OpenVXML Without Tears

Vikram Ramanarayanan, Zhou Yu
vramanarayanan@ets.org zhouyu@cs.cmu.edu

Version 1.10: 30th Aug 2015
(with inputs from the OpenVXML Setup and User Guides, developer interactions & the author's own experience)

Step I: Installation and Setup:

0. Install the Java Development Kit 8 from http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. Accept the license agreement and download the version corresponding to your local operating system (if this is Windows 64-bit, the link URL is http://download.oracle.com/otn-pub/java/jdk/8u20-b26/jdk-8u20-windows-x64.exe). Install the JDK 8 software. This is a prerequisite for installing Eclipse on your machine. Eclipse is an interactive developer environment which will allow you, among other things, to design call workflows.

1. Install Eclipse to a new location on your machine from the following URL (don't use the latest version, use the one at the following URL *only*): http://eclipse.org/downloads/packages/eclipse-rcp-and-rap-developers/keplersr2

2. Start the new Eclipse. Using the menu, select: Help → Install New Software...

3. In the Install dialog that appears, in the "Work with:" field at the top, add the VTP's update site: http://build.openmethods.com/downloads/OpenVXML5.0/repository/ and hit Enter. (Note: If nothing appears in the box below, you might want to try typing "*OpenVXML*" in the "Work with:" field and trying again).

4. Check the *OpenVXML 5.0* as well as the *Uncategorized* packages at the top and click Next. Note that this part takes a little while.

5. When the Install Details screen comes up, click Next >

On the Review Licenses screen, there aren't actually any licenses to review. Just click "I accept the terms..." and click Finish.

During the installation you'll need to click OK once, to allow unsigned content to be installed.

6. After the installation, restart Eclipse. The first time you open OpenVXML, you'll be in the default Plug-in Developer's perspective (a perspective is basically just a set of window tabs that are open simultaneously in your workspace; this is just for programming convenience). Change this to the OpenVXML perspective by clicking:

Window → Open Perspective → Other... → OpenVXML

In the dialog that appears, select OpenVXML and click OK.

7. Every VoiceXML project consists of 2 components – the *Voice*, which stores all your audio files and grammars (which are essentially the list of words/phrases you pass to the speech recognizer to choose from when it receives a voice input from the user), and the *Workflow*, which basically specifies a flowchart of the voice flow you want to design. If you are designing an item, you mainly need to focus on the workflow.

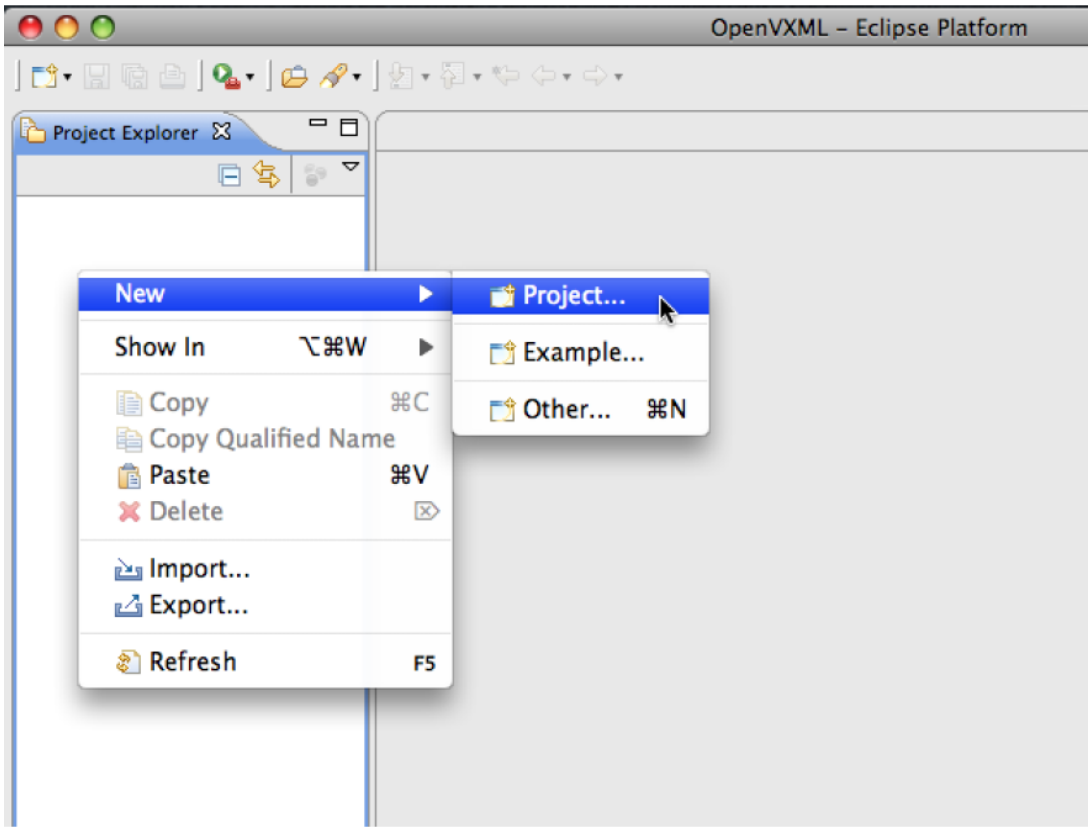
8. Once you're in the OpenVXML perspective, create a new Voice project.

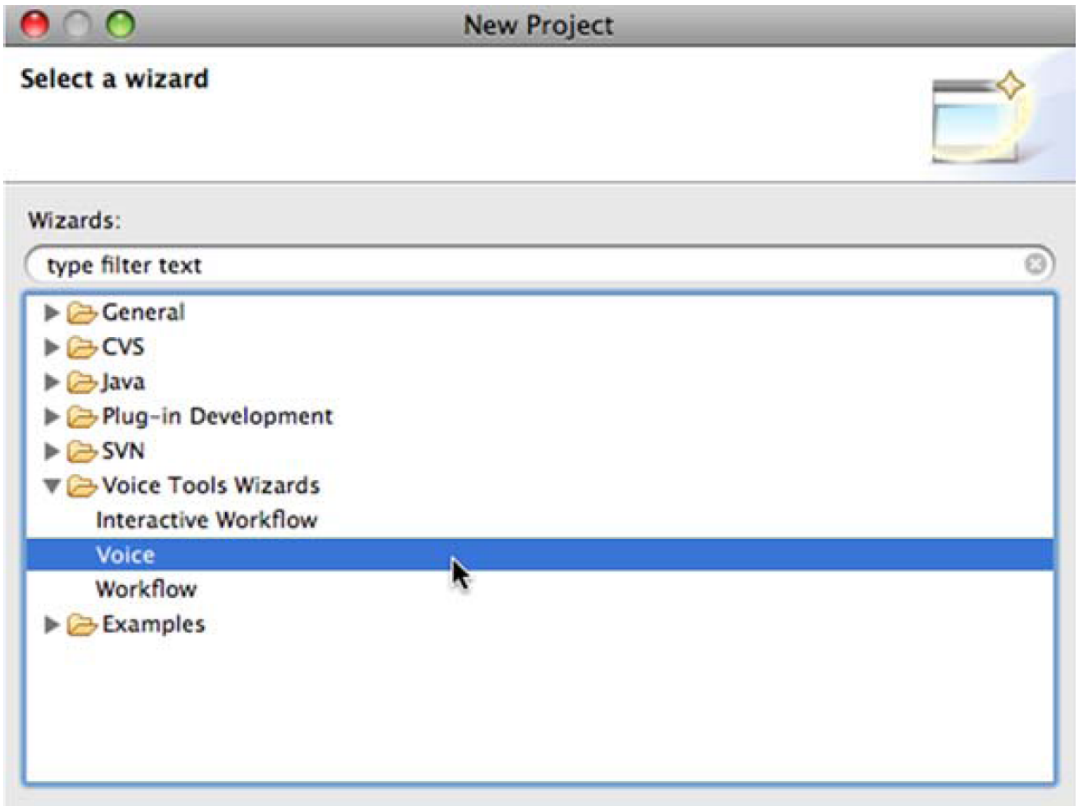
File → New → Project → Voice Tool Wizards → Voice

Alternatively, right click in the leftmost window tab of your Eclipse environment (this is called the Project Explorer) followed by:

New → Project → Voice Tool Wizards → Voice

Then click on "Next >" to bring up the "Voice Information" window. In the "Name:" field you can type in a name for this Voice project. Let's name this voice project "*Test_Voice*". After typing in "*Test_Voice*", select "Finish" to create the Voice project. Once created, you should see your new Voice project in the Project Explorer.

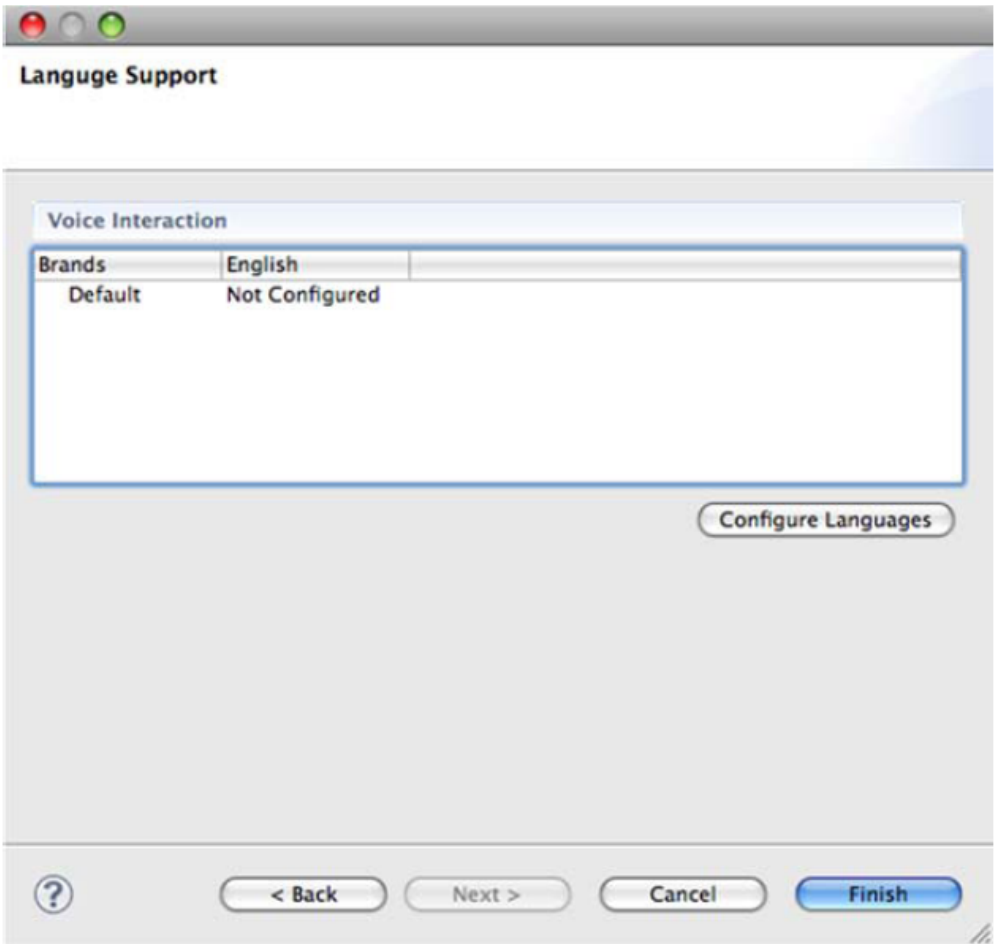




9. After the Voice project has been created, create the call flow project in a manner similar to what you did earlier for the Voice project:

File →New →Project → Voice Tool Wizards → Interactive Workflow.

