ArcView/Avenue:
Coding Styles and Utility Scripts for Efficient Development

John Ganter, Sandia National Laboratories

Abstract
The effectiveness and efficiency of software development can be greatly increased by writing modularized code using informal (styles) and formal (standards) work approaches. Software development is about connecting pieces into a coherent whole. Thus consistent work approaches provide a structure that allows individuals and teams to minimize the time and thought that they devote to making these connections. These investments in structure return even more benefits in the maintenance phase when old code has to be examined by new programmers (or even the same authors after time has passed). While this philosophy is widely advocated in the software engineering community, what does it mean for the Avenue developer?

We present some examples of coding style for Avenue. These include a simplified form of Hungarian notation (notationHungarian, stringCustomerName, etc.), script naming prefixes and suffixes, and options in script headers. We demonstrate several modular, object-like utility scripts that can be used alone or combined into other utilities. These include developer tools such as a System.Echo substitute for Windows, a Window inspector, and a script for detecting and dealing with multiple display resolutions.

Contents
1. Introduction
2. Coding Styles
3. Utility Scripts
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
1. Introduction

The quality and ease with which computer applications are developed depends greatly on the effectiveness and efficiency of the developer. Effectiveness means doing the right thing. Efficiency means doing the right thing with minimal effort and wasted motion. The classical advice for increasing effectiveness is to perform careful, iterative design and "don't re-invent the wheel" (Davis 1995, Principle 66). (Design is beyond the scope of this paper, but its absence should not be misinterpreted. This paper suggests some practices for the work that comes after what is assumed to be good, thorough design.) I will present some ideas on how ArcView/Avenue developers can avoid re-inventing both their own and other's wheels.

Another key aspect of software development is respect for cognition, or human information management. Software is compiled and executed by computers, but it is written, debugged, reviewed, edited, and maintained by people. The valuable advice to "think first of people" (Davis 1995, Principle 92) can lead to a number of suggestions for developing habits and mnemonics. A few simple procedures can help compensate for the passage of time and the limitations of human memory, as well as helping the person who may inherit or acquire your code.

2. Coding Styles

*Style* suggests a shared consistency of appearance and organization. In writing code, style can mean anything from the rules of thumb for an individual programmer (informal) all the way to corporate standards that everyone is encouraged or required to follow (formal). Regardless of where an individual programmer falls in this spectrum, coding style is worth cultivating. Before long, it becomes ingrained and you don't even have to think consciously about it. Style and consistency helps your mind, and the minds of code readers or future programmers, to focus on the *meaning* of code and ignore noise. Stated another way, a program without style may be comprehensible to a compiler but noisy, chaotic, and hard to follow for a person. In this section we will talk about style as it applies to program documentation (headers and comments), and the naming of variables and files in ways that impose a useful structure.

**Internal Documentation**

*Headers*: The key to organizing a usable collection of scripts is a well-organized, consistently-applied header. A header can be thought of as the interface between a program and a person. It is also a *template* that can be used to start new scripts. We are often told to "not judge a book by a its cover," but a programmer must constantly judge programs by their headers. *What does this script do? Which version is this? Did Bill make that change in this script? How do I call this thing?* These are the kinds of silent
questions that programmers ask constantly. The quality of the answers that the header provides has a large impact on the effectiveness of the programmer. The standard Avenue script collection has the AVHeader script that is a good starting point for your (or your team's) customization. I have made a number of enhancements to my own header that are discussed on the avu-aaTemplate page.

Comments: Headers are an important form of comments, which should be used liberally throughout programs. As code is re-used or maintained, by the author and/or others, well-constructed comments become more and more valuable with age. I have found that it pays to review comments about a week after they are written. Try it. You will be amazed. You will find all sorts of omissions and assumptions. Each of these corresponds to the context that was in your mind at the time you wrote the comment, but which is gone now. One week is a good review time, because you can now see the problems but also can remember enough to fix them.

Loops and If...Then...Else constructs are challenging both in construction and maintenance. To debug or modify these structures you have to form a mental construct of them. Careful indentation and comments after each end statement are essential. This will help you to detect and correct the subtle errors that are typical in this kind of logic.

