SCLM Suite Demo
Demo Script

Review Project Definition

We will have a look at the project definition used to create the IBMDEMO SCLM project and the macros that it contains.

1. Using ISPF 3.4 go into edit on the IBMDEMO project definition source dataset. This is where the characteristics of the SCLM project, such as hierarchy, dataset types and language types, are defined.
2. Member CNTLPDEF contains the project definition along with the JCL required to assemble it.
3. We can see from the top of the job that there are two steps, an assembly followed by a link-edit. SCLM expects the project definitions to be in a dataset called ***project***.PROJDEFS.LOAD. Even though this is a standard Assembler assembly job the SCLM macro's bare little resemblance to standard assembler. They are a macro language which is fairly easy to understand.
4. Scrolling down we now look at the actual project definition control statements. The first statement the FLMABEG control contains the name of the project we are defining.
5. The FLMAGRP macro defines the authorisation codes that will be used. Authorisation codes are used to control parallel development.
6. We then see the FLMTYPE macros. These define the different dataset types that you will have in your SCLM project. We can see from the list here that there are both conventional dataset types such as COBOL and JCL as well as e-business types such as JAVA and DOC. The latter e-business types are used by Cloud 9.
7. We then see the definitions for our hierarchy. Our project hierarchy consists of three levels DEV, QA and REL. With DEV being promoted to QA and QA being promoted to REL. The AC parameter is the authorisation code. We can see that we are using a single authorisation code so no parallel development will take place in this sample project. KEY=Y specifies that these are all Key groups. This means that when a promotion takes place from one group to the next the code being promoted is purged from the group it is coming from.
8. We now see the FLMCNTRL macro which tells us some control information about the project such as
    a. The name of the account and versioning files
    b. The names and locations of any user exits being used. In this case we can see the Breeze userexits being called that control package approval.
9. The set of definition macro's are related to versioning and define which types are going to be versioned by SCLM.
10. We now have the language translators. This are macro's that define how a particular language type is going to be compiled or processed. The FLM@COB2 translator for instance will be the compile process for a COBOL II program. The FLM@L370 is the link-edit process. You can equate these translators to jobs you would run that perform the same function. What they also do though is update important accounting information required by SCLM to keep control of the members within the project. The translators are generally kept in separate members and are COPYed into the main project definition. IBM supplies a large number of sample project

definitions with the product for most of the main language types. Also you will find on the forums any number of user willing to share their more obscure translators.

11. The final part of the job is the input into the link-edit. The primary project definition always has a load module name of the name you gave the project in the FLMABEG macro. It is also possible to have alternate project definitions. For instance an alternate project definition may use slightly different language translators, for example: different debugging options that you don’t want shipped with your final product. The developer can specify a different alternate project to pick these up. Or you may have an alternate project defined that uses a slightly different project hierarchy for emergency promotion processes. These alternates are defined in the same way as the primary project definition and have all the same macro’s defining the project definition. How they differ is the FLMABEG still specifies the primary project name, but when you run the link-edit step the load module will have a different name.
12. We now browse the project definition load module to show the project definition after it has been link-edited.

SCLM Suite Demo
Demo Script

VA Java support

A new addition as part of the Cloud 9 product is VA Java support. This will allow you store your Java source and resources on the mainframe along with your conventional code types such as COBOL. Developers will be allowed to check the module out from the mainframe and work on it in the VA Java IDE and once finish check it back into the mainframe. (At the time of writing this is still in Beta test and should be available as a Cloud 9 PTF in 4Q 2002.)