Click on "Next >" to bring up the *Create Application* window. In the "Application Name:" field you can type in a name for this Workflow project. Let's name this Workflow "*Test_Workflow*". After typing in "*Test_Workflow*", click on "Next >" to bring up the *Branding* screen. Click "Next >" on the *Branding* screen, followed by "Next>" on the *Interaction Type Support* screen. Then you should arrive at the following screen:



10. Now associate or link your Workflow with your Voice. To do this, you'll need to click "Not Configured" under the *English* column and select the name of the new Voice project (*Test_Voice*) that you just created from the drop-down menu. Once this is done, click Finish. Now you should see your new Workflow project named *Test_Workflow* in the Project Explorer.

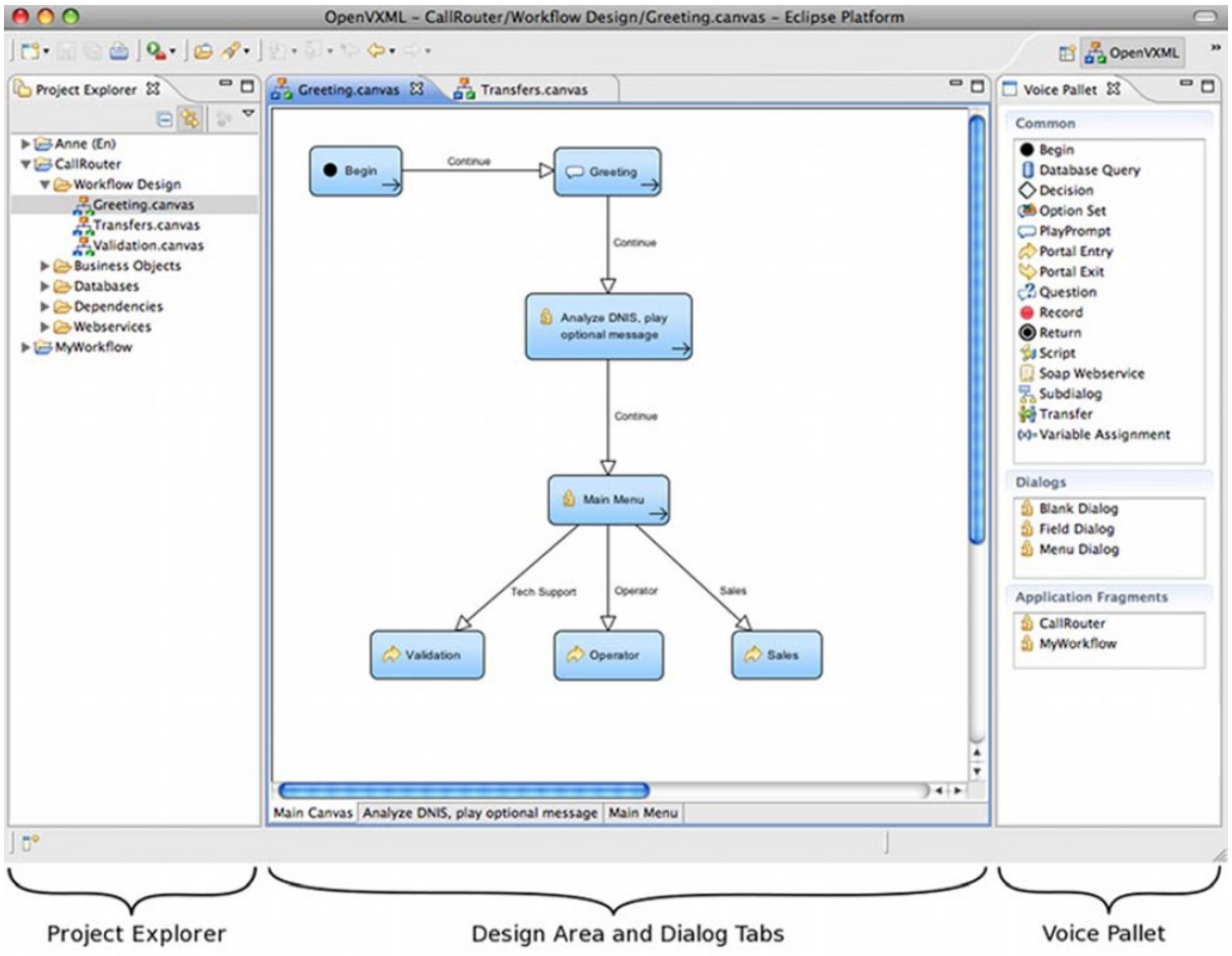
Step II: Creating a Hello World application workflow:

1. Clicking on the right arrow to the left of your new Test_Workflow project in the Project Explorer window will allow you to examine the files within that project. Navigate to:

Test_Workflow →Workflow Design →Main Canvas.canvas

Double click on Main Canvas.canvas to open it up in Eclipse. As the name suggests, this is the canvas on which you will be drawing flowchart-like representations of your desired call flow.

2. Your Eclipse environment should now look something like this (ignore the exact contents of the Project Explorer and the flowchart in the middle; you haven't created anything yet for your own application):



The "*Project Explorer*" (on the left) is where you should see the Voice and Workflow projects in your Eclipse workspace.

The *"Design Area"* (in the middle) shows the call flow of the currently selected canvas. Below that is another way to both indicate the current canvas and to select another canvas to view: the *"Canvas Tabs"*.

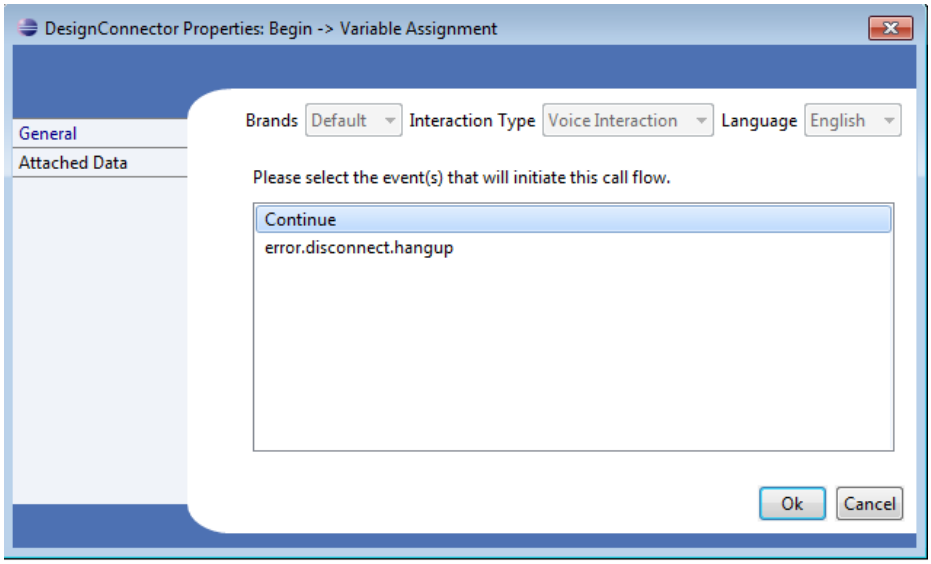
The *"Voice Pallet"* (on the right) displays the available building blocks for the application. This includes the primitive objects (prompts, decision blocks, database queries, etc.) as well as the more complex structures (dialogs, application fragments, and custom integrations).

Note that it's possible to drag-and-drop all of these windows in Eclipse to different locations on the screen and to resize and minimize them. Therefore, it's also possible to organize these three windows into a different layout, if desired.

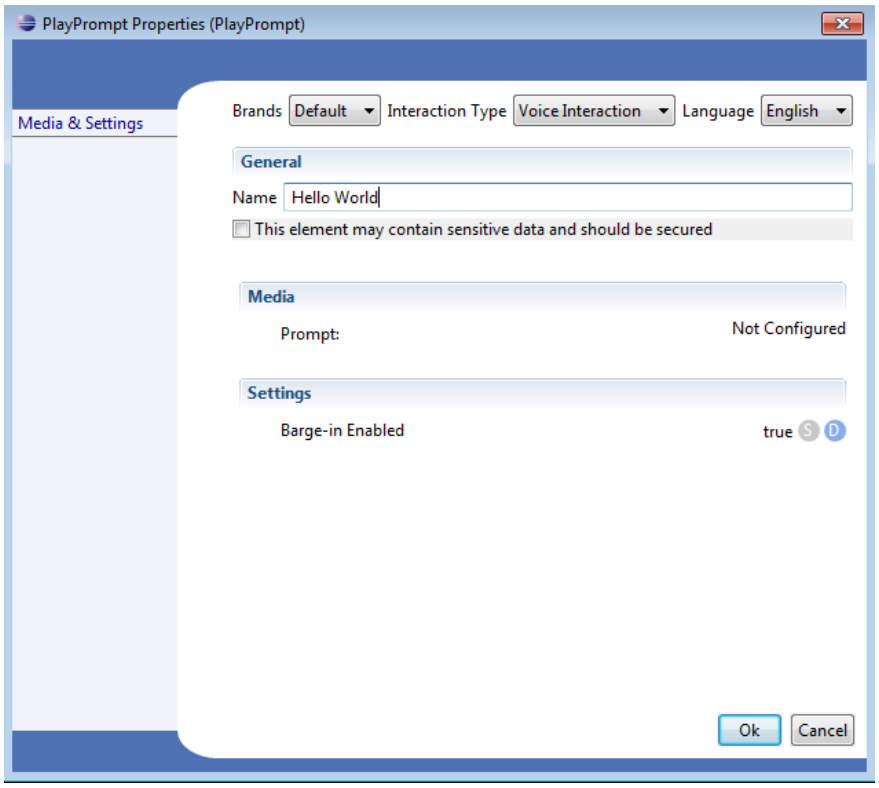
2. Now we will create a "Hello World" application, which will basically just greet the user with a machine-synthesized prompt "Hello World!".

