DYALOG APL

Simple APL Library Toolkit (SALT)

Version 1.26 dated 2008-7-31

Introduction

SALT is a source code management system for Classes and script-based Namespaces in Dyalog APL. The source code for each object (class or namespace) is stored in a single Unicode text file with a file extension of ".dyalog". SALT also supports the loading and starting of applications from an "application file" with an extension of ".dyapp".

Classes provide a convenient mechanism for wrapping tools in a way which makes them easy to share. SALT is intended to provide a common mechanism for APL users to develop and share code in "Open Source" libraries.

SALT aims to provide the minimum useful set of functionality for a small team of developers, and provides the following set of functions (in a "likely" order of usage):

Function	Modifiers	Discussion
List '*folder|object*'	-Folders	Only list folders, not individual objects
	-Recursive	Recursively list contents below named folder
	-Versions	Also list intermediate version copies of object
	-Raw	Returns unformatted data
	-Full=1 or -Full	Show full pathnames below root folder
	-Full=2	Show complete pathnames
Explore '*name*'		Opens explorer if name is a folder or Notepad on source if name is an object
	-Mode=Normal|Max	in Normal or Maximized window
	-Use=	Use specific program to compare
	-Permanent	Remember permanently the program to use
Load '*name*'	-Version	Loads a particular version
	-Target=Namespace	Defines the object in a ns
	-Source	Returns the source as a nested vector instead of defining the object in the workspace
	-NoName	Returns a ref to the object but does not "name" it in ws
	-NoLink	Do not manage the source for the object after loading it into the workspace
	-Disperse	Brings in the objects specified (or all)
New '*name' [arg]*	Same switches as Load (-noname forced)	Creates an instance of class 'name' without naming the class in the workspace
Save 'ref *file*'	-Version	Save particular version number
Compare '*name*'	-Version=n	Compare current (last) version with version n (default is to compare last 2 versions)
	-Version=n1 n2	Compare two particular versions
	-Use=	Use specific program to compare
	-Permanent	Remember permanently the program to use

if "APL" has been specified	-Zone= -Trim	# of lines to show before and after matches Ignore spaces at the ends
RemoveVersions *'name'*	-Version=[<>]n -Collapse	Drop specific version(s) n keep last version
	-All	Forget ALL backup versions
Boot 'app'		Loads and runs an application from a .dyapp file
Settings'[id [v]]'	-reset -permanent	Reload settings from registry Save settings to registry

The use of text files as a storage mechanism means that SALT and other tools written in APL can be combined with industry standard tools for source code management. For example, SALT allows comparison between versions of a class to be done using an external add-on.

SALT is included with a standard Dyalog APL installation for Windows but needs to be activated before it will be loaded when APL starts. See the section on **Configuration** below.

Some Implementation Details

File Format

Each version of each source object is stored in a Unicode text file with an extension of ".dyalog". The Unicode file format used is known as UTF-8. These files can store text which uses the "Basic Multilingual Plane" of Unicode, which contains most of the world's languages and the APL character set. This format is supported by very many applications (including Windows Notepad).

The source code for SALT is itself "*salted*", consisting of two files in the Classes\Dyalog\SALT folder.

Application files are stored in text files with an extension of ".dyapp". When Dyalog APL is installed, it sets up associations for the new extensions:

.dyapp Opens with Dyalog APL, which should Boot the application. Edits with Notepad.
.dyalog Opens with Notepad.

Configuration

The registry key "HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.0\SALT" contains the following values which provide configuration information for SALT:

Value	Default Setting	Discussion
AddSALT	1	Controls whether SALT is initialized when the session file is loaded.
CompareCmd	apl	Specifies how to call a file comparison utility.
SourceFolder	[Dyalog]\Classes	Specifies folders for source files separated by ";". Default folder is located below the main Dyalog folder.
EditorCMD	notepad.exe	Specifies which program to run to edit script files

SALT will be loaded into the session if the registry string **SALT\AddSALT** has the value "1" (the default). If SALT is active, you should not get a VALUE error if you type

```
]SE.SALT
```

If SALT is not enabled you can enable it using the SALT workspace. Simply type

```
      )LOAD SALT
      enableSALT
```

With V12 you can also use the configuration menu en/disable SALT.

