocument = GetParentDocument(thePage);

  // Bail out if no parent document
  if (theDocument == null)
    break;

  //
  // Switch the document measurements to points. Keep the original measurements
  // so we can restore them later
  //
  var savedHorizontalMeasurementUnits =
    theDocument.viewPreferences.horizontalMeasurementUnits;
  var savedVerticalMeasurementUnits =
    theDocument.viewPreferences.verticalMeasurementUnits;
  theDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
  theDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;

  //
  // Calculate some parent page attributes
  //
  var pageTopLeftX = thePage.bounds[1];
  var pageTopLeftY = thePage.bounds[0];
  var pageWidth = thePage.bounds[3] - pageTopLeftX;
  var pageHeight = thePage.bounds[2] - pageTopLeftY;
```

```
  var pageCenterX = pageTopLeftX + pageWidth / 2.0;
  var pageCenterY = pageTopLeftY + pageHeight / 2.0;

  //
  // Calculate similar attributes for the frame to be handled
  //
  var subjectTopLeftX = theSubject.geometricBounds[1];
  var subjectTopLeftY = theSubject.geometricBounds[0];
  var subjectWidth = theSubject.geometricBounds[3] - subjectTopLeftX;
  var subjectHeight = theSubject.geometricBounds[2] - subjectTopLeftY;

  var subjectCenterX = subjectTopLeftX + subjectWidth / 2.0;
  var subjectCenterY = subjectTopLeftY + subjectHeight / 2.0;

  //
  // How far from the page center is the frame?
  //
  var subjectRelativeDistanceX = (subjectCenterX - pageCenterX) / pageWidth;
  var subjectRelativeDistanceY = (subjectCenterY - pageCenterY) / pageHeight;

  //
  // The further away from the page center, the more pronounced
  // the drop shadow will be
  //
  var dropShadowDeltaX = subjectRelativeDistanceX * kDropShadowMaxPoints;
  var dropShadowDeltaY = subjectRelativeDistanceY * kDropShadowMaxPoints;

  //
  // Apply drop shadow. Different code needed for CS/CS2 (versions 3.0 and 4.0)
  // and CS3 (version 5.0)
  // Also take care not to cause an endless "subjectModified" event loop:
  // we only adjust the drop shadow parameters if they are not correct yet
  //
  if (parseFloat(app.version) < 5.0)
  {
    if (theSubject.shadowMode != ShadowMode.drop)
      theSubject.shadowMode = ShadowMode.drop;

    if (theSubject.shadowXOffset != dropShadowDeltaX)
      theSubject.shadowXOffset = dropShadowDeltaX;

    if (theSubject.shadowYOffset != dropShadowDeltaY)
      theSubject.shadowYOffset = dropShadowDeltaY;
  }
  else
  {
    if (theSubject.transparencySettings.dropShadowSettings.mode != ShadowMode.drop)
      theSubject.transparencySettings.dropShadowSettings.mode = ShadowMode.drop;

    if
(theSubject.transparencySettings.dropShadowSettings.xOffset != dropShadowDeltaX)
      theSubject.transparencySettings.dropShadowSettings.xOffset =
        dropShadowDeltaX;

    if
(theSubject.transparencySettings.dropShadowSettings.yOffset != dropShadowDeltaY)
      theSubject.transparencySettings.dropShadowSettings.yOffset =
        dropShadowDeltaY;
  }
```

```
  //
  // Restore the measurements to what they were
  //
  theDocument.viewPreferences.horizontalMeasurementUnits =
    savedHorizontalMeasurementUnits;
  theDocument.viewPreferences.verticalMeasurementUnits =
    savedVerticalMeasurementUnits;
}
while (false);

// End of event handler; utility functions below

function GetParentPage(pageItem)
{
  var page = null;
  do
  {
    var err;
    try
    {
      page = pageItem.parent;
    }
    catch(err)
    {
      page = null;
    }

    if (page == null)
    {
      break;
    }

    if (page instanceof Page)
    {
      break;
    }

    if (page == pageItem)
    {
      page = null;
      break;
    }

    pageItem = page;
  }
  while (true);

  return page;
}

function GetParentDocument(pageItem)
{
  var document = null;
  do
  {
    var err;
    try
    {
      document = pageItem.parent;
    }
    catch(err)
```