Add VA Java project to version control

1. What we see here is the VA Java IDE with a number of projects in our workbench. The project we are going to work with is the Petdemo project.
2. As a developer we have designed and coded our project in VA Java and we are now ready to have the project added to our SCM system so that controlled updates can take place. So the first thing we must do is, using the standard interface supplied with VA Java, add the project to version control.
3. By right clicking on the project we get the context menu with the options we are permitted to perform against the project. From here we select ***Tools*** and then ***External Version Control*** followed by ***Add to Version Control***.
4. The Add to Version Control dialog box appears where we select our source handler. Cloud 9 has utilised the Microsoft SCCI SCM handler. We select that.
5. We then select the system we are going to use as our SCM, in this case Cloud 9 is the only option available to us. So we select that.
6. Next we are shown a dialog box where we must select a local directory. This is a working directory that is used in the transfer of code to and from the mainframe.
7. We must now enter some information in order to connect to the mainframe. Our userid and password are those that we use to log onto the MVS system where our SCLM projects reside. The IP address and port are those that the Cloud9 HTTP server is using in order to communicate with the mainframe. For this particular VA Java project the IP address and port will be remembered for subsequent activities, such as Checking In/Out, but we will need to enter our userid and password each time we need to connect.
8. The progress box appears and tells us what it is doing. At this point Cloud 9 is determining if the project already exists under version control. In our demo it is a new project so Cloud 9 is building a list of all the SCLM projects available to give us a selection list.
   Note : One of the points to come out of Beta testing has been that this process takes some time so it would be better to give the user the opportunity to enter the SCLM project name if they know it to save Cloud 9 going through this list building process.
9. Cloud 9 gives us a list of SCLM projects to choose from. In our demo we are going to select the IBMDEMO SCLM project that we reviewed previously. Cloud 9 will then go and retrieve the project definition information for this project, such as groups, types and languages.
10. The first thing that Cloud 9 is doing is building a project manifest for this VA Java project. The manifest will contain a list of all the Java source and resources that make up the project.

Cloud 9 will store this information in an "architecture definition" or ARCHDEF. This is an SCLM control member that will control the building and promotion of this VA Java project within SCLM. If there are any mainframe components required for this application they can be added at a later time to the ARCHDEF member. We select our type as PACKAGES. This is an SCLM controlled PDS that will contain a member that contains the manifest. The language type is left as ARCHDEF as this is the standard SCLM language translator for architecture definitions.

11. Again the progress dialog box tells us what it is doing.
12. We are shown the Action details dialog box where we can get to set some parameters. The authorisation code we wish to select, in our case this is just DEV plus a change code if we wish to enter it. For the purpose of the demo this is left blank.
13. We click OK to continue and the progress box pops up again telling us what is going on, in this case Cloud 9 is starting the process of adding this project to SCLM.
14. The Unable to Detect Files dialog box appears giving us a number of options. As this project is new to version control we would not expect there to be any members on the mainframe so we click the continue button.
15. The list displayed lets us select the individual Java source members that we wish to add. In our case we want to add all the source to version control. We select all types and click next.
16. The next list displayed is a list of the resources attached to this project. In our case we have number of GIF files. Once again we select all of them and click next.
17. The next pane displayed gives us the opportunity to add a path name to the objects as they are stored in SCLM. This path name could be required in deployment to have a particular directory structure. Again in our case we leave this screen as defaulted.
18. Again Cloud 9 tells us what is going on and shows us a list of the Java source members that will be added to SCLM. We click OK
19. The action details dialog is again displayed for the Java source members and once again we leave as defaulted and click OK.
20. We must now give SCLM some information about the files we are adding. SCLM needs to know the dataset type where the files need to added as well as the language type. This will be required later when be "Build" the objects. In both cases here the type and the language are both "Java". We click OK to continue.
21. Cloud 9 again tells us what is going on. First the files need to exported from VA Java to the local directory we specified earlier. Then they are transferred over to the mainframe and added to SCLM and migrated into the development level.
22. Once the processing has finished for the Java source members the resources are then processed. Cloud 9 shows us the list of .GIF files we selected earlier. We select OK.
23. We go through the same process that we did with the Java source members. Again we must tell SCLM how to process the .GIF files. In this case they are stored in the GRAPHICS library type with a language of EBIZBIN.
24. We are then shown progress information as the resources are added to SCLM. Cloud 9 tells us it has finished adding the files to SCLM so we click OK to continue.

SCLM Suite Demo
Demo Script

Test the dialog in VA Java

1. We now want to test the project to show an error that needs to be corrected. In the VA Java IDE we right click on the project to get the context menu and select ***Run*** and then ***Run main.***

2. The Petdemo project starts up.
3. We select a number of creatures we wish to purchase and once finished we click on the Check Out button.
4. The order confirmation box appears and we notice that a mistake has been made in the pickup address. The state has been made NT instead of IL.
5. So we close down the application and return to the VA Java IDE in order to make the code correction.

