|  |  |
| --- | --- |
| 4D SUMMIT 2020  ARGUS Development Environment(ADE)  Presented by: **Jody Bevan** | page1image27590656 |

Introduction- Who is ARGUS

Jody Bevan has been developing commercially with 4D since version 2.0.10 (~1988). Wrote many custom databases, then developed a Paperless, knowledge base Medical application that became the largest medical application of its type in Canada with 20% of the Canadian population cared for in that system. We went international with the application (JonokeMed®) until the application was sold. After a few years break, I have now taken that experience, and have been developing our ARGUS Development Environment on top of 4D.

Adam Bevan has been developing commercially with 4D since 2000. Adam became part of the development team for the Medical application. Over the years, he worked his way under the high expectations of Jody, to become the Development Manager. In January 2020 Adam got his work permit to go work in the Medical Software Field in Australia. He is now there adding his skills and experience.

We have used our experience of managing teams of developers, both in house and afar. Writing applications with thousands of users over all in systems with 120+ concurrent users. The highly regulated medical industry requires the highest quality code, uptime, and proof of meeting standards. Having taken our company through ISO 9000-2000 software validation gave us knowledge of many of the requirements of driving a company’s methods to be the best.

With this background of over 52 years (combined ) of commercial 4D development, and the new features that 4D has built in, we decided to work on making a new shell for us to work on new projects with. What we found is that many of the new features of 4D permitted us to do things that we could only have dreamed about previously. Now we can do so many more things in all areas of being professional developers to make our products superior, and our lives and with less pressure on us because we can be assured things are done right. Therefore what we created using only 4D code, was more than a shell we have been told, but rather an Enhanced Development Environment for 4D.

Purpose of Presentation

The presentation’s intention is to show how at ARGUS we have used the 4D language to create tools to improve the speed of our coding, to improve the quality of our code, improve the management of our projects, and prove to a higher degree the quality of our code. This is all done using 4D language code – nothing else. These are issues that every programmer should be concerned about, and want to continually improve. This is a tool that we now use for all our projects. As we use it, we get more ideas and add more things to the ADE tool. We did this using versions 4D v16 and v17. There are features already in v18, that we are looking at implement in ADE that will enhance this even more. We are especially eager to create forms via code. This is likely the biggest speed and quality improvement we can implement. Form creation (user interface) always take a lot of time, even with the tools we have in ADE. Though I laughed at creating new forms in v17 with ADE, with v18 I am hoping we can make this extremely fast and simple.

Format of Presentation

First I will introduce you to the Argus Development Environment (ADE) as used within a real project. This will let you see a more feature rich interface than is provided in the Demo provided to you on the 4D Summit flash drive. I will describe a feature or set of features that I plan on targeting in the demo. I will demonstrate the feature to you. In some cases (especially the QC module) there are programming ramifications that I will describe. For some of the features I will show some of the key code in behind that lets you do what we have done. This will then give you an idea of how to write your own if you have the time. The Demo application that I have written for this gives you both examples of it working, and the code behind it. The code examples are included in the database that comes with the 4D Summit 2020 presentation data. Note that you cannot compile this app, because on purpose I have methods that have errors in them. I need these errors for the tool to work with and find.  
  
The demo code covers topics that can lead to improvements in your quality and speed of development. The following topics are covered. These represent a small percentage of the features in ADE.

Resources

The resources that come with this presentation are a small application that I have created just for the presentation. I have extracted the code I use out of the ADE (or wrote it from scratch) and put it into the demo application. I have taken out our comments, and a lot of our ‘bling’ that we have in methods to make the interface shine in ADE. I decided to do this so that you would not be distracted looking through our code trying to find the critical information you need to make your application do these various tasks. You can then add the type of Bling that you have developed that fits with your own user interface bling that you use anyway. This resource comes on the USB key from 4D for those that attend the Summit. Through this document I will reference the database that I created for this. In this document I will let you know what button to click on to go look at the code that does each of these tasks.

When you launch the database you will have a window open up with buttons on the window. The reference in this document will reference the button name.

The Demo does two things – demonstrates the feature working, and then lets you look at the code to do it.

* See the Feature Working: Click on the button in the ADE Code Demo Window
* See the Code for the Button Select the button name from the Menu: File -> Open Code.