Shaking the SALT

SALT is used to maintain itself, and the source can be found in the "SALTed objects" in the Dyalog folder SALT\SALT. As mentioned earlier, these files are loaded if the registry entry **SALT\AddSalt** has the value "1". If so, `SALTUtils` and `SALT` are loaded into `⎕SE`, and the function `SALTUtils.EditorFix` is connected to a callback on exit from the Dyalog Editor.

When SALT loads an object, it inserts a special namespace named SALT_Data in it, and variables inside this namespace contain the source file name, the version number and the last write time of the file when it was loaded. The last item of information is used to prevent accidental updates of the same version by two different users or from two different sessions.

SALT Applications

In addition to managing individual source code files, SALT is able to load and run applications defined by files with an extension of ".dyapp". The format of these files is documented under the Boot command. The Version 11.0 installation sets Dyalog APL as the application which is used to "start" these files, and SALT examines the command line

Comparing Files

Since the source code for each version of an object is stored in a Unicode file, any file comparison tool which can compare Unicode files can be used. The author did a quick search on Google and ended up evaluating and subsequently spending $29 on a product called Compare It! from http://www.grigsoft.com/index.htm (if you download this product, make sure to get the Unicode-capable version). SALT is able to use any product if it can be launched using a command which takes the names of the two files to be compared as parameters. The registry string **SALT\CompareCmd** tells SALT how to launch the comparer. If using the above product it should be:

"[ProgramFiles]Compare It!\wincmp3".

SALT appends the two file names and calls "Compare It!" If such a program is unavailable the registry entry should be left blank and SALT will use it own primitive comparison code and show the results in the session.

Versions

SALT files may be *versioned*. When versioning is switched ON for an object, SALT creates files which have a version number immediately before the .dyalog extension (for example, MyClass.3.dyalog). The List function in SALT shows this number as [3].

Each time an object saved in a versioned file is changed, a new file is created. You can quickly end up with a large number of intermediate versions. You will need to use RemoveVersions to tidy up.

Using SALT

A standard Dyalog APL installation contains a collection of classes which can be used to explore object oriented programming. SALT commands allow you to explore and use this library.

List

The List command takes an object or folder name as its argument. An empty argument will list the top-level files and folders (immediately below the first folder named in the SourceFolder registry entry[1]):

```
      ]SE.SALT.List ''
 Type   Name     Version  Size  Last Update
 <DIR>  Dyalog                 14-05-2006 21:32:54
 <DIR>  Samples                11-05-2006 08:21:15
```

If you get a VALUE ERROR when you try to use SALT functions, check the setting of the SALT\AddSalt registry string. This string needs to be set to "1" for SALT to be loaded into the session on startup.

The Dyalog folder contains official Classes Library supported by Dyalog (in version 11.0, they should be considered experimental). s*tudy* contains code which is referenced in documentation, or provided for self-study.

The List function takes a number of modifiers. All SALT functions can be called with a single '?' argument, in which case they remind you of the available modifiers:

```
      ]SE.SALT.List '?'
List pathname Modifiers:
-Full[=1|2]     1 shows full pathnames below first folder found;
                2 returns "rooted" names.
-Recursive      Recurse through folders
-Versions       List versions
-Folders        Only list folders
-Raw            Return unformatted date and version numbers
```

We can get a complete list of class folders as follows:

```
      ]SE.SALT.List '-recursive -folders'
 lib
 SALT
 spice
 study
 study\data
 study\files
 study\GUI
 study\math
 study\OO
 study\OO\QuickIntro
 tools
 tools\DanB
```

[1] the *SourceFolder* registry entry may contain more than one folder but they must be separated by a semi-colon. When listing a file or folder the first one found is listed

```
 tools\SJT
```

List the contents of the study\OO\QuickIntro folder:

```
      ⎕SE.SALT.List 'study\OO\QuickIntro'
 Type  Name     Version  Size  Last Update
       Product           570   08-05-2006 08:59:45
       Sale              756   05-05-2006 13:15:20
```

Load

The Load command takes an object name or a pattern as its argument:

```
      ⎕←⎕SE.SALT.Load'study\files\ComponentFile'
#.ComponentFile
      cf←⎕new ComponentFile 'c:\temp\compfile'
      cf.Count
2
      ⎕←⎕SE.SALT.Load'\myutils\gui*'
guiin guimsg …    guiout
```

Load returns a shy reference to the loaded class(es) or the result of []FX for functions. By default, Load also gives the loaded class/namespace a "global name" – in this case ComponentFile. See the description of the **New** command below for a description of the **–noname** option, which allows you to avoid the creation of the global name and use a class "without loading it into the workspace".