Correct the code in VA Java

1. The first thing we must do is check the module out from SCLM on the mainframe. We select the main module, indicated by the module with the little man next to it, an right click on it to display the context menu. We then select ***Tools***, ***External Version Control*** and ***Check out***.
2. We get some progress information that tells us the module name in SCLM that will be checked out. We click OK to continue. Cloud 9 then continues and after displaying the action dialog box, goes off to get the source from SCLM on the mainframe.
3. We then get a dialog box telling us that this module is currently checked out by another user and we therefore cannot check it out.
4. But what we can do is send Roy an e-mail to let him know that we need to do some work on this module. So we click the E-mail user button and that takes us directly to our mail system to write a note to the user.
5. Once Roy has received our e-mail and finished up with his changes we can then re-attempt to check the module out.
6. This time Cloud 9 is successful in retrieving the member and checks the module out to our userid. A small padlock appears next to the program in the VA Java IDE to show that it is checked out.
7. We now double click on the member to go into edit on the Java source. Using the IDE we find the piece of code that needs to be changed and we change the state to IL and close and save the code.
8. We could test the change again in the VA Java IDE but as it was such a minor change we can go ahead and just check it straight back into SCLM.
9. So we select ***Tools***, ***External Version Control*** and ***Check in*** and Cloud 9 starts the process to add the member back to SCLM. Once checked back in the padlock disappears next to the module to show it is no longer checked out.

SCLM Suite Demo
Demo Script

Process the members in Cloud 9

1. From VA Java we can jump directly to Cloud 9 by selecting ***Tools***, ***External Version Control*** and ***Open Version Control*** . This starts of your default web browser with the IP address and port that has been stored for this particular VA Java project once you have entered your userid and password.
2. In the demo our Cloud 9 HTTP server also requires us to log in.
3. Cloud 9 for SCLM is loaded into the browser. We can select "LIST SCLM FILES" to get a list of the files that exist in SCLM on the mainframe for our IBMDEMO project. We could use the various filtering options that Cloud 9 offers such as type, userid, language or wildcards of these but for this demo we will just select all members that exist only at the DEV level in the IBMDEMO project. We click the Submit button to retrieve the data from SCLM.
4. The list returned shows all the Java source files and GIF files that we added to SCLM earlier.
5. The first thing we are going to look at is the manifest member that Cloud 9 created for us of the VA Java project that we added to SCLM. As you can see it has a "long name" even though it is stored in a normal PDS in SCLM. This is because Cloud 9 uses a file called the SLR (Short to Long name registry) to contain a list of the long names with the resultant generated short name that is used within SCLM. Whenever we look at these member lists in Cloud 9 we will see the "long name" or the real name of the member. If we were to look at a similar list in native SCLM we would see the generated short name. We will see evidence of this later. We select the package member that contains the manifest and click ***View Member***.
6. A new browser window is spawned that is loaded with the contents on the manifest package. We can see in this member some information including the long name of the file. As this is an SCLM architecture definition we can see the SCLM architecture control statements for controlling the build and promotion activities. These are the INCLD statements that you can see. As you can see SCLM only knows the members by their PDS member name so that is a generated name that is stored in the Cloud 9 SLR.
7. We close the browser window and go back to the member list. Up until now we have just added the source members from VA Java to SCLM. We do not have any executables, so we need to generate these. Using the package archdef that Cloud 9 created for us we click on the build action and select a batch build and press submit.
8. A batch job is submitted that will "build" all of the members specified in the archdef member. SCLM uses an intelligent build so will only build those members that have changed. In the case of our VA Java package it will build all the members as they have never been built before. What that "build" process does depends on the language type of the members. In the case of the Java source members, they will be copied over to USS and a Java compile will be issued against them. The resulting class files and listings will be returned and stored in SCLM. In the case of the .GIF files they will just be copied to the USS. This will enable you to run them from a browser if you require.
9. Once the build job has finished we go to SDSF and take a look at the job output. When we look at the SCLM messages we can see that build has completed for $VI00001 which is the PDS name associated with the long name given to the package. SCLM only knows the PDS member names.

10. We now look at the build messages and can see the messages for each individual member that was built. We again can see the short name that SCLM knows along with the type that was built.
11. We now look at the SYSTSPRT output and we can see detailed information produced by the translators. This is fairly verbose because we have the Cloud 9 Java traces turned on, but it is useful to see what is going on. We do a find on Snake.Java and look at the various messages relating to this until we can see the output from the Java compile.
12. We now go back to Cloud 9 to look at the list of generated members. We go back to the main SCLM Query screen and just submit the original query again to get an updated list. As we can see from the displayed list there are now class and listing files that were not there before. These were all generated by the SCLM build process.