Notation

A typical program has more variables than any other construct. Variables are a programmer’s nouns; we use dozens, even hundreds in a single application. It follows that any technique that helps us to mentally remember, distinguish, compare, contrast, differentiate, or manage variables can have a big impact. We need a way of making mnemonics. The best way that I have found is what is known as Hungarian notation. Hungarian notation, which was invented by a Hungarian and looks vaguely Hungarian, is popular in C and Windows programming (McConnell 1993). In Hungarian notation, the most significant part of a word goes in the most significant position. In English, this is the leftmost place in a word. For instance, instead of writing

MyNumber ++ MyNewNumber = MyString

we would write

numMy ++ numMyNew = strMy

The advantage is that now you can read each "word" (variable) as type of variable, variable. This is very effective since, in programming, keeping track of the type of things is often as important as the things themselves. Linguists have shown that we read words as shapes. In Hungarian notation, all the numA and numB, and all the strA and strB, begin to look similar. This allows the reader to mentally filter out the types that they are interested in, then focus on the details that differentiate them.

Since coding style has a big impact on maintainability of code, it is a frequent discussion topic in many languages. Visual Basic, perhaps because of its widespread use in corporate IS, has a CompuServe sub-forum (GO VBPF) devoted entirely to the subject. But style has not received much attention in Avenue. This is ironic since Avenue is a completely untyped language: a variable can be an object, the name of the object (a string object), a number, etc. The Avenue compiler does not care. But Avenue at runtime does care! If you
give a Request a string argument, and it is expecting a View object, you're going to hear about it. So it is up to the programmer to create their own coding discipline.

Below are some examples of Hungarian notation for Avenue:

**Examples of Hungarian Notation for Avenue:**

- `angdTheta` angle, degrees, theta
- `angrTheta` angle, radians, theta
- `angdAzimuth` angle, degrees, azimuth
- `angrInterior` angle, radians, interior
- `boolResponse` boolean, user response
- `boolExists` boolean, exists
- `countLoops` count, implicitly a number, for Loop
- `docNew` Document object, new one
- `ftMain` Ftab object, for Main
- `fieldFrom` Field object, From
- `fnInput` File Name object, Input
- `fnOutputNew` File Name object, Output, the new one
- `fInput` File object, Input
- `intPeople` Integer, People (must be whole)
- `listSuffixes` List object, suffixes
- `listRoutes` List object, routes
- `numCount` Number, counter
- `numWindowHi` Number, Window, height
- `numWindowWi` Number, Window, width
- `objChosen` Object, undifferentiated
- `strViewName` String, View, name
- `viewMain` View object, Main
- `vtAddresses` VTab object, Addresses

Hungarian takes a little getting used to. One possible objection is that it could increase the length of variable names, and thus the amount of typing that the programmer must do. But I find that I use Cut/Copy-and-Paste Clipboard operations for most variables anyway, so it does not make much difference. If you don't already know the Windows Ctrl-C (copy), Ctrl-X (cut), and Ctrl-V (paste) hot keys, take a few minutes to learn them. Combined with a quick double mouse-click to highlight sections of text, they will save you huge amounts of time and motion.

Since starting to use Hungarian notation seriously, I have been finding that I can copy whole sections of code from one script to another, and it fits right in -- the variable names turn out to be the same. Also, longer, more descriptive names require less commenting. In looking over my code, I notice that my usage has evolved and there are some inconsistencies. But overall, I think that Hungarian notation has greatly improved the readability, reusability, and maintainability of my code.

**Script naming**
Hungarian notation is especially useful for script and file names, since it alphabetizes things, again, by type first and individual second. Consider a large Avenue project, with dozens of scripts. By using Hungarian notation, the scripts fall into natural categories. So when you come back in six months and have to work on the Table processing code, voile, all of the table scripts are grouped together in the Project window. There is tableCreate, tableCheck, tableOutput. It all starts to come back to you. The alternative is trying to remember: Did I have a MakeTable? Or was it CreateTable? Is CreateDoc actually a generic script that ??... etc.