Code Samples in Resource Database

There are some specific items that I will be demonstrating. Though I have created a new much simpler interface for these tasks, there is good code in there to show you how it is done. I have also added some other code into the resource application so that the interface is acceptable for a demo. Of course you get those too. The code contained in the Demo will show you how to:

* Create a menu via code rather than using the graphical menu editor we have known for decades.
* Open up a finder window for various 4D Resources, and logs that we create in our code.
* Process a method on screen with the purpose of declaring all local variables, removing non-used declarations.
* Writing to a method to change the code
* Having blocks of code that you can then add into an open method as needed
* Making the contents of objects much easier to read than as displayed in the debugger
* Having the Object Viewer display in compiled mode
* Creating a much nicer Method Explorer, that also contains additional information
* How to write your code so you can then write other code that will validate your code is written to your standards. Quality Assurance
* Determine the front most window in the Design environment, so you can automate modifying it.

ADE IMpetus

We wrote ARGUS Development Environment for our own use. As we were creating our shell from scratch again, I was not under any pressure to get it done for a client. Starting a new database from scratch – clean sheet, is very painful. None of the legacy code was there. All of the tools we were used to having, were not there. We wanted to do it this way though, so that we would recreate the tools we wanted, but using all the new commands and features offered by 4D. We started this with 4D v15, near the beginning for the v15R series. Therefore we really started this development in the early v16.

At the start it was challenging as to what to create first. There were hundreds of design decisions to be made. Fortunately Adam is very good at that stage of development in thinking about some very granular details that have major ramifications later on. I am very pleased with the work he did with ADE. As my last years at Jonoke[[1]](#footnote-1) were more high level management my mind set was on higher level tools. This is where ADE got its impetus. I would be coding and get distracted by ideas on how to make my tasks easier, more precise, and more consistent. I would then utilize code we had already written to create the tools for the ADE. I would start using them, and be raving about how good they were to Adam. I would show him and sometimes he would be unimpressed. I needed to listen to how he worked, and where his pressure points were to then write the tools I was making deal with his pressure points[[2]](#footnote-2). Once I got him hooked on a feature, he would then get carried away with it and make a whole bunch of great enhancements. This would create ideas for me and on we went. It was great to have a few years where we could be our own clients without major pressure.

ADE Tool Target

* New to 4D Developer or Developer Team: 4D is a sophisticated, rich, all encompassing development environment itself. As far back as the early nineties I thought it was an easy development environment to learn. Though it is, it still takes time. There is the learning of the commands, the form design environment, widgets, plugins, and more, and much more than the 90s. Then how to start a project, what design direction to take, and many more decisions to take. Having a ADE that is filled with shell code helps to make this much easier. The code that comes with it can get you up and running much quicker. You will have many tasks, design decisions, common interface features, security, and more automatically included in your application without writing any code. Adding in new features is much easier. Double clicking to add in code blocks to continue with a consistent rich environment is fast and easy. This gets you up and creating applications much faster, while having good standards, tracking of work done and much more.
* Making a leap from an old Version to v17 or v18: When one makes a big leap (say from 2004, or 6.8) there are very many new features of 4D that were just not possible back then. Learning the new widgets, language, form editor, report editors, and unlearning what used to be good habits, that are now bad. Instead of going through your code to modernize it, you could use the Shell inside ADE, and then add your tables, forms and code into it. This will get your code, forms, and commands thoroughly modernized.
* Need more Reporting on Development Team: Are you a development manager, and you would like to have a better handle on what your team(s) are doing? The ADE has the tools built into it to give you just that. These same tools can be used by the developers themselves. A typical work flow would be to code for the day. Then have the system run the various methods that review the code and update the reporting, and quick access to your code to the items in the report.
* Standardize Your Code: If you have a legacy application that you have brought through a few decades you have likely adjusted your coding style. Coding style consistency makes reading code much easier. This is especially true when you have a team, and you bring new people onto the team. Standardize code, forms, graphics, logging, and much more come all wrapped up in ADE. The great things is that the extensive manual that comes with ADE explains the standards (i.e. positioning of buttons) the built in forms, and code blocks are all there ready to quickly use. Your team is now all on the same page (though some grudgingly). Within a month everyone will be used to the standard and much quicker at reading code, understanding code, and putting good quality code in.
* Need Many New Features: There are many features in the ADE that you may not have in your code. This is a collection of features that we have put together over 52+ years, developing for diverse clients, and having to meet multiple different standards across many countries. Some examples might be:
  + Clairvoyance
  + Flexible date entry
  + Logging
  + Audit Trail
  + Extensive Password Control
  + Individual user modifiable interface
  + Individual user difference in security, what they can see, or modify… to name just a few
* Quality Assurance of Code: This is an important aspect that I think most development managers have struggled with over the years. When you have a team how can you be certain that all the expected features are there, all the code has been written, that it follows your standards, and that it works. The ADE will help you move toward this goal a long way. Having reports to management, dialog that identifies the deficiencies to each developer help you to know when your project is going to reach that point of delivery.
* Identify Speed issues in your Code: Other than the length a help dialog is on the screen I have never seen a user complain about to much speed. The more a user gets used to the application that faster they need it to run. Knowing where the speed problems are, is important to be able to target. Once you have the methods identified, you can target those methods to speed them up. There are almost always things you can do in special situations to speed up the code. The problem is this often takes time, so it is important to know which methods are slow, so you do not waste time on code that does not need speeding up.
* Who Might Not Want it: A person, or team that already has well defined standards, and extensive code. We recognize that we would not have changed our mature code and standards to adopt a good tool. That would just be too much of a disruption. Therefore to this end, I am looking to add in the feature of letting you customize the code contained in the ADE to your standards. You will also be able to create and add in your own AC rules to report on. We have this in our target as we move all of the ADE into a component and v18. We will be looking at what v18 might have for us to this end, and use those new ways of doing things to help us achieve this.