SCLM Suite Demo
Demo Script

Using Native SCLM

The next thing we will do is use the native ISPF interface to SCLM to list some of the members we have already been working with in VA Java and Cloud 9 and then work with a COBOL source member. We could work with the COBOL member through Cloud 9 also but for completeness of the demo we would like to show native SCLM also.

1. From the ISPF main menu enter option 10, which is the standard location you can find SCLM in the released version of ISPF.
2. We have already entered a PROJECT of IBMDEMO. We have left the alternate blank so this will also default to IBMDEMO. This is how SCLM knows which project definition load module it needs to load. If we had an alternate project definition, as discussed earlier, we could enter it here. The group is set to DEV which is the development group we will be working in. SCLM only lets you edit members in the groups defined at the lowest level of the hierarchy.
3. We are going to use option 3.1 to work with members as this is a list processing view, similar to option 3.1 in ISPF. From here we can enter various SCLM actions against members.
4. In the library utility panel we are going to view the Java source library that was migrated in from VA Java. The project and group are derived from the main SCLM panel, so we enter a type of JAVA and set the member to *.
5. The list of members returned obviously have the generated member name because SCLM doesn’t know about Cloud 9 and the SLR database for short to long name conversion.
6. We view the first member PET00006 and as you can see it is just normal Java source.
7. We go back out to the library utility panel and change the type to COBOL and enter a new member name of COBPGM01 and press enter. We get a message saying no data to display. SCLM has gone and looked up it’s hierarchy for the member and hasn’t found it. If there had been a member SCLM would have shown it to us in a list. We could then have edited it, at which point SCLM would have taken a copy of it from wherever it found it and placed the member in the DEV library.
8. As this is a new member we enter an “e” on the command line to go into edit on an empty member, where like every good programmer we copy someone else' s work.
9. Once we have finished editing our program we save it by pressing PF3 (or we can type Save) and at this point SCLM needs to know some information. SCLM needs to know the language type associated with this member so that when we build the member SCLM will know how to compile it. The languages were defined in the project definition we reviewed at the beginning of this demo and a list of the valid language types can be retrieved by putting a ? In the language type field.
10. From the resulting list we select a valid language type, in this case COB2, as this is a COBOL II program. When we press enter SCLM stores the member and also updates some accounting information relating to the member.
11. The next thing we edit is an ARCHCC. This is a special architecture definition called a compilation control ARCHDEF. These are optional but provide a means for the developer to override certain compilation characteristics that are defined in the language translator. For instance 95% of your project may use the standard compiler options defined in the COB2

language translator. For the other 5% you could override them using ARCHCC members. You could also have a separate language translator, called COB2X for example, and use that language type instead of COB2. In which case ARCHCCs would not be necessary. Again we copy the ARCHCC in from somewhere else.

12. The ARCHCC contains SCLM control statements to give SCLM some information about what to do with the member. The OBJ and LIST keywords are telling SCLM the name of the library types OBJLIB and LISTLIB that will contain the outputs from the build process. The SINC keyword is a source include and specifies the name of the COBOL source member from the COBOL library type. Finally the PARM keyword provides compiler option overrides that will be used when the language translator is invoked.
13. Again when we press PF3 to save the member we are asked by SCLM to enter the language type. Again we can select from a list and this time we select ARCHDEF as this is the language type for all of the SCLM architecture control members.
14. The next thing we are going to create is an ARCHLEC member. This again is a special architecture definition called a Link-edit Control ARCHDEF. This contains information to pass to the link-edit language translator. Again we copy the member in from somewhere else.