Since I like to share scripts and Copy-and-Paste code, I tend to work on several applications with a single huge Project file (backed up hourly at a minimum) containing scores of scripts. So I’ve taken Hungarian notation one step further. Here is a typical Project window:

You can see the boundary between three categories of scripts. The avu- prefix denotes Avenue utilities. By virtue of names starting with "A," these utilities are always handy at the top of the list. Whenever possible, I write code in the form of a utility so I never have to write it again. Within the utilities, I can find scripts filed by category of Avenue object: App(lication), Doc, List, Symbol, Table, Tool, Window, etc. Simply by placing the name of the object type in a more significant position (to the left), the scripts follow the Avenue object model.

The second category of scripts is an application; the names all begin with eis-. So, this whole application stays together as a compact cluster in the Project window. The third category is another application, this time beginning with g. Like the other sets, it begins with a Template, which includes a header and is based on the standard template discussed below. But it has some of the header lines filled in with data specific to this application. Many of the scripts also have a -Tester suffix. As the name indicates, this is a ready-to-run test script that illustrates the calling syntax and perhaps shows some useful variations.
The idea is to let the programmer quickly try out a script that they may have forgotten, and give them plenty of Copy-and-Paste code snippets.

Compare this to a collection of more typically-named scripts. Your problem: you need to find a script dealing with object tags in Graphics. The first thing you are required to do is search through every single script name in the collection, looking for the phrase graphics. Clearly, if all the scripts associated with Graphics were named "GraphicsXXXX," this task would be greatly simplified.

File naming

Smart naming of files can also save time and jog the memory. When a Script is in a text file, I append the .ave suffix to the filename. I do not use any dot suffix (.ave) for Script objects that are in a Project or open in a script editor (SEd). This helps to keep the various forms of a Script clear. It also is very useful in another way. DOS/Windows and UNIX attach special significance to the .suffix part of a file name. It is easy to list or perform operations on *.ave, as discussed next.

File operations

One of the advantages of having a directory full of Avenue text files is batch operations. For instance, you can search your code for specific words or phrases. This is handy when you need to do something a little complex, and you know you’ve done it before...but in which script? By using a utility like grep, it is easy to find out and then you can Copy-and-Paste the code.

Another handy search is for flags in your code. For example, I often insert a ’ ! ! ! ! comment as a reminder that a section of code needs to be revisited. The code might need to be checked over, enhanced, made more efficient, etc. Using grep, it is easy to create a "TO DO" list for a whole project. Here are some grep commands that I have found useful:

- list all lines that are comments:
  ```bash
  grep -n \' *.ave
  ```
- list all lines flagged with exclamation points:
  ```bash
  grep -n \!' *.ave
  ```
- list all lines with the letters 'ft':
  ```bash
  grep -n ft *.ave
  ```
- list all lines with the phrase 'CALL AS' in my collection of utility scripts:
  ```bash
  grep -n 'CALL AS' avu-*.ave
  ```

Grep and related utilities like awk, cat, and diff are not limited to UNIX systems. The MKS Toolkit brings grep and most other UNIX commands to Windows 95/NT and other environments. While programmers not raised on UNIX may judge these utilities by their odd names, they are very powerful and complementary to GUI-based tools.
3. Utility Scripts

Writing utility scripts is a habit that can return long-term benefits. Most functionality is
generic. You may need to throw together a quick prototype, or do more formal
development. But there is no reason that either type of development cannot consist of 20-
60% new code and 80-40% carefully-crafted utilities that you have assembled from
previous projects.

Avenue does not give you much help in code reuse. The developer has to "fake" their own
utility libraries, version control systems, objects, object request brokers, etc. But this
situation will change, and habits of mind (thinking objects) will be even easier with new
tools and techniques (language-supported, user-defined objects). For now, you can
achieve code reuse through Copy-and-Paste, cloning of existing scripts, and designing
your applications so that a few specialized scripts make calls (the av.Run() request) to a
large collection of utility scripts.

The sections that follow are some of the early stages of a typical software life cycle: a
new script, development of functionality, code monitoring and management, and finally,
runtime configuration. At each stage I will present a few utility scripts and describe how
they assist in the process. (To show the scripts in a different way, an
Index by Avenue object is also provided.) Remember, some scripts call others. Check the
'CALLS TO:' line in the script header for these dependencies.