3. Drag and drop the following 3 modules from the *Voice Pallet* onto your canvas (note that the Begin block might already be there on the canvas for you by default):

4. The idea is to connect the *Begin* block to the *PlayPrompt* block and in turn connect that to the *Return* block. To do this, on the arrow in the lower right of the *Begin* block and drag and drop it onto the *PlayPrompt block*. In the ensuing dialog box, select "Continue" and click "Ok". Then, follow the same procedure to connect the *PlayPrompt* block to the *Return* block.

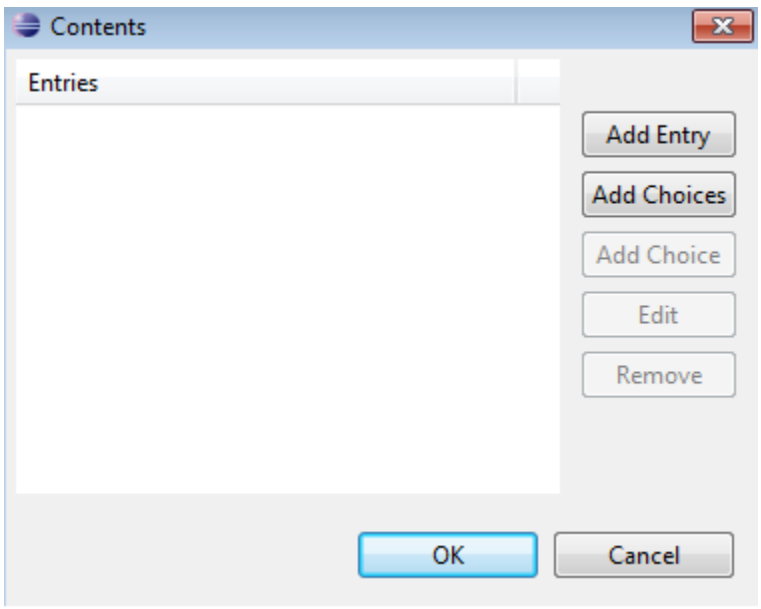


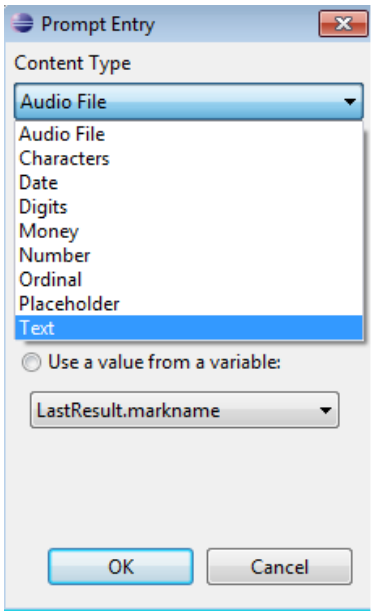
5. Now we just need to configure the *PlayPrompt* block for our prompt of choice, "Hello World". To do this, double-click on the *PlayPrompt* block. You should see something like this:



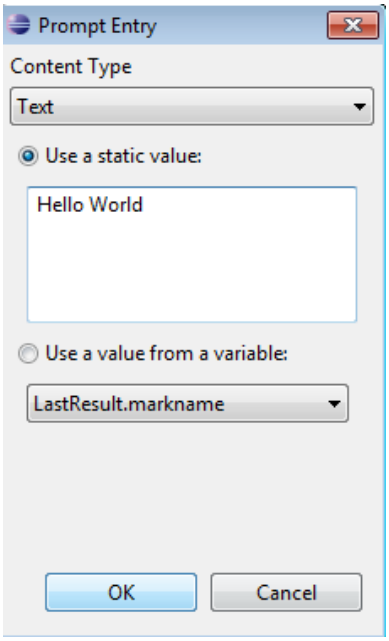
Enter “Hello World” into the *Name* field. This is just for bookkeeping purposes. In order to tell the spoken dialog system what to say, double-click on the text corresponding to *Prompt* under the *Media* tab (in my example, you would click on “Not Configured”).

Then click on “Add Entry”, and select “Text” from the “Content Type” dropdown menu.





In the field, “*Use a static value*”, enter the text “Hello World”. This is the input that the spoken dialog system will have the text-to-speech synthesizer generate. So in case you want the system to ask a user/test taker a question like “How are you?”, you would input that here (irrespective of what you enter into the *Name* field).



Click “Ok” three times and you are done!

6. Now we need to build the new voice project. To do this, right-click your Workflow (*Test_Workflow*) in the Project Explorer and click "Build Project".(Note that by default, Eclipse may set projects to build automatically, in which case you will not need to build the project every time you alter something. To make sure that this option is enabled , go to "Project" in the dropdown menu and make sure "Build Automatically" is selected).

7. Finally we just need to export the newly created workflow into a Web ARchive (WAR) application that can be read by the Voice Browser of a spoken dialog system. A voice browser does with voice/speech web pages (VoiceXML format) what a web browser like Firefox and Internet Explorer does with text web pages (HTML format). To do this, first save all your changes by navigating to:

File → Save All

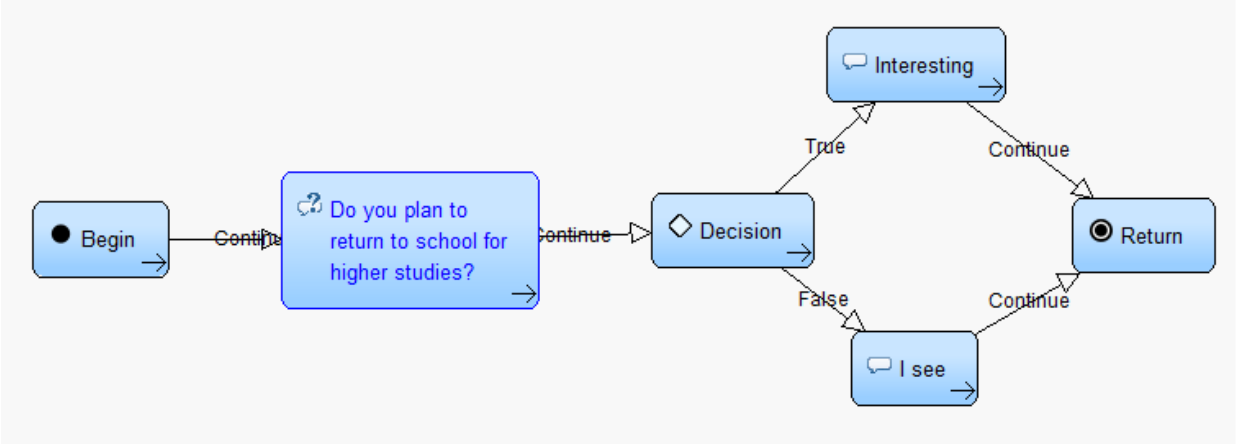
Then right-click your workflow (*Test_Workflow*) in the Project Explorer and navigate to:

Export → Voice Tools →Web Application

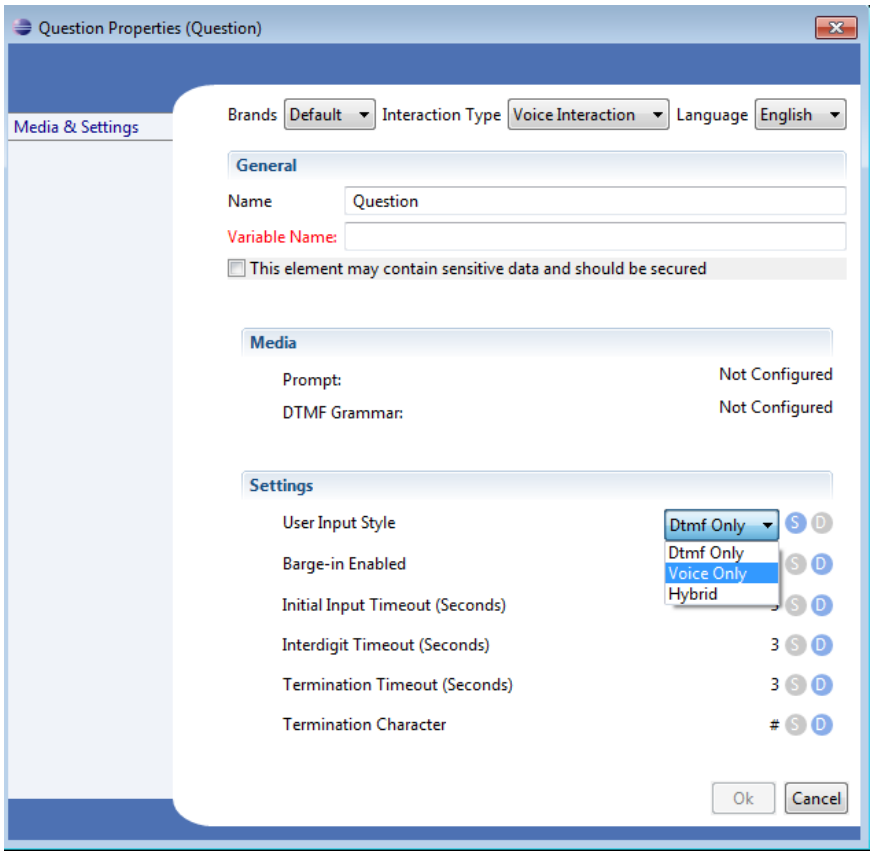
Click "Next >" to bring up the *Select an Archive File or Directory* screen. Now select the "Archive file:" radio button and enter the desired name of the WAR file (say something like "Test.war", for example) in the "Archive file:" field and click "Finish". By default, the WAR file will be located in your Eclipse directory (i.e., where the eclipse.exe executable file was unpacked). If all goes well, congratulations, you have created your first voice application, ready to be deployed on a spoken dialog system!