The -**Source** modifier will make Load return the source instead.

The**–Target** modifier allows you to load a class into a particular namespace:

```
      'MyFiles' ⎕NS ''
      ⎕←⎕SE.SALT.Load'study\files\ComponentFile -Target=MyFiles'
#.MyFiles.ComponentFile
```

The -**Disperse** modifier allows to bring in the objects IN the file as opposed to the object itself into the target namespace. If only specific objects need to be brought in they can be specified after as in –disperse=obj1,obj2,etc. The result of Load in this case is a global 1 (OK) or 0 (failed). No tracking information is kept in this case (see NoLink below).

```
      ⎕←⎕SE.SALT.Load'GUIutils -disperse'
1
```

The **–Version=** modifier allows you to load a particular version of an object, we'll show examples of this a bit later.

Finally, you can use the modifier **–NoLink** to specify that SALT should not insert tracking information into the object. If you use –NoLink, editing a SALTed object will NOT cause SALT to offer to save the source on exit from the editor.

New

The Load command takes a modifier called **–NoName** allows you to specify that you do not want the global name created. This allows you to load use a class without giving it a name in the workspace. The following example defines an unnamed class which is used to open a

component file and return the number of components, but leaves no trace in the active workspace:

```
      (⎕NEW (⎕SE.SALT.Load 'study\files\ComponentFile -NoName')
                                              'c:\temp\compfile').Count
2
      ⎕NC 'ComponentFile'
0
```

The **New** command provides a more direct way to instantiate objects from a source file:

```
      cf←⎕SE.SALT.New 'study\files\ComponentFile' 'c:\temp\compfile'
      cf[1]
 comp 1
```

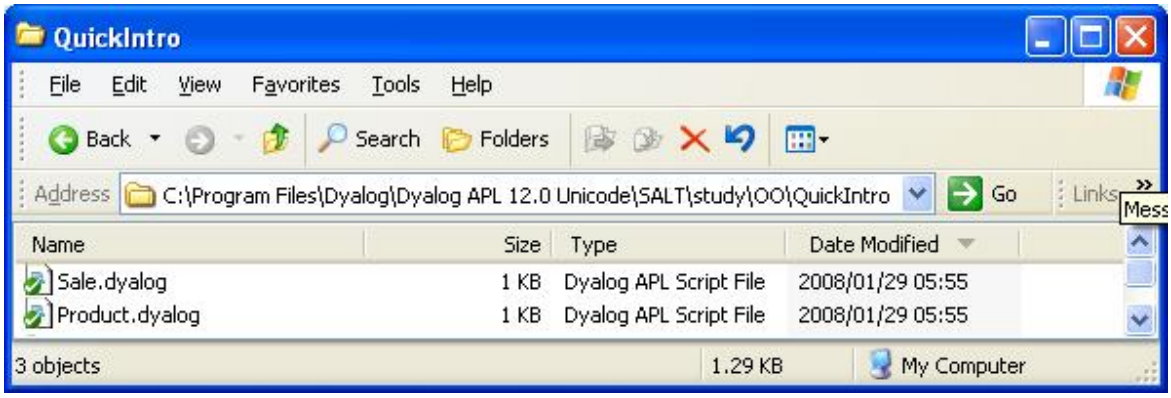
New passes the first element of its argument to Load, appending the **–noname** option, and then makes a new instance of the loaded class using the rest of the argument.