Stage 1: A New Script

It is useful to have a standardized template as a starting point for your scripts. The
avu-aaTemplate script is mine; you can customize it or other templates to fit your own
needs. A new script can be created either from an existing script or your template. Either
way, the avu-DocCloner utility will make a clone of any active Document (View, Table,
Chart, Script, etc.), and ask you for a new name. If, as suggested above, you use
consistent variable names, cloning a script will often provide a large amount of re-usable
code. This is why, once a project is underway, I mostly clone new scripts rather than
going back to the template. Chances are, most of the variables are already present and can
quickly be modified through Cut-Copy-and-Paste operations.

Stage 2: Development of Functionality

How big is this window? Many applications require that ArcView Windows be resized
dynamically. For instance, the windows may need to be tiled in a certain way when the
user adds another. Or ArcView may be sharing the desktop with another application. The
developer must therefore be able to experiment with window sizes. Under UNIX, the X
windows environment provides a 'digital readout' of window size and position whenever
it is resized or moved. Not so in Windows. The avu-WinHowBig script pops up a
MessageBox that reports the size (in pixels) of the display and the size/location of the
Application and the active Window.

Easier Trigonometry Doing trigonometric calculations and making custom graphics can
become interesting in mixed coordinate systems (i.e. coordinates may have a mixture of
positive and negative values). Avoiding standard angles (where 0 is east) and using azimuths (0 and 360 are north, 90 is east, 180 is south) can greatly simplify matters. The avu-Line2Azimuth script accepts two X,Y pairs (the start and end point of a line), and returns an azimuth.

**Echoing in Windows** Debugging code, especially loops, is often easier if you can System.Echo a string to a window. Unfortunately, there is no way to do this under Windows, Macintosh, and other non-UNIX environments. The avu-EchoSED script uses a SED script editor window for echoing. The user can then save, copy, or print the SED, which may be an advantage even on UNIX systems.

**Stage 3: Script Management**

**Printing long lines in scripts** Printing out scripts is useful for debugging, reviewing code, and making hardcopy backups. Since Avenue has no line continuation character, it is sometimes necessary to have long lines of code in scripts. If you are writing an article or report with code in it, you may have to insert comments like "next two lines are one line," etc. Most Windows print drivers will, instead of wrapping or ignoring characters on long lines, print them on separate pages. The avu-PrintWrapper script will wrap lines, with a line continuation character of your choice. The avu-PrintWrapperCaller script will operate on the Selected scripts in the Project window.

**Make safe text files often** ArcView Project files (.apr) are large, complex, and have been known to become corrupted. It's a good idea to make safer Avenue text files frequently. The avu-ScriptSaver script is a modified version of the SaveScript file that is included in the Avenue code library. It allows you to choose a directory, saves all the Selected scripts in the Project window, and preserves long, mixed case filenames. The avu-ScriptSaverCaller script is a helper application that can be invoked by a Button. The avu-DocList and avu-DirectoryGet scripts also help out.

**Stage 4: End-User, Runtime Configuration**

Once an application is in the field, being run by a user, all sorts of things can occur. One of the most common, specially under Windows, is a change in display resolutions. For instance, you may have developed an application using a 800x600 pixel display, but the user may only have a 640x480 pixel display. The script avu-AppSizeFix, when run as a startup script (or from a startup script), will help manage this situation by adjusting the size of the ArcView App(lication) object.

---

**4. Summary**

Life is too short, and the carpal tunnels too tender, to write unnecessary code. This paper has suggested, as have many articles and books by software engineers, that code should be viewed as an investment with a life-cycle. The classic advice for those who want to be effective (do the right things) and efficient (do the right things with minimal effort) is: *Design. Prototype on paper. Design some more. Build with components. Treat your code base as a re-usable investment. Do it right the first time.* Here I have attempted to bring these principals down to ground level for the practicing Avenue developer. Cultivating
habits of style, like Hungarian notation, can quickly become a painless procedure that returns benefits in the readability and comprehension of code. Coding habits, like the development of object-style utility scripts, can allow the developer to amass a collection of pseudo-objects long before there is built-in Avenue support for them. The net results of these habits and disciplines is that the developer can write less code, in less time, which works better and more reliably.

5. Acknowledgments

This work was supported in part by the Transportation Data Analysis and Visualization, Energy Intelligence Information System (EIIS), and Dynamic Simulation Systems and GIS projects at Sandia National Laboratories. This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

6. References


DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.