Step IIIA: Creating more advanced applications: Questions and Decisions

1. Now we will create an application where the spoken dialog system has to ask the user/test taker a particular yes/no question, and play back different prompts depending on the user's input. In our case we are going to assume that we have to design the app such that the SDS asks the user "Do you plan to return to school for higher education?". To give you a sneak peek, the call flow might look something like this:



2. To create this call flow, we are first going to invoke the question block. Create a new Voice and Interactive Workflow according to the instructions in Step 1. Then, open up the Main Canvas, and drag the "Question" option from the Voice Pallet and drop it onto the canvas and connect the Begin block to it by dragging the arrow from the Begin block (select "Continue" for "Please select the event(s) that will initiate this call flow" in the pop-up window). Double-clicking on it should allow you to see something like this:



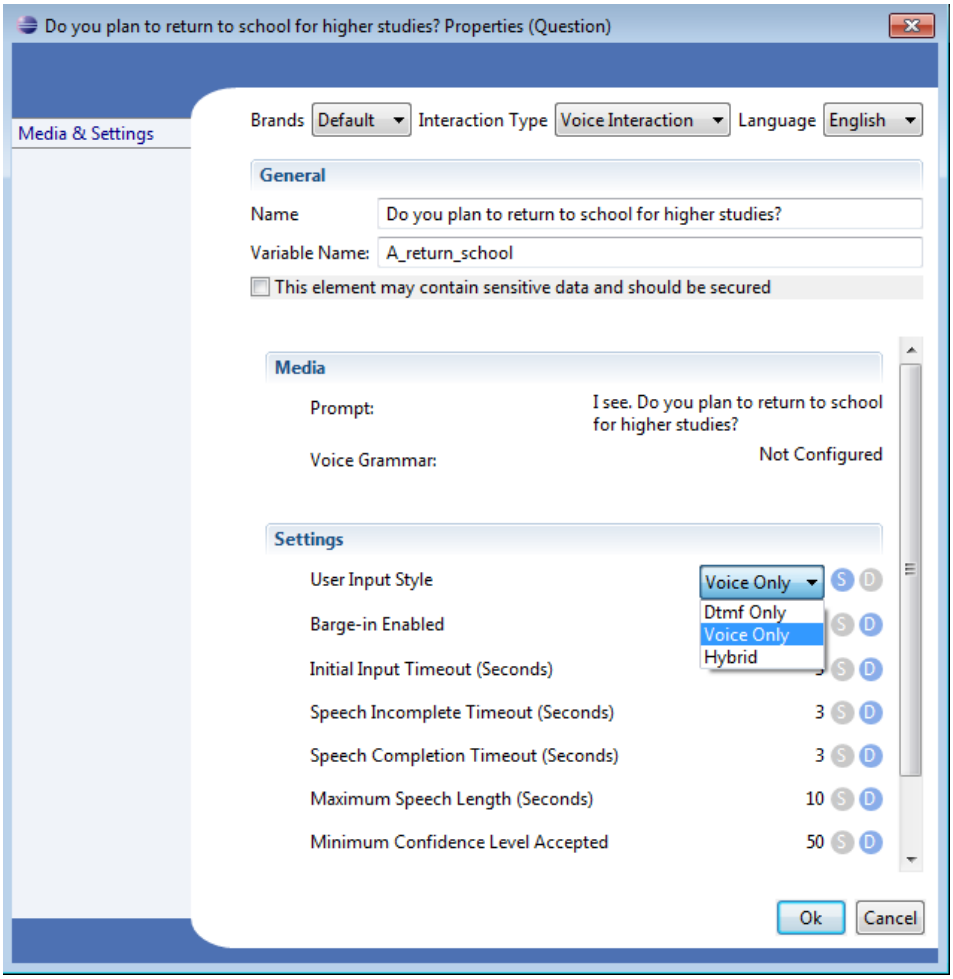
3. The "Name" field, as before, is used for bookkeeping purposes (so that you can look at a large call flow and figure out what that question is asking, so to speak). In designing large call flows with a lot of Question blocks, a good practice is to prefix the contents of the Name field with the question number, such as "Question1:<what is the question?>".

4. The "Variable Name" field tells the spoken dialog system the name of the variable that the speech recognition result will be returned in. I'd suggest naming this something like: "A_<question key words>" for consistency (an example would be one we have used throughout this tutorial, A_return_school.

5. The "Prompt" field contains the exact text of the question (could be same as the text in the "Name" field or different) that you'd like played to the receiver in order to elicit a response. You would set this in the same manner as you would for the "Play Prompt" block, as described in Step II of the tutorial.

6. This brings us to the "Grammar", which is one of the key parameters to set for any Question block. We will come back to this in just a minute.

First, under *Settings*, the first option you should see is "User Input Style". On the right hand side, you should see two letters, "S" and "D". You need to make sure you select "S" (for speech input). From the drop-down menu to the immediate left of the letters, select "Voice only". Now the "Grammar" or "DTMF Grammar" option in the Media section should have changed to "Voice Grammar". Now you are ready to specify your grammar file.



7. Now before we specify the grammar, let's generate it. A grammar is, simply put, a list of words and phrases that we'd like the automatic speech recognizer (or ASR) to recognize. Note that there are two main types of grammars we'll be working with: rule-based grammars, and statistical grammars. Though the latter offers more flexibility (and is something we'll eventually transition to in the long run), for now we will focus on one particular type of rule-based grammar, the JSGF (Java Speech Grammar Format). Rule-based grammars basically allow you to list out the words and phrases that you'd like the ASR to recognize. (It is important to note that in the most basic JSGF grammars, the ASR will *not* recognize anything outside this set). Here's an example of a simple JSGF grammar called "`yesno`" that just recognizes the words YES and NO.

```
#JSGF V1.0;
grammar yesno;
public <yesno> = yes | no;
```

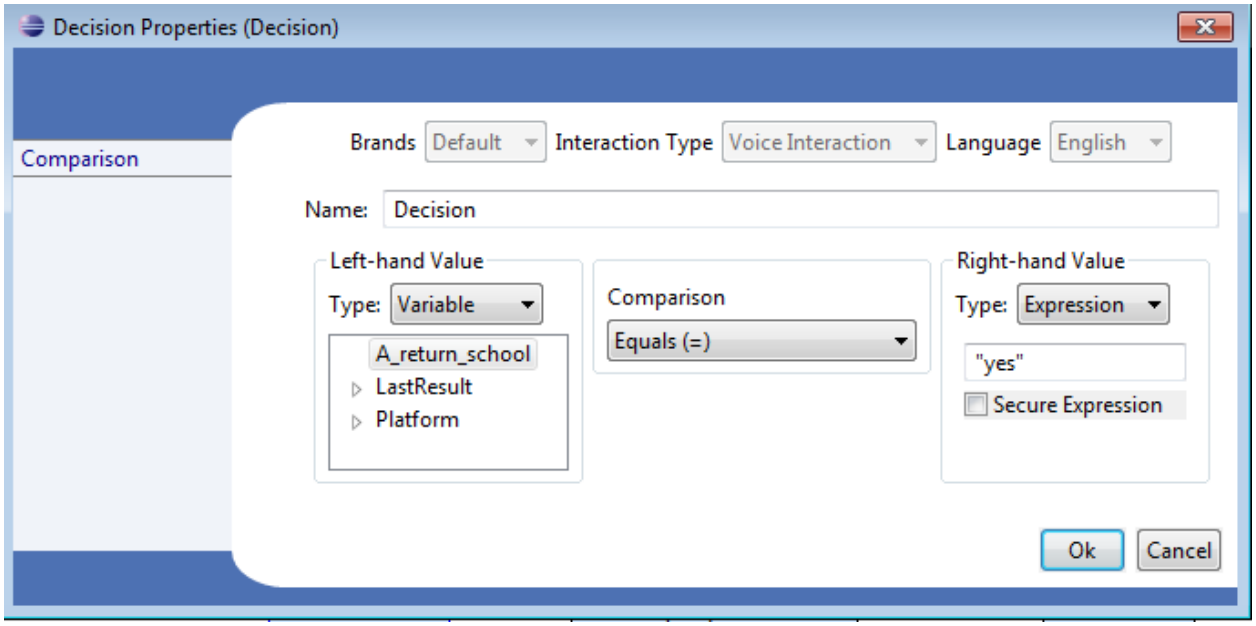
The only thing you need to change is on the third line; you can add more words/phrases by just writing them in and making sure that there is a pipe character ' | ' between each grammar entry.

Open a text editor like Wordpad or Notepad and paste these three lines into a new (empty) file. Save the file as "yesno.gram" *inside your Voice Project directory (that you initially created at the beginning when you set up the

location for the Eclipse workspace on your system)* under the "Media Libraries/Default" directory path (This is important: the application will find your grammar only if it inside this <Location of Voice Project>/Media Libraries/Default/> directory). If you're not sure where this is, you can find out the location of the Voice Project by selecting the Voice in the Project Explorer window and choosing File -> Properties. The system path for the Voice Project will be listed under Resource -> Location; this is the directory where the yesno.gram file should be saved.

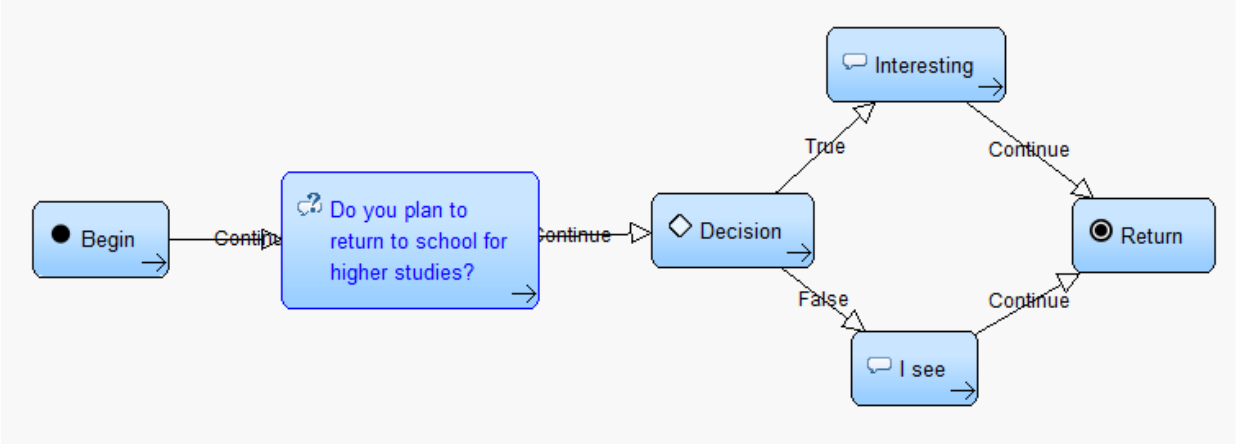
Now double-click on the "Not Configured" piece of text opposite the "Voice Grammar" option in the most screenshot. You will see a dropdown menu – select "GRXML grammar" and manually type in "yesno.gram" into the text box. Click "Ok" twice. You are done configuring the Question block.