Explore

As an example of using SALT, we are going to load one of the QuickStart example classes, modify it and save it under a new name. We need to manually create a new folder to contain our class, because SALT is not capable of doing this for us. The Explore command can be used to open Windows Explorer up on a folder, for example:

```
      ⎕SE.SALT.Explore 'study\OO\QuickIntro'
```

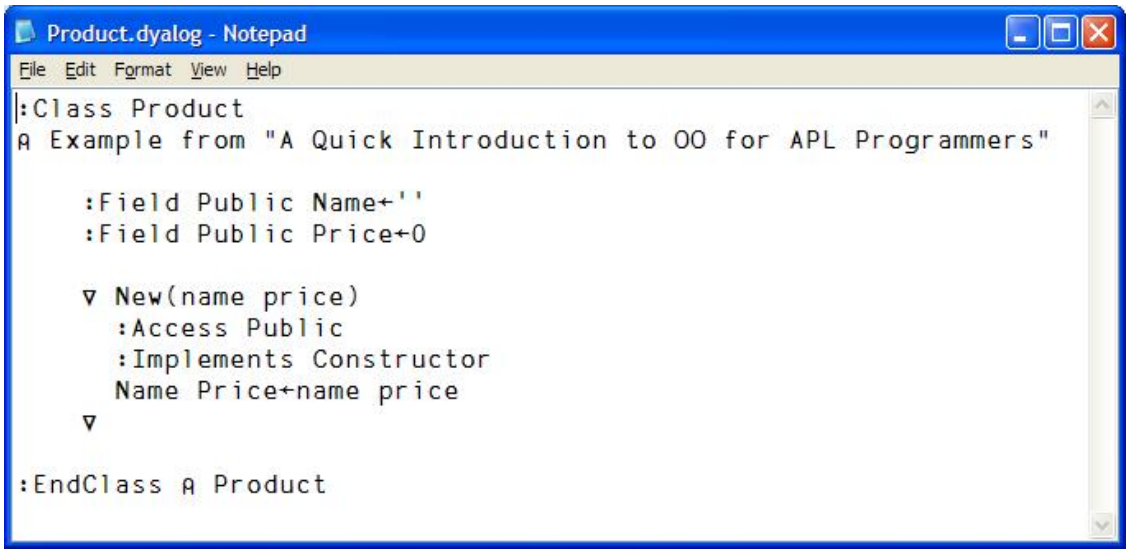
You should see explorer open up to show the contents of the Dyalog folder:



Move up a level and create a folder called Mine under the SALT folder. If the argument to the Explore command is an object name rather than a folder name, Explore will start the Windows Notepad:

```
      ⎕SE.SALT.Explore 'study\OO\QuickIntro\Product'
```

This is the class we are going to experiment with:



Load it into the workspace and edit the class, changing its name to MyProd:

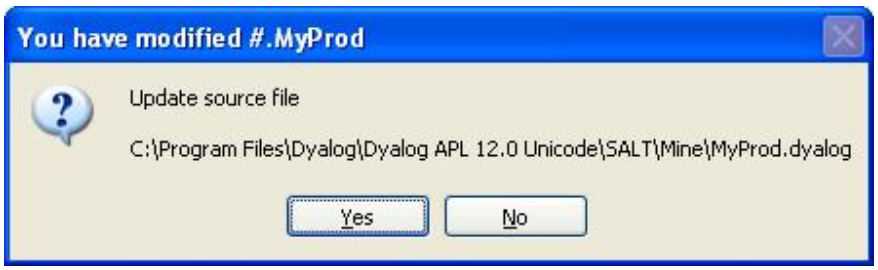
```
      ⎕←⎕SE.SALT.Load 'study\OO\QuickIntro\Product'
#.Product
      )ed Product
```

Save

We are now ready to save our class called MyProd in the folder Mine (which we created in the previous section). Save returns the full name of the file which was created:

```
      ⎕SE.SALT.Save 'MyProd Mine\MyProd'
C:\Program Files\Dyalog\Dyalog APL 12.0\SALT\Mine\MyProd.dyalog
```

Now, edit MyProd again and make a small change – for example to the comment. As you exit from the editor, you should see a pop-up similar to the following:



Click Yes and use Notepad and Explorer to verify that the file contains the new version of MyProd[2]. Experiment with clearing the workspace, loading Mine\MyProd, and verifying that all the changes you make are being saved in the file.