15. The ARCHLEC member is similar to the ARCHCC member in that it contains SCLM control statements this time for the link edit process. The LOAD and LMAP define the output library types LOADLIB and LMAPLIB. The INCL statement is including the ARCHCC member we created before. The INCL statement is used to include other architecture members, where INCLD and SINC is used to include actual source members. Again the PARM keyword contains link-edit option overrides that will be used when the linkage editor language translator is invoked.
16. Again we press PF3 to save it and this time we just type in the language type rather than getting it from a list. And press enter to save the member.
17. If we are just writing a new program we will want to compile it a couple of times to get the compilation errors out of it. We can do that by just building the COBOL source member. Changing the library type back to COBOL we get a list of our COBOL modules and enter a “C” next to it. This brings up the SCLM build entry panel. These options are standard so we just do a foreground build. We can send the listings to datasets or to the screen, in this example we have chosen datasets. The build comes back with a return code 0 to show that the compile worked OK.
18. We are put into browse on the build report and can see in the report the things that were created by the build process. We have an Object deck and a listing created. It tells us the member that was the reason for the rebuild. In this case it was the COBOL member. SCLM is an intelligent build and will not build a member if it doesn’t require it. But if for example a copybook is changed and the ARCHDEF containing it is built then you will see a list of the objects created and this will contain object decks for all the COBOL members including that copybook. The reason for rebuild will show the copybook that changed as being the reason why all the object decks were created.
19. We now show what happens when we build the ARCHLEC member. The process is the same if we are building a single source member or an ARCHDEF that contains multiple source members, such as the one we built for the VA Java manifest. Again we run a foreground build and take a look at the build report. We can see that the OBJ and LIST were created

again, but this time we look at the reason for rebuild and we can see that it was because it was the first time we had built the ARCHCC member. If we had changed the COBOL source then that would have been the reason for rebuild. We also see that a load module and a load map have been created. If we had a ARCHLEC that included many source modules then by just building the ARCHLEC we can get SCLM to compile all the source modules we have changed and then recreate the load module all in a single step. Rather than compiling each individual source member followed by a link edit. We can let SCLM determine for us what needs to be done.

20. We now list the LOADLIB type and see the load library we have just created in the list.
21. The next step is to add the COBOL application we have just created to the architecture definition that Cloud 9 created for us for the VA Java manifest. If you have groups of modules that are related to each other these can all be listed in a single architecture definition. You could have Java code for NT along with some mainframe server code and maybe some documentation. These can all be stored together in SCLM with a controlling ARCHDEF keeping track of them all.
22. We change the type to PACKAGES and get a member list up and see the generated short name of the VA Java manifest package. Even though this was generated by Cloud 9 we can still edit it here. Or we could use Cloud 9 to add new members to the package. For now we will just edit it on the mainframe.
23. We go to the bottom and repeat the last two lines to overtype them. We then enter a comment and create an INCL control statement for the ARCHLEC member we created earlier. This way whenever the overall architecture member is built the COBOL program will be compiled and re-link edited, if required.
24. When we press PF3 to save the member SCLM does not ask us for the language type as it already knows what the language type is from when this member was created.

Using the Cloud 9 SDSF viewer

1. As the package contains some e-business type objects we will use Cloud 9 to build it, so we return to the browser we had open. The package is still selected so we just click on the build action and submit a batch job to do the build.
2. We will now check the output from this build job using the Cloud 9 SDSF viewer. We start up a new browser window and change the URL to invoke the SDSF viewer HTML.
3. The SDSF viewer is displayed in the browser. So we change the job prefix to the prefix of the build job we have just submitted and click on the status action.
4. This returns a list of jobs where we select the latest one and select the view action.
5. This spawns a new browser window with the job output displayed. Notice the condition codes of each step are extracted and displayed at the top of the listing, color coded. In this case all steps finished with a zero return code so they are all green.
6. We scroll through and look at the output and see that no new outputs were generated for this build. All that had changed was the actual ARCHDEF member, so this is the reason for the rebuild. None of the source members that it had included had actually changed so SCLM did not rebuild them.

SCLM Suite Demo
Demo Script

Package approval using Breeze

Breeze can be invoked from native SCLM or from within Cloud 9 as it is hooked into the build and promote processes via standard SCLM user exit processing. For this demo we will invoke Breeze from within Cloud 9.