All Done with 4D

* This is all done with 4D: There are no plugins. This means that you can just work with the 4D Code you know, and new commands in v17/v18 to do what we have done. It means that you could do this without purchasing any products and do it your way.
* All done in 4D v17: This means that if you have an older project and you do not want to jump into ORDA, or have all 64bit OS, you can do what I have done here in v17. On the other side of this coin, it means that there are likely more things we will be able to do in the next version of the ADE beyond this with the new features in 4D v18.
* Moving all of ADE to a Component[[3]](#footnote-3): There are a few reasons that moving our ADE from a Shell, to an component will do for us and for you.
  + - Running many of the utilities of ADE in a compiled version should show substantial speed improvements. This is especially true for the QA code, Code Changing Code, Method parsing code. It should also be true for our ADE dialogs.
    - Moving to a component will permit us to put all of our Shell code into a data store in the component, not in your application. That will make it cleaner for you for those methods you do not use. It is also going to provide the framework for us to permit you to add in your code into the ADE interface. You will be able to also mark code we have as not visible so that your programmers do not use our code. This permits you to customize the ADE more.
    - Another reason for the component data store version will we will let you set up multiple preferences that we will support in the ADE. This could be how you name your local variables so that our declaration method will know how you name them, and thus will recognize them and declare them correctly.
* Real Lesson for You: As this is done all in 4D, you can write your own. This presentation is designed to show you some of what we did, and then give you snippets of code to show you how you can do this for yourself. I expect many of you will do a bunch of this, while others do not have the resources to do this. They will prefer to purchase the ADE. It is your choice.
* Many New Features: With the enhancements to the language to manipulate methods, and now even create forms via text the possibilities for the ADE is only expanding greatly. Since we have a mature Shell, my hope is that by 2021 I will have ADE present me a dialog where I define the table name and fields, which fields will have a predefined list, who can modify the list, and the ADE will then create the table, create the forms I requested with all the wonderful interface features, the appropriate code will be added to all of the ADE shell methods. This will make the writing of an application extremely faster, with high quality code, and be rich in feature set. This means that when developing a new application, writing all of the standard things for the application will be done in a few hours. This includes all tables, lists, forms, and code for handling adding, modifying, deleting, record locking, security, users, etc. will all be done in this time. Then the application specific code will be where you will spend your time. For the client they will then be getting great value for their money.

How To Use ADE Code Demo

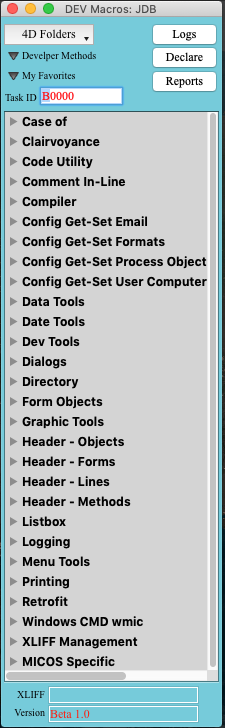
To run the ADE Code Demo, launch 4D v18 and select the ADE.4dbase file. This will run in uncompiled mode so that you can have full access to the code contained in it.