8. Now we have to take different actions (play back different prompts) based on whether the ASR recognizes the user's voice response to the question you just created as "yes" or "no". For this we will use a "Decision" block. Drag-and-drop the corresponding icon from the Voice Pallet onto the Canvas and connect the Question block that you just created to it by dragging the arrow from the Question block on top of the Decision block and selecting "Continue" for "Please select the event(s) that will initiate this call flow" in the pop-up window. Double-clicking on the Decision block should give you something like this:



The block is pretty self-explanatory; this is where you set up the decision rule. For the left-hand value, select the variable name corresponding to the last question block that you created (in this running example, it is "A_return_school"; note that this variable name will only be available to be selected in the Decision block after you have connected the Question block to it), and for the right-hand value, make sure that "Type" is set to "Expression" (you have to set this appropriately depending on the previous question, but we will mostly deal with "Expressions") and enter:"yes"; (note the double quotes here are part of what you need to enter into the box) into the text box (the first word that you want it to recognize). Select the appropriate choice of comparison ("=" in our case). Click "Ok".

9. Now create 2 different "Play Prompt" blocks of your choice that play back *different* prompts, and connect the Decision block to both of them in turn. When you try to make a connection starting from a Decision block, you should see 2 main options (other than the hangup/disconnect option) – True and False – these are the paths that the call flow takes if the decision rule that you just wrote evaluates to True or False, respectively. Then, set the response that the system will provide to the user in each of the two cases by following the instructions for the Play Prompt block in Step II of the tutorial. In the case of the example below, I've chosen to play back "Interesting" to the user when the decision evaluates to True, and "I see" if it evaluates to False.



10. Finally connect both “Play Prompt” blocks to a “Return” block (again, select “Continue” in the pop-up window after connecting each Play Prompt block to the Return block) to complete and terminate the call-flow. You are done!

Step IIIB: Beyond grammars with just 2 options (Optional)

1. Now let's go one step further. There are obviously more ways to answer the question that we just designed than just "yes" or "no". So even if we incorporate all these other options into the grammar, the problem comes at the Decision block, where we have to specify *only one* expression for the block to compare against in order to evaluate to True. To allow us this additional functionality we are going to do the following:

For any affirmative answer, we will set a variable called "flag" to "yes" (corresponding to the semantic category, "yes"). For any other (or negative) answer, the "flag" value will be "no". This way, in the Decision block, we can set up a rule that compares the value of the "flag" variable to "yes" (or "no").

2. In order to achieve this functionality we are going to use a "Script" block from the Voice Pallet *in between* the Question and Decision blocks. Drag-and-drop this block from the Voice Pallet, remove the arrow connecting the Question and Decision blocks (by right-clicking and selecting Delete), connect the Question block to the Script block, and connect the Script block to the Decision block (select "Continue" for both of these connections). This Script block allows you to add custom Javascript code to the callflow (by just copying-and-pasting code into the textbox that you see when you double-click the Script block).

3. If you are comfortable with writing Javascript code, that's great and you can add code that achieves the functionality mentioned above. But if you are not, no worries, we've written some Python code that will generate *both* the script as well as a custom grammar corresponding to the Question block that just preceeds the Script block (the latter is important especially when there are many Questions in your callflow, each requiring its own grammar and script to make decisions based off of). All a designer will need to do is input a set of phrase—semantic category pairs into a TSV (tab-separated value) file that can be edited using Microsoft Excel. **I would strongly suggest that you name the TSV file with the same name as the name of the *return VariableName* specified for the corresponding *Question* block.** For example, entries in this file ("return_school.tsv", if we use our running example) would look like:

---

yes yes `(Tab space separating entries on each line)`

yes i do yes `("yes I do" corresponds to the semantic category "yes")`

no no

---

4. In order to run the Python code, you will simply need to run the executable file "ggs.exe" (stands for *generate grammar and script*; this is under the "dist" folder in the ZIP archive provided with this section of the tutorial). Make sure the "return_school.tsv" file is in the same folder (you can move it from the top-level directory in the .zip archive), and on correctly executing the "ggs.exe" executable file (see below), 2 separate files should be generated: i) a GRAM (grammar) file, the contents of which can be copied to your desired grammar file, and ii) a SCRIPT file, the contents of which can be copied as is into the script block. Note that all of these files will have different extensions, but the same file name (return_school in our running example). It is important to adhere to this convention for two reasons: (i) it lets us form a standard naming convention which will be useful when we are creating and debugging large call-flows, and (ii) the SCRIPT file that you create will be expecting as input the correct *VariableName* of the *Question* block preceding the *Script* block.

RUNNING THE EXECUTABLE: *By default, if you just run the executable as is without specifying the filename of the TSV file, the program will assume that the filename is "phrases.tsv"*. If you want to start from your own TSV file (*highly recommended*) that has a different filename (say "return_school.tsv"), then you will need to run the executable from the command line. To do so:

a) Bring up a terminal in Windows by going to the Start menu, and typing "cmd" into the text box located just above the Windows icon of the Start menu.

b) Navigate to the folder where the "dist" folder from the .zip archive was unpacked

(Example: `cd C:\ETS\Eclipse_Workspace\dist`)

c) Make sure your file "return_school.tsv" is in the same "dist" folder and type:

`ggs.exe return_school` (Note NO extension ".tsv"!)

If ggs.exe was executed successfully, you should see something like this in your terminal window:

```
Processed the following entries:
no : no
yes : yes
yes i do : yes
```

Furthermore, the following two files should be created in the dist/ directory: return_school.gram and return_school.script.

5. If for some reason the executable (step 4) does not work, then you will need to do the following (ELSE IGNORE THIS STEP AND PROCEED TO STEP 6):

a) Download Python from https://www.python.org/downloads/. Choose the Python 2.7 version for now.

b) Install it. Note down the installation folder.

c) Once Python is installed, bring up a terminal in Windows by going to the Start menu, and typing "cmd" into the text box located just above the Windows icon of the Start menu. At the terminal, type the following command:

`set PATH=%PATH%;C:\Python27` (replace `C:\Python27` with the folder location where you installed Python)

This basically tells the Windows system where to look for the Python executable.

d) Navigate to the folder with the actual Python code (ggs.py) and simply execute the command:

python ggs.py <optional file name without extension>

Make sure that the TSV file (return_school.tsv in our example) is in the same folder, and the 2 output files – GRAM and SCRIPT – should be generated. Now copy these 2 files into the "Default" directory under your "Voice" directory (the same location where you created your grammar file earlier).

6. Now we need to change the grammar that was previously configured in our Question block to the one that was just generated using the Python executable/script (return_school.gram). To do this, as before, double-click the Question block and double-click on the text opposite "Voice Grammar" and enter the name of the GRAM file that was just created into the text box. Click "Ok" twice to accept these options.

Now we also need to add in the script that we generated into the callflow. Double-click on the Script block in the conversation flow and copy-and-paste the contents of the return_school.script file into the "Script" box and click "OK". Once the script block has been added and contents copied successfully, we should be done. Congratulations!

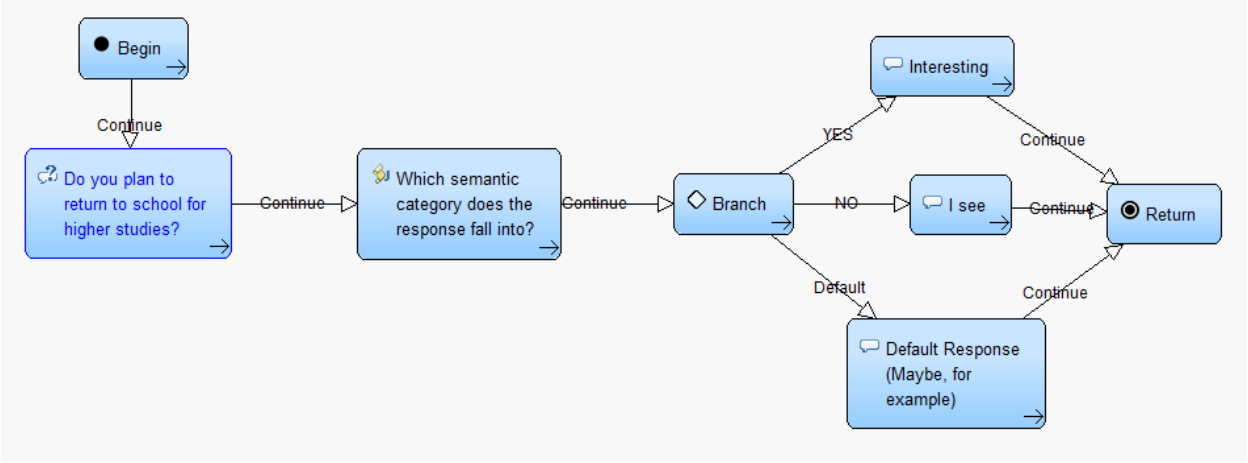
PLEASE READ: *An important note about standardization in designing Questions:*

As in the previous case, you will run an updated script/executable ggs.exe/ggs.py that will generate the script that you will need. For purposes of standardization, I would strongly suggest that you follow the following standard naming conventions:

- When you are creating the question, first choose an appropriate identifier for that question. (Ex: "`return_school`")
- Choose the variable name as "`A_<question_identifier>`" – the 'A' here stands for 'answer'. (Ex: "`A_return_school`")
- Specify the list of phrases and semantic categories in a Tab Separated Value (TSV) file called `<question_identifer>.tsv` (Ex: `return_school.tsv`)
- When specifying rules in a decision block, the variable name that you are comparing each semantic category to *must* be of the format "`Variables.SC_<question_identifer>`" – 'SC' stands for 'semantic category' – (Ex. "`Variables.SC_return_school`")

Step IV: Decisions with multiple options (branching) (Optional)

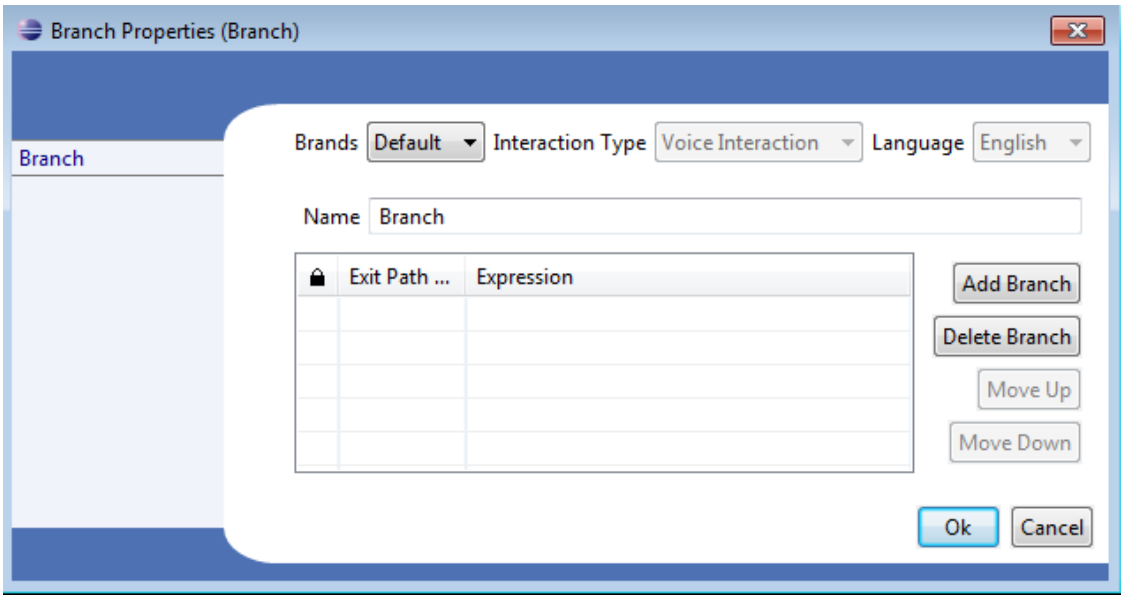
1. Now that we've created applications configured with question blocks that accept answers in one of two pre-specified categories (binary decisions), we will proceed to the next logical step: multiple decision paths (corresponding to multiple semantic categories of interest). In this case, the main change from the previous workflow will be that our Decision block will be replaced by a *Branch* block. Here's what the workflow might look like (notice that I have named my script block as "Which semantic category does the response fall into?"):



Notice that we have the same Yes and No transitions from the *Branch* block that we had earlier (set up as True and False blocks), but now we also have an additional "Default" transition. The flow follows this latter transition path when none of the conditions corresponding to the other paths are satisfied (see below for instructions on how to specify these conditions).