Versions

By default, SALT maps your class or namespace to a single file, and any change you make to the object overwrites the file. If you give your object a version number, SALT will start taking

[2] There is a way to prevent SALT from asking confirmation each time you edit a script, see Settings below

backup copies each time you make a change (note that modifiers can be abbreviated, so long as they are uniquely identified):

```
      ⎕SE.SALT.Save 'MyProd -ver=1'
C:\Program Files\Dyalog\...\SALT\Mine\MyProd.1.dyalog
```

When SALT notices that you are giving an object a version number for the first time, it starts saving the existing class under a similar name that includes a version number.

Each time you change MyProd and update the file, you will see a message confirming the creation of a new file. Make one or two more changes to MyProd and then call List with the *-Versions* modifier :

```
      ⎕SE.SALT.List 'Mine -ver'
Type  Name    Version  Size  Last Update
       MyProd  [2]       301  2008/02/02  9:28:30
       MyProd  [1]       301  2008/02/02  9:27:05
       MyProd            301  2008/02/02  9:26:08
```

Imagine that we are now planning a new release of MyProd, which we are going to save under version 10. We start the version 10 project by saving the new version:

```
      ⎕SE.SALT.Save 'MyProd -version=10'
C:\Program Files\Dyalog\…\SALT\Mine\MyProd.10.dyalog
      ⎕SE.SALT.List 'Mine'
 Type  Name    Version  Size  Last Update
       MyProd            301  2008/02/02  9:29:32
```

Since we haven't made any changes to MyProd yet, this version is identical to the last one. To see all versions we need to include the *–version* switch:

```
      ⎕SE.SALT.List 'Mine -ver'
Type  Name    Version  Size  Last Update
       MyProd  [10]      301  2008/02/02  9:29:32
       MyProd  [2]       301  2008/02/02  9:28:30
       MyProd  [1]       301  2008/02/02  9:27:05
       MyProd            301  2008/02/02  9:26:08
```

Now, change the constructor function in MyProd so that it sets the display form for the instance (this will create version 11), for example:

```
    ∇ New(name price)
      :Access Public
      :Implements Constructor
      Name Price←name price
      ⎕DF'[',(⍕Name,'@',(⍕Price),']'
    ∇
```

By default, the Load command will load the most recent version of an object. Verify that Load is loading the latest version by default, but that the other versions are still available:

```
      ⎕←⎕SE.SALT.Load 'Mine\MyProd'
#.MyProd
```

```
      ⎕NEW MyProd ('Widget' 100)
[Widget@100]
      ⎕SE.SALT.New 'Mine\MyProd -ver=1'  ('Widget' 100)
#.[MyProd]
```

If we combine the *-version* and *-noname* modifiers, we can in fact work with multiple versions of the same class at once.

```
      pclasses←{⎕SE.SALT.Load 'Mine\MyProd -noname -ver=',⍕⍵}¨1 11
      pclasses
 #.MyProd  #.MyProd
      pclasses.SALT_Data.Version ⍝ SALT version tags
 1  11
      {⎕NEW ⍵ ('Widgets' 100)}¨pclasses
 #.[MyProd]  [Widgets@100]
```