1. We go back to the Cloud 9 browser that we had previously and the package member that we have just built is still selected. We now select the promote action against it.
2. This brings up the promote pane that has a section in the middle that contains Breeze related fields. We can enter promotion window parameters and a description at this point if we like. For the demo we just enter a meaningful description of the package and let the promotion date window default. We click submit to submit the promotion as a batch job.
3. We then flip over to MVS to take a look at the output from the promotion job in SDSF.
4. Looking in the SYSTSPRT we see there are additional messages that have been produced by Breeze. The messages tell us that Breeze is initially checking that it is interested in the package being promoted. It then gets the defaults for the promotion window dates along with the description we entered in the Cloud 9 promotion pane.
5. When it is doing Content and Approver collection it is checking to see if any of the modules promoted as part of this package meet the records that have been defined on the Breeze database. If there are any matching “Inventory” records then notification is sent to the required approver groups either via e-mail or TSO SEND messages. The final message box is telling us how many approvers there are in the approval of this package and how many of them are “Required” approvers.
6. From here we switch to our e-mail system where we can see we have received an e-mail notifying us of a package that requires our approval. From here we can double click on the supplied URL and a new browser window is opened where the Breeze applet is downloaded.
7. Once downloaded we enter our MVS userid and password and click the OK button. Breeze will automatically show us the packages that require our approval. As you can see the list contains the package we have just attempted to promote.
8. We select the package and the Summary window shows us some information about the package such as it’s current status and the SCLM hierarchy that it effects.
9. We select the Contents tab and this shows us the contents of the package. By using the exclusions control member you can exclude certain types from this list.
10. Now we select the Ballot box tab which shows us the groups that are required to vote on this package with the individual voters id’s and their vote. The status is also show which changes once the quorum has been met.
11. We now click on the approve button to vote for the package. We can add some notes if we want and then we click Submit to submit our approval. The browser is refreshed and we now no longer have any packages requiring our approval.
12. By using the filter in conjunction with the “Packages by Status” area we can list packages that do not require our approval and browse them. We can select the package we have just approved and check out the ballot box again. This time we see the package status is approved in all groups as all the quorums have been met.

13. Again we check the Summary tab and see that the package status is showing it is approved, which means we can now promote the package.
14. So we jump back to Cloud 9 and go back to the promote pane and click the submit button again. Again a batch promote job is submitted.
15. We jump over to SDSF to check the output from the promote job. We check the promote messages and see a number of promote step messages for each of the files. This is because as part of the promote translator there is an FTP deployment to an FTP server running on a different windows 95 machine.
16. We check the Promote report and see that the COBOL, Java and all associated files have been copied from DEV to TEST.
17. Now looking at the SYSTSPRT we can see the breeze messages. Breeze recognises the package can be promoted and so SCLM begins the promotion process. We can see that the first thing done has been to send the .GIF files to the FTP server with IP address 9.190.174.244.
18. We also notice that as well as SCLM promoting the code through the SCLM hierarchy that the USS file system hierarchy is also updated to copy the Java and Graphics up to the QA level directories. This would allow us to run directly from the QA level from a web browser if we had some supporting HTML.
19. After paging through some pages of USS copy messages we see the FTP message that the Java class files have also been deployed to 9.190.174.244. We will do something with these files in a moment.
20. First we go back to Cloud 9 to show that the code has been promoted. Leaving the group as DEV we click on the submit button and see that there are no members found at the DEV level of the project.
21. So we change the Hierarchy view to be “First Found”. This will allow Cloud 9 to start at DEV and look up through QA and REL for occurrences of the code. We click Submit. When the list is returned we see that all the source and executables now reside in the QA level of the project.
22. We go back to the Breeze browser and refresh by clicking the “Go” button. The data in the browser is refreshed and by checking the summary tab we see that Breeze has also been updated to show that the package has been promoted.

SCLM Suite Demo
Demo Script

FTP Deployment

As mentioned during the Breeze promotion so files were deployed to an FTP server on a remote Windows 95 machine. To finish the demo we will retrieve those files and install and run them on our PC.

1. We have two files zC9FTP.bat and zC9FTP.dat that will be used to signon to the FTP server and get the class files and GIFs and run the application on our PC.
2. We select the zC9FTP.dat file and review the contents. These are standard FTP commands for retrieving the files.
3. We now review the zC9FTP.bat and see the various directory statements, the FTP statement that uses the .dat file and connects to the IP address that we saw during the promote step, and finally the Java statement that runs the application.
4. We run the .bat file by double clicking on it and we can see the script logging onto the FTP server and retrieving the files.
5. The Petdemo application then starts up again and we select some pets to buy. We click on the Check out button and the pick up details are displayed. We can see that the code that we modified to contain the correct state is what we have just run.
6. Finally we just review the files that we have FTPed back from the FTP server.

End of Demo