2. Keeping the same workflow as before, delete the Decision block and all links associated with it, and drag a Branch block onto the canvas in its place(or if you prefer to create a completely new workflow, redo what you did earlier, but now use a Branch block instead of a Decision block).

Double-clicking on the Branch block should give you something that looks like this:



3. Now you can add decision (comparison rules) for each semantic category that you've previously defined in the file <question_identifier>.tsv (for example, return_school.tsv in Step IIIB of the tutorial). To add a decision rule, click "Add Branch". You will need to provide 2 entries:

- The Exit Path Name is simply the name of the path that will be displayed on your workflow canvas. This is for bookkeeping purposes only.

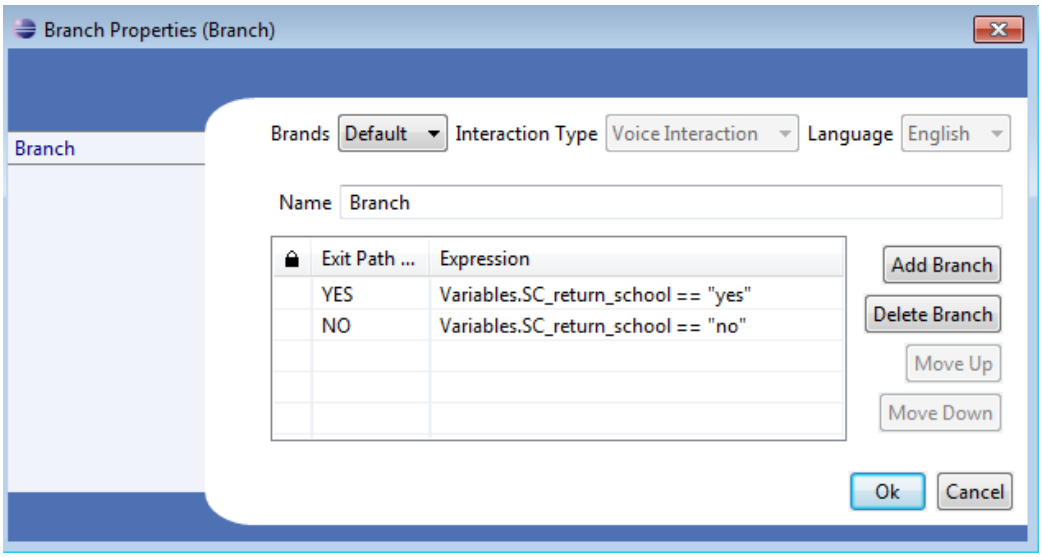
- The *Expression* is the key thing you need to focus on. This field requires that decision rules be written in JAVASCRIPT CODE format, which will essentially be of the following format:

`Variables.SC_<question_identifier> == "<semantic_category>"` [Note the 2 "equals to" ("=") symbols and the double quotes around the semantic category]

So, for example, if we are comparing a variable corresponding to the question_identifier "return_school", and we'd like to specify a branch for the "yes" semantic category, the syntax would look like:

`Variables.SC_return_school == "yes"`

Once you are done, the branch block specification could look like this:



As mentioned earlier, if neither of these conditions are satisfied, the "Default" transition path is chosen.

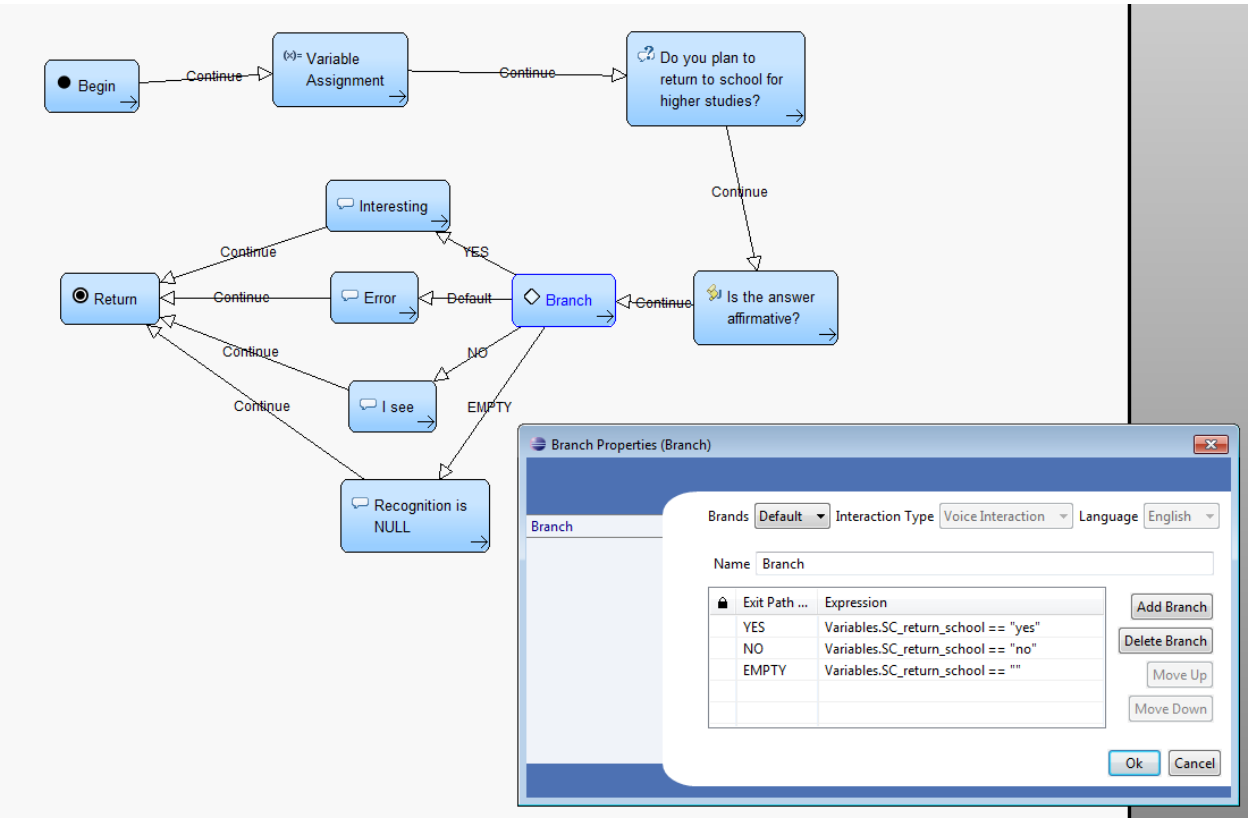
4. Now create Prompt blocks corresponding to each of the transition paths (including one extra for the default path) that you just created and connect each of them to the Branch block in turn. When you try and connect them, a window should pop-up that asks you to "Please select the event(s) that will initiate this call flow": followed by a list of available unconnected transition paths. Once you are done connecting the blocks as desired, you are ready to export your workflow to a Web Archive (WAR) application for deployment on a spoken dialog system.

5. Currently the HALEF system has been configured such that whenever the speech recognizer returns a NULL result (i.e., it did not understand or recognize the input utterance as being in grammar), the value of the corresponding answer variable of that question is set to an empty string (''). The ggs.py script checks for this as well, and copies over an empty string value into Variables.SC_<question_identifier>.

This functionality allows you to design specific actions of your choice in the event that the recognizer did not catch what was being said (for example, you could choose to play a prompt of "Sorry, I did not catch that", followed by a redirect to the original question again). So in this case, you would add an additional line when configuring the "Branch Properties" as shown in the last figure as:

```
Variables.SC_return_school == ""
```

For an example of what a workflow with this modification might look like, see the following figure:



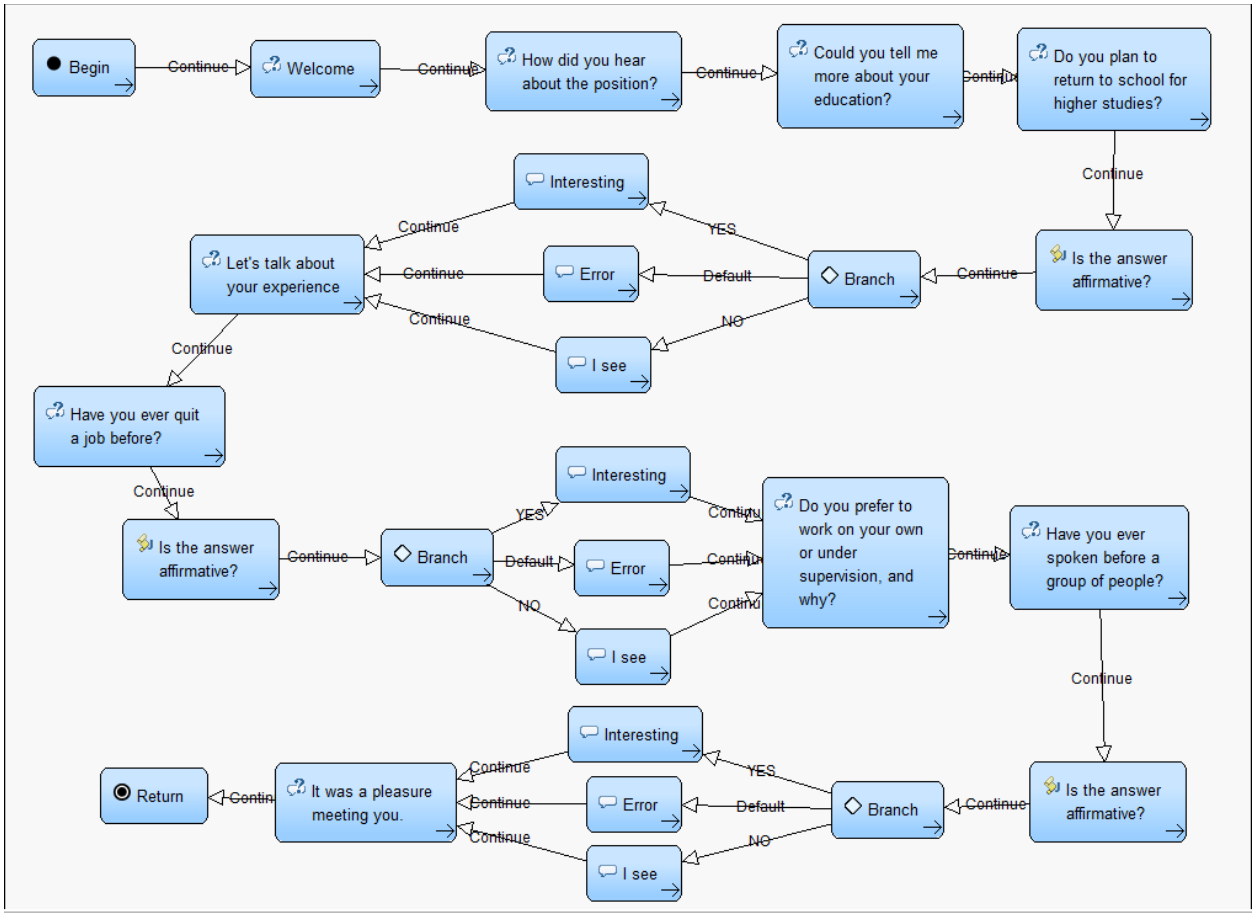
Note: This is still different from the "Default" branch (which, again, needn't be specified in the branch properties). A quick note about the current behavior of the default option:

Expected behavior: This is the branch the control flow goes into in case none of the other conditions are satisfied *no matter what*.

Current behavior: This is the branch the control flow goes into if the Variables.SC_<question_identifier> is set to any semantic category value other than (i) an empty string or (ii) any semantic category specified in the input TSV file *for which a branch condition has not been specified*. This is because currently the system is set up only to recognize exactly the words/phrases that are specified in the grammar file (we are actively working to extend this functionality). What condition (ii) above means is that the control flow will enter the default branch only if the ASR recognizes a word/phrase that is both in grammar AND belongs to a semantic category for which a branching rule hasn't been explicity written yet.

Step V :Creating a complete dialog workflow for a test interview item (Optional)

Now might be a good time to try creating a complete dialog workflow based on the simple interview item that was designed earlier. Here's what it might look like (of course, feel free to customize it as you see fit):



Step VI: Portals – Extending workflows to span multiple canvases (Optional)

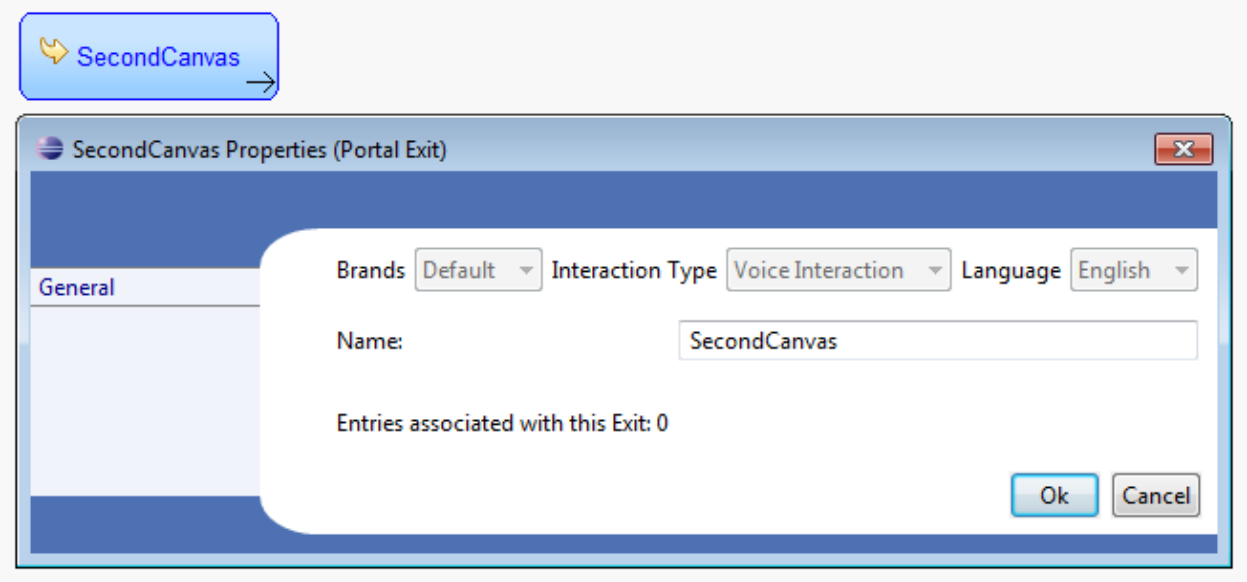
Using multiple canvases is a great way to tokenize a call flow into logical units, reducing a monolithic application into more manageable pieces. Portals are a convenient way to provide connections to other canvases in the Workflow, and new Begin blocks open up new access points to other Workflows.

To add a new design canvas to a Workflow (and configure a portal between the new and existing canvas):

1. Right-click on the *"Workflow Design"* folder for the appropriate project in the Project Explorer and select: *"New" -> "Design Document"* from the context menu (if "Design Document" does not appear in this menu, select "New" -> "Other", then select "Voice Tools Wizard" -> "Design Document" and click "Next"). This will open the new design document wizard.

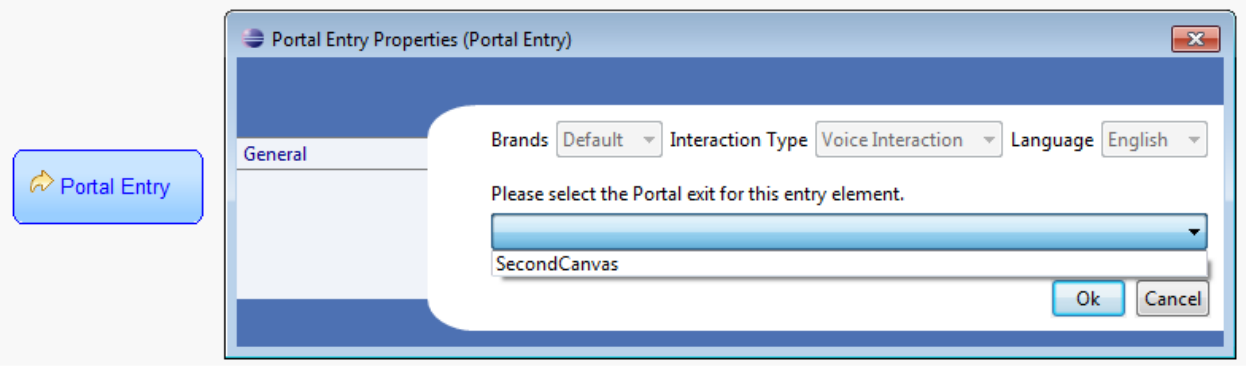
2. Enter a name for the new design canvas in the input box (for example, "SecondCanvas". **Note:** This name must be unique amongst the existing design canvases in the application. An error message will be displayed if the name already exists.

3. To finalize the creation of the new design canvas, click the Finish button. The canvas will be added to the application along with a new *Portal Exit* called "SecondCanvas" on the new canvas, ready for use.



4. The *"Portal Exit"* module is half of the link between two canvases; specifically, it is used on the destination canvas where it marks the exit of the portal. A *Portal Exit* must be created before their corresponding module, the *Portal Entry*, can be configured. A *Portal Exit* can have more than one *Portal Entry*, but a *Portal Entry* can only have one *Portal Exit*. Double-clicking on a Portal Exit in the new canvas ("SecondCanvas" in the above example) will allow you to configure the name and view how many Portal Entries are associated with this Exit.

5. The next step is to configure the Portal Entry in your original workspace. Drag and drop the "*Portal Entry*" icon from the Voice Pallet in your original workspace (say Main Canvas.canvas). To use a Portal Entry:
- Make sure a Portal Exit on the destination canvas has been created; know the name of this Exit.
- Go to the originating canvas; select the Portal Entry module from the Voice Pallet; drag this on to the canvas and then drop it. The new module should appear on the canvas.
- Double-click the new module to see the properties pop-up box:



From the dropdown menu, select the "SecondCanvas" exit (or whatever the name of the portal exit/new canvas was that you created earlier). You are done!

Step VII: Importing existing OpenVXML workflows or WAR files into the Eclipse workspace (Optional)

Eclipse allows you to import existing OpenVXML workflow projects (or WAR files) into the current workspace, enabling you to work on a project that someone else has created or worked on, or potentially work on something you created earlier and them subsequently deleted from your workspace (not your computer). (How to export a workflow has already been covered earlier on page 8 of this tutorial).

Before importing a project into your workspace, make sure that a project of the same name does not *already* exist in your workspace. Note that need *not* necessarily be the same as the name of the WAR file that you are importing (it is the name of the workflow that was used to create the WAR file in the first place → so for example, if the project was exported to 7700.war from the workflow "Deploy_Workflow", then you'll have to make sure that you delete the *latter* (or workflows with that name) from your workspace.

To import, go to File → Import… → General → Existing Projects into Workspace. Click on Next>.

You will see a window with a couple of text boxes, one ("*Select root directory*") allowing you to enter either the root directory where the Eclipse project is stored, or alternatively, a WAR file ("*Select archive file*"). Once you select either one of the above, the corresponding Voice and Workflow contained within your selection should populate in the text window below. Check the appropriate boxes and also check the option for "Copy projects into workspace". Click Finish. You are done!

Note: If you want to work on another version of the same call-flow (say, "Deploy_Workflow") without deleting your own version/changes, you can work on both versions by simply changing the workspace for each of those versions. You can do this by navigating to File → Switch Workspace. If you can't find your desired workspace from the options shown (which will be the case if you haven't switched a workspace before), then click on Other. This will allow you to select a new folder on disk to establish a new workspace in.