```
    {
      document = null;
    }

    if (document == null)
    {
      break;
    }

    if (document instanceof Document)
    {
      break;
    }

    if (document == pageItem)
    {
      document = null;
      break;
    }

    pageItem = document;
  }
  while (true);

  return document;
}
```

This script will influence any page items that have a script label set to *autoShade*, and it will apply a more or less pronounced drop shadow depending on how far removed they are from the page center. Try it out to get a feel for what it does.

The problem: suppose we'd love to get both behaviors grafted onto a page item: we want both *autoSquaring* and *autoShade*. Because both controllers use the script label to 'pick' their subject we seem to be caught in an either/or situation.

Wildcard characters to the rescue: the trick is to change the expression in the *List of Subjects* for both controllers. Change the *List of Subjects* on the first controller to **autoSquaring** and the *List of Subjects* on the second controller to **autoShade**. Now set the script label on one of your test frames to *autoSquaringautoShade* (simply concatenating the two script labels) and suddenly this particular frame will get both behaviors at the same time.

Avoiding false positives

There are still some issues with the above approach: depending on the script labels selected, we might accidentally get some false positives because of a possible unfortunate concatenation of the script labels.

To avoid this, the easiest is to use some special characters (for example < and >) to 'wrap' the script labels: the List of Subjects on both controllers become **<autoSquaring>** and **<autoShade>**, and we set the script label on the test frames to *<autoSquaring><autoShade>*.

Because of the mingled-in non-letters we avoid false positives.

For example, if the labels we picked would have been *active* and *reactive*, the label *reactive* would match both a *List of Subjects* set to **reactive** and one set to

active,the second one being a false positive.

Using *<active>* and *<reactive>* instead, and **<active>** and **<reactive>** in the lists of subjects avoids the false positive.

5.3. Clearing the undo stack in InDesign CS3

The functionality described below can only be used from 'regular' scripts; it does not work from an APID event handler.

Suppose you're running a lengthy script in CS3 that performs large amounts of undoable operations (e.g. dynamically building complex table layouts).

As a result, InDesign will build up a massive undo stack, which eventually will start slowing down things.

By regularly calling the following lines inside some of your script loops, you can clear the undo stack, which often leads to a marked performance improvement.

```
function DiscardUndoStack(theDocument)
{
  const IID_IACTIVEPAGEITEMSCRIPTUTILITIESEXTENSION = 0x90B6C;
  const kOpCode_DiscardUndo = 10006;

  app.callExtension(
    IID_IACTIVEPAGEITEMSCRIPTUTILITIESEXTENSION,
    kOpCode_DiscardUndo,
    theDocument);
}
```

You should call this function fairly often during processing.

How often is often enough? That depends on a lot of things; you probably don't want to call this function inside your innermost loops, as that might become counterproductive because of the functions overhead.

As a rule of thumb, when we use this function, we try to call it roughly after about 30 undoable operations. The 'sweet spot' for your script might be different – it pays to run a few tests and measure times.

5.4. Adding context menus to a text selection

From APID 1.0.43 onwards, you can also attach contextual menus to text selections.

The following example adds two context menu items to all frames in a document, and a third context menu item that will be used for text selections.

Create a frame on the pasteboard, set the *List of Subjects* to *** (i.e. all page items), and the *Event Filter* to *subjectLoadContextMenu, subjectRotateCW, subjectRotateCCW, subjectCapitals*.

Set the script of the controller frame to:

```
if (theItem.eventCode == "subjectLoadContextMenu")
{
  var theMenu = new Array();
  theMenu.push(["1/8 turn","subjectRotateCW"]);
  theMenu.push(["-1/8 turn","subjectRotateCCW",false]);
```