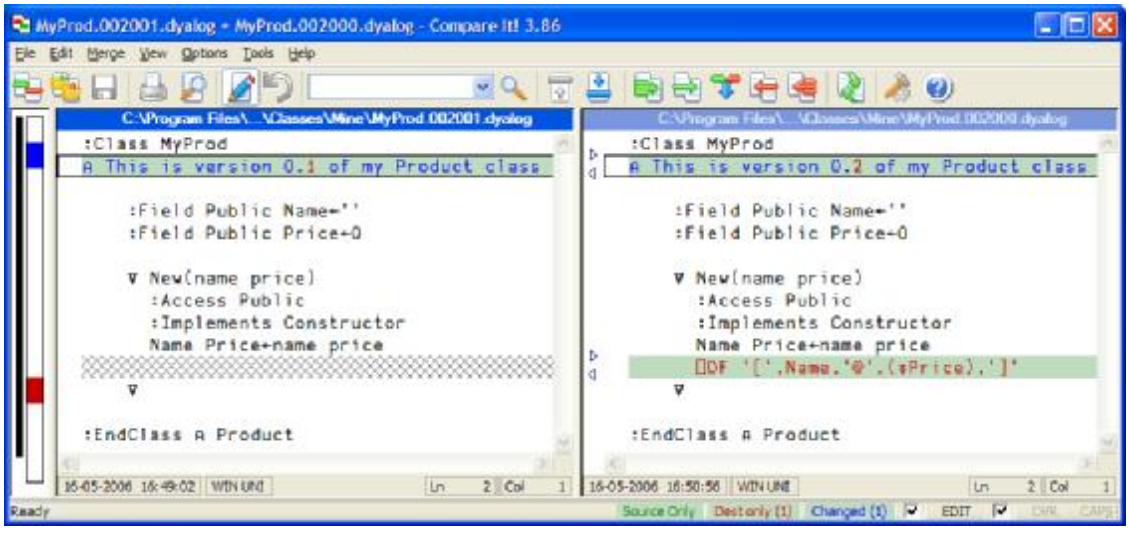
Compare

The Compare command allows you to compare versions of an object. Note that if you want the Compare command to use a third party product like the Unicode version of "Compare It!" or a different file comparison tool, you must use the **Settings** command OR modify the registry to show how to call the tool OR use the configuration menu (V12 only). See "HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.0\SALT\CompareCmd". Typically the entry will contain a string like "[programfiles]comparetool". If you leave the entry empty APL will use its own simple comparison function.

By default, Compare shows you the differences between the 2 most recent (highest) versions of the file given as argument (here *MyProd*).

```
      ⎕SE.SALT.Compare'Mine\MyProd'
```

Should bring up a screen which looks like this if you are using "Compare It!":



Compare also takes a modifier which allows you to specify exactly which versions you want to compare. You can compare the code for version 3 to the most recent version using:

```
□SE.SALT.Compare 'Mine\MyProd -ver=3'
```

If you want to compare two non recent versions, you need to provide 2 version numbers, for example:

```
□SE.SALT.Compare 'Mine\MyProd -ver=1 10'
```

If you want to compare the latest version of a class with a class with the same name IN THE WORKSPACE you can specify –version=ws:

```
□SE.SALT.Compare 'Mine\MyClass -version=ws'
```

Using a different program
Should you decide to use a program other than the one specified in the registry to perform the comparison you can use the *-use* switch to specify which program to use. For example, if you have 'Beyond Compare', another comparison tool from the Net installed and you just want to try it you can do

```
□SE.SALT.Compare 'Mine\MyProd -use=[ProgramFiles]\BC\BC2.exe'
```

This will not change your registry entry and subsequent use of Compare will use whatever setting you currently have set in your session.

RemoveVersions

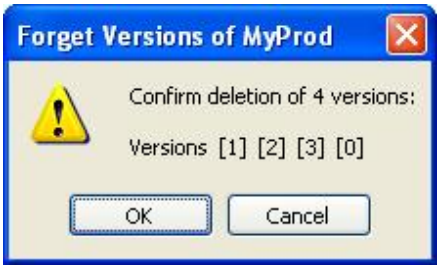
SALT creates a new file every time you edit a class or namespace. Therefore, you need to clean up versions occasionally.
This command takes three modifiers, two of which are mutually exclusive:

-Version=n	Specifies the version(s) which should be deleted
-All	All but the last backup copies should be deleted

For example (version 0 is the original version):

```
□SE.SALT.RemoveVersions 'Mine\MyProd -ver=<4'
```

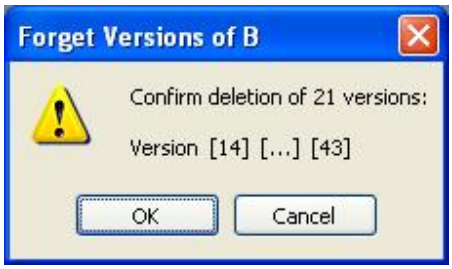
produces




```
4 versions deleted.
```

You can also delete trailing versions. If trailing versions are deleted they can be collapsed into one using –collapse. For example, suppose you have been working of a script starting at version 13 and you are happy with the result after you have made many modifications, only the last of which (version 43) you want to keep. Instead of listing all the versions to remove as in –ver=14 15 16 17 18 19… you can type

```
□SE.SALT.RemoveVersions 'Mine\MyProd -ver=>13 -collapse'
```




```
20 versions deleted.
1 version renamed
```

Versions 14 to 42 are deleted and version 43 becomes version 14.
In any case (-all or -ver=) versioning resumes at the highest number+1 when changes are made.