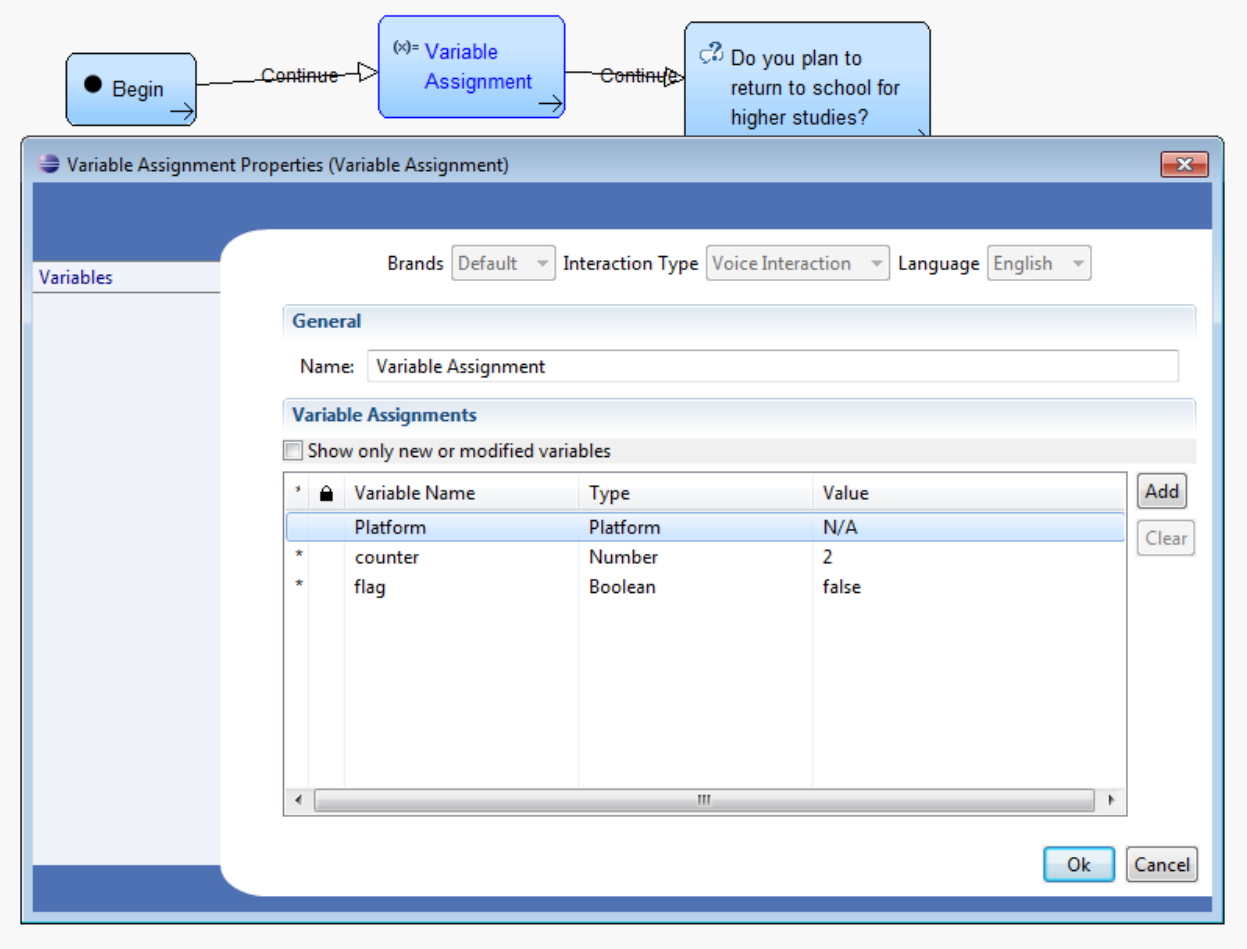
Step VIII: Implementing counters to loop through dialog states a given number of times (Optional)

One way to implement a counter and incorporate it into your workflow is to initialize a "counter" variable to the number of times you want to count up to (using a 'Variable Assignment' block) and then add 3-4 lines of code to a 'Script' block that (i) checks whether the value of "counter" is 0, else (ii) progressively decrements the "counter" by 1 every time it is executed.

Alternatively, you can implement a counter by initializing the counter variable to 0 and adding code to a 'Script' block that (i) checks whether the value of counter is N (where N is the number of times you want to loop through a question or dialog state), else (ii) *increments* the counter by 1 each time it is executed.

1. The following example demonstrates the first case. Here the 'Variable Assignment' block sets a "counter" variable, which is a Number of value "2" (assuming that we want to loop through a question/dialog state two times).

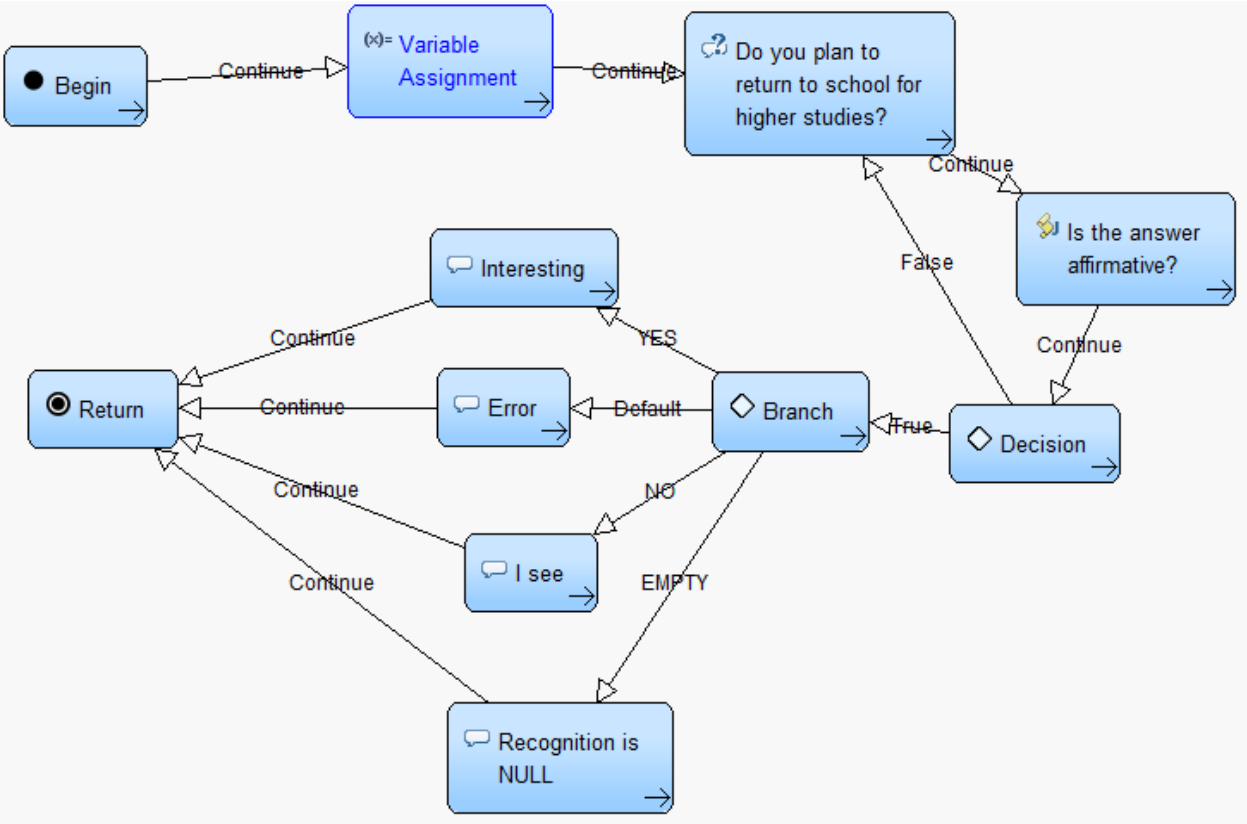
I would suggest standardizing the name of the counter variable to counter_<question_identifier>. For example, if the question identifier (also name of TSV file defined earlier) is "return_school", the corresponding counter variable could be named "counter_return_school".



Extra Tip: Note that in this case I have also set a "flag" variable (which can take values "true" or "false"). You can use this variable to check whether a given dialog state/question has been visited during the call, by setting this flag

to true in the script block corresponding to that question and checking whether the flag is true in a decision block as shown below.

The following example workflow loops through the question "do you plan to return to school" 2 times, irrespective of what was said the first time.



2. Once we've initialized the counter, the next step is to add a small code snippet to the "Script" block to decrement the counter variable. This could look like this:

```
//Check if counter is greater than 0, and decrement counter.
if(Variables.counter_<question_identifer> >0)
{
  Variables.counter_<question_identifer> = Variables.counter_<question_identifer>-1;
}
```

where the question_identifier is replaced accordingly.

3. The next step is to add another simple 'Decision' block after the script block that checks whether the value of the counter is zero or not. The implementation of a 'Decision' block has been described earlier in Section IIIA (pg. 12).

As mentioned earlier, you can implement "flag" variables to check whether a particular dialog state has been visited in a similar manner. Though this is but one way that we can implement counters and flags, this should be sufficient to give you an idea of the logic required for these kinds of applications.

Staging workflows on the HALEF Spoken Dialog System

One interacts with the HALEF (Help Assistant – Language-Enabled and Free) spoken dialog system (SDS) by calling the number +1 (206) 309-5423. The system will then prompt you to enter an extension of your choice. Each designer should have been assigned a unique four-digit extension to HALEF, on which you can stage your own callflow applications. Dialing this extension should then tell HALEF to execute that particular pre-staged application. In order to stage the callflows (or more precisely, the WAR files exported from your workflows) on the HALEF spoken dialog system, please follow the steps listed below:

1. In the Project Explorer, make a copy of the completed workflow that you'd like to deploy on HALEF (this can be done by selecting the workflow and doing a Ctrl+C and Ctrl+V (or copy-and-paste) in Windows. Rename the newly-created workflow as "**Deploy_Workflow**". If you already have a folder thus named, please delete the workflow from Eclipse (if you get an error message, just continue), *exit Eclipse*, and then delete the folder (if it exists) as well as the copy of the folder stored in the following path:

`<Eclipse Workspace location>\.metadata\.plugins\org.eclipse.core.resources\.projects`

"**Deploy_Workflow**" will be the name of the workflow that you will use to export all your designed workflows to Web Archive (or WAR) files. (This allows us to standardize the deployment of applications on HALEF with minimal modification of configuration files on the HALEF servers.)

2. Make sure that you have export your workflow (now in Deploy_Workflow) as a Web ARchive (or WAR) file (the procedure for doing this was described in Step II of the tutorial) *and that this file is named after the extension that was assigned to you, e.g.* "`<extension_number>.war`."

3. Email Zhou, so Zhou (zhouyu@cs.cmu.edu) will upload it to the server and deploy it.

4. Now you can call +1 (206) 309-5423, using Google Voice. Just open google voice and dial +1 (206) 309-5423, when asked to enter extension, enter the extension you are assigned and you will be able to talk to the dialog system you created.