```
  if
  (
    app.selection instanceof Array
   &&
    app.selection.length == 1
   &&
    app.selection[0] instanceof Text
  )
  {
    theMenu.push(["All Caps","subjectCapitals",true]);
  }
  theItem.contextMenu = theMenu;
}
else if (theItem.eventCode == "subjectRotateCW")
{
  theItem.eventSource.absoluteRotationAngle += 45;
}
else if (theItem.eventCode == "subjectRotateCCW")
{
  theItem.eventSource.absoluteRotationAngle -= 45;
}
else if (theItem.eventCode == "subjectCapitals")
{
  app.selection[0].capitalization = Capitalization.allCaps;
}
```

Now create a text frame on the page, and insert some text into it. Use the normal selection tool to select the text frame, and right click it – there should be an API menu with two entries (*1/8 turn* and *-1/8 turn*).

Now switch to text selection mode and select some text in the frame, and then right-click the selected text. Now there should be an API menu with a single entry *All Caps*.

Each menu item in the context menu corresponds to a 3-item array (but the third item is optional, and if omitted is assumed to be `false`).

If the third item in the menu item's entry is `false`, then it is a 'normal' context menu that shows up in regular selection mode. If the third item in a menu item's entry is `true`, then the context menu will only appear in text selection mode.

Keep in mind that context menus are rebuilt on-the-fly. This example includes code that checks whether there is a usable text selection available before adding the third menu item. If you right-click in text selection mode without some text selected, then the third item won't appear.

5.5. Using your own menu entry instead of the 'API' menu.

APID normally uses simple strings for menu items and context menu items, and these strings appear in InDesign under a menu item 'API'.

However, you can also use a different menu item instead of this default 'API' menu item.

Internally, InDesign uses so-called 'menu paths' composed of multiple colon-separated strings to represent the location of a menu item. For example, the standard InDesign preferences menu path is `Main:&Edit:Preferences`.

To insert a new entry into the standard InDesign preferences menu you could use a menu item called `Main:&Edit:Preferences:My Own Preferences` instead of simply `Preferences`.

Normal menu items are all somewhere 'below' the *Main:…* root menu path entry.

Context menu items are somewhat different: they are either below a root entry called *RtMouseLayout* (for normal model context menus) or below a root entry called *RtMouseText* (for text selection context menus).

We will use the previous example to demonstrate how this works. Change the script in the previous example as follows:

```
if (theItem.eventCode == "subjectLoadContextMenu")
{
  var theMenu = new Array();
  theMenu.push(["RtMouseLayout:My Very Own Menu:1/8 turn","subjectRotateCW"]);
  theMenu.push(
    ["RtMouseLayout:My Very Own Menu:-1/8 turn","subjectRotateCCW",false]);
  if
  (
    app.selection instanceof Array
   &&
    app.selection.length == 1
   &&
    app.selection[0] instanceof Text
  )
  {
    theMenu.push(["RtMouseText:My Very Own Menu:All Caps","subjectCapitals",true]);
  }
  theItem.contextMenu = theMenu;
}
else if (theItem.eventCode == "subjectRotateCW")
{
  theItem.eventSource.absoluteRotationAngle += 45;
}
else if (theItem.eventCode == "subjectRotateCCW")
{
  theItem.eventSource.absoluteRotationAngle -= 45;
}
else if (theItem.eventCode == "subjectCapitals")
{
  app.selection[0].capitalization = Capitalization.allCaps;
}
```

This creates a new menu entry in the context menus called *My Very Own Menu* and it inserts the new menu items there instead of under the generic *API* heading that is used by default.

It is also possible to 'nest' multiple levels of menu items to create submenus and sub-submenus.

Remark: using submenus does not work properly for context menus in InDesign CS and CS2 – in those versions, the 'intermediate' menu levels are ignored. This is due to limitations of those versions. In InDesign CS3, things works fine.

5.6. Processing events with a non-default scripting engine

In InDesign CS3 and above, there is support for alternate, persistent ExtendScript 'engines'.

Building further onto that feature, APID allows you to 'route' specific events to a specific, named engine by appending a # and the engine name to the event code in the filter

expression.

One of the advantages of using a named engine is that named engines are persistent – i.e. if you define a global variable at one time, that global variable will retain its value and be available on the next invocation.

An example to demonstrate a 'selection counter' – a page item that counts how many times it has been selected. This example will only work in InDesign CS3; in CS and CS2, the engine name is ignored.

Create a new document in InDesign CS3, and set the *Event Filter* to *`selected#SelectionCounter`*.

This means that the page item will react to *`selected`* events, and these events will be processed by the event handler while running in a scripting engine called *`SelectionCounter`*.

Set the event handler script of the page item to:

```
var theCounter;
if (theCounter == undefined)
{
  theCounter = 1;
}
else
{
  theCounter++;
}
alert("This item was selected " + theCounter + " times");
```

Each time you deselect/reselect the item, you'll see a message with an ever-increasing counter value: the variable *`theCounter`* is persistent and keeps its value.

You can open and close the document as many times as you want – as long as you don't quit InDesign, the variable will remain available.

Now remove the engine name and change the event filter to simply *`selected`*, and try again.

This time, the value displayed for the counter will always be 1 because the default scripting engine is not persistent.

If you quit out of InDesign, then the persistent values provided by the named engine will be lost – so this mechanism is not feature-identical to what *`setDataStore()`* / *`getDataStore()`*, and *`insertLabel()`* / *`extractLabel()`* offer. (*`insertLabel()`* / *`extractLabel()`* are standard member functions provided by InDesign; *`setDataStore()`* / *`getDataStore()`* are provided by APID)

The critical difference is that these function pairs offer persistent storage of data attached *to the document*, not to the scripting engine. This persistent data is stored inside the document, and remains available even if InDesign is stopped and restarted.

You can use an *Event Filter* set to *`selected#SelectionCounter`* with InDesign CS and CS2 without ill effect– but the engine name will be ignored because InDesign CS and InDesign CS2 do not support alternate, persistent ExtendScript engines.

5.7. Locking InDesign in modal mode while an external application executes

APID provides a special member function called *launchWith()*, attached to the application object. *launchWith()* has a number of uses.

Its main purpose is to allow you to open a particular document with a particular application.

InDesign *File* objects have an *execute()* member function, but the problem with that is that when you *execute()* a document file, you have no control over which application will be used to open the file – similar to what happens to double-clicking a file.

launchWith() resolves this by allowing you to also designate the application file that must be used to open a particular document.

A second feature of *launchWith()* is that you can lock InDesign into modal mode at the time of launch, and InDesign will remain modally locked until the launched application terminates. While InDesign is locked, your script continues to run; you can also still process idle events.

This offers an easy way to integrate InDesign with external applications – at Rorohiko, we often use REALbasic to create little 'satellite' programs with more complex user interfaces than can easily be built in InDesign; typically these REALbasic programs will use a 'global floating palette' window to simulate a some 'palette-like' window. By using *launchWith()* we can better integrate these REALbasic applications: InDesign remains locked in modal mode until the satellite application terminates.

An example REALbasic project, together with an InDesign document and compiled versions of the REALbasic project are provided with the example material.

Look out for a folder called *ExampleSatelliteApp* – it contains an example application and an example document.

5.8. Create a shared subroutine library

See the example *LibraryPlugin* and *MyLibraryUser* plugins provided in the *APIDToolkit* release. *LibraryPlugin* sets up a shared subroutine library that is then used by the *MyLibraryUser spln* file.
In CS and CS2, libraries are not persistent, and are re-loaded each time they are needed. In CS3, because the APID engines are made to be persistent, a library will only be loaded once, and remain available.
Libraries need to be 'carried' by an object – so library subroutines are 'attached' to a library object.
The sample LibraryPlugin contains the following code:

```
myOwnLibrary = {};
myOwnLibrary.myfunction = (function(x)
{
  alert(x);
});

this.myOwnLibrary = myOwnLibrary;
```

This creates a sample library object with a single sample function, which is then 'grafted'

onto the global object 'this'.
The .spln source file subscribes to the user-defined event *loadLibrary.* The library-using code then looks like:

```
if (this.myOwnLibrary == undefined)
{
   var libraryPlugin = GetScriptedPlugin(theItem,"libraryplugin");
   libraryPlugin.handleScriptEvent("loadLibrary");
   alert("Library loaded");
}
this.myOwnLibrary.myfunction("test");
```

i.e. if the library has not been loaded, it is loaded from the *LibraryPlugin.spln* by sending the scripted plugin a *loadLibrary* event code.