Boot

With the Boot command, you can use a script file to describe the loading and initialization of an application – as an alternative to using a saved workspace. The Boot command reads files with the extension .dyapp. Each line of a .dyapp script, except the last one, is a SALT Load command. The final line must start with the work Run, followed by the name of a method to call.

For example, a .dyapp file might read as follows:

```
Load study\files\ComponentFile
Load study\files\KeyedFile
Load MyApp
Run MyApp.Main
```

Note that, if there are dependencies between classes (as above, where KeyedFile derives from ComponentFile), base classes must be loaded before any classes which derive from them. SALT does not perform any dependency analysis but you can include statements to tell SALT to load other classes before. For example, if script A requires script B you should add this statement somewhere in A:

```
A∇:require path\B
```

Autostarting SALT Applications

If SALT is active, and APL is started with the name of a .dyapp file on the command line instead of an APL workspace, SALT initialization will call the Boot command on the named file. In this way, a .dyapp file can be used to auto-start APL applications which are based on SALT. Note that the whole application does not need to be "salted": Once started, the application can use]CY or other mechanisms to bring in additional source code.

Platform independence

SALT should perform the same way under Windows® and *nix platforms. To avoid confusion for people dealing with both environments SALT will accept SALT pathnames (only) using either / or \ as folder separator.

Under Unix there exist a version without GUI, which works in "terminal" mode. Under that system SALT must be enabled manually through the workspace 'salt'. Simply)LOAD salt and use <enableSALT>.
The workspace can also be)LOADed at startup time, just like any other workspace, by issuing the apl startup command followed by the path of the salt workspace as in

```
startapl ws/salt
```

If another workspace must be)LOADed afterwards or if a .dyapp file must run after simply put it in between, e.g.:

```
startapl  myws  ws/salt
```

Settings

Some commands require *global* parameters. For example, the **Compare** command needs to know which program to run to perform the comparison. This information is taken from the registry and loaded into SALT at boot time. It becomes a *session* parameter and can be modified using the **Settings** command.
In some cases Settings can also be specified on the line with the command using the *–USE=* switch only for the duration of the command.
For example if the default **Explore** program is not satisfactory and you want to try another one, say, vi.exe, then you can specify it on the command with –use=\myprogs\vi.exe
If you find this is useful you may want to make the setting for the duration of the session by entering

```
      ]SE.SALT.Settings 'editor \myprogs\vi.exe'
```

Should this prove unacceptable you can enter

```
      ]SE.SALT.Settings 'editor -reset'
```

to reload the value from the registry. On the other hand if those values are quite acceptable and you wish to make them permanent you can issue

```
      ]SE.SALT.Settings 'editor -permanent'
```

and the registry will be altered accordingly.

To see the list of all settings enter

```
      ]SE.SALT.Settings ''
```

Other settings are **workdir** and **edprompt**.

workdir allows you to have multiple working directories separated by semi-colon. To add a directory use comma, to remove one use ~, like this:

```
      ]SE.SALT.Settings 'workdir ,\proj\p1' ⍝ add \proj\p1
```

from then on files are stored under \proj\p1 but retrieved from where they are first found in the list of directories. SALT's files are always assumed in **[Dyalog]\SALT** even if that path has been removed.

edprompt determines whether you are prompted for confirmation to overwrite the file each time you make a modification to a script. The default of 1 prompts you each time.

Conclusion

The Simple APL Library Toolkit (SALT) provides basic source code management features for APL classes and namespaces stored in Unicode script files. By themselves, Classes provide new ways to make code sharing easier within the APL community. However, we believe that the full benefit of Classes will only be felt by the community if it also has a common source code management system, or at least a common file format which can be manipulated by a family of tools.

We hope that SALT will prove to be powerful enough that many users of Dyalog APL will decide to use it in real applications – at least as a tool to load shared utilities - and that it can be the beginning of a simple common source code management system which will provide the required platform for APL users to share utility classes and namespaces more effectively than they have been able to do so in the past.