The ADE Demo will open up a small dialog where you can see the demonstration of the code contained in the demo. Click on the button for the feature you want to see.

If you would like to see the code, click on Open Design button. This has a trace command in it. Once you see the code click on the stop execution and modify code icon.  To see the code that button has, go to the File Menu – Open Code and select the button name. The starting method will be opened on screen.

That is it, that simple to use. My hope is that you will find the demo database simple to use, a simple demonstration, and the code contained easy to understand, and for you to grab and implement in your projects.

Dev Macros

This is the first dialog I want to describe. This was the first dialog we created in writing our ADE. 4D has a rich language, but to do things one needs to put together several language commands in specific ways to accomplish tasks. Do not get me wrong, I like this way that 4D has done this. After spending years working around the world, I learned quickly that not everyone wants the same thing done the same way. If 4D were to dictate the appearance of buttons, or dialogs, etc. without us being able to make our own, most of us would not have chosen 4D. So 4D works in that area of coding from low level, up to where we need to work. Sure they could offer complete finished feature groups, but we need the choice to use or create our own.

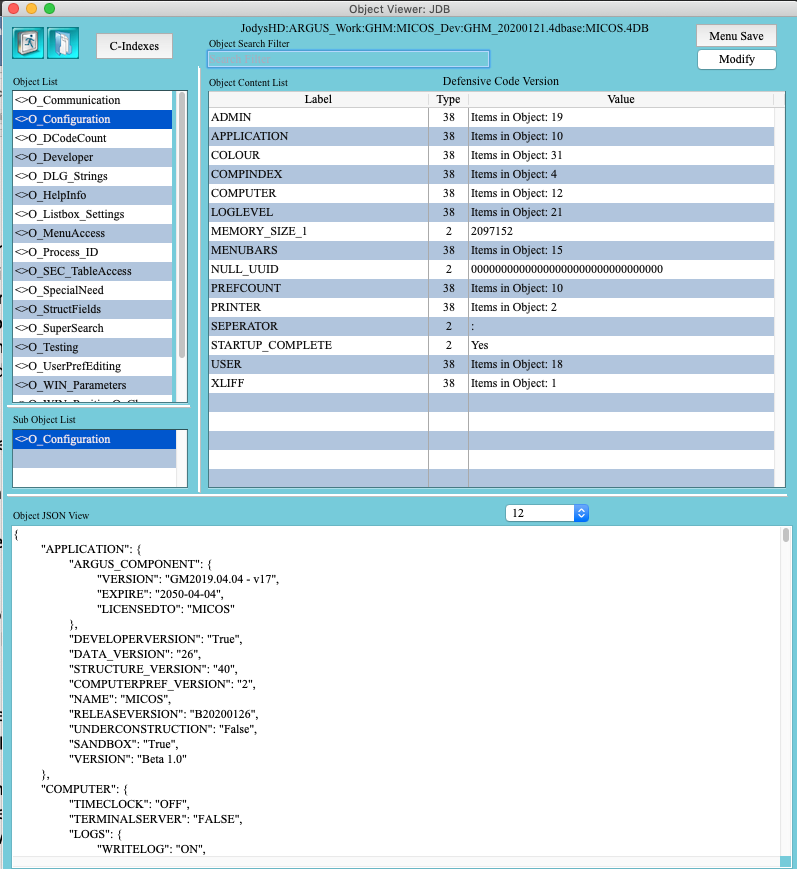
The dialog that I currently have in the ADE has over 100 code blocks. I have a couple of these that I have also placed inside the 4D Macro file to place in with a few keystrokes. This is a great feature of 4D that has been around for years. The problem with this, is there is just no way that I am going to remember the name I have assigned to all the code blocks I have available to me in the ADE. Therefore an easy way for me to get this is through a hierarchical Listbox. I have them in logical groupings to make it easier for me to know where to select the code block I want.

As we used this dialog, we started adding a few more features to it, and now we are removing a few as they are taking over in another ADE dialog. To have an organized way to go through this dialog I will go top left to right and down – as westerners read the page.

* **Button with Menu of Directories to Open**: 4D has various directories that it stores information in. For most work you do not need to know where these are. Things just work. When a problem occurs, you are typically busy with other things. Having to remember what is stored where could be a challenge. Looking it up in the 4D Documentation is a real pain when in a crunch. Click on the button, the list of 4D Directories are shown. Select the one you want and it is opened on screen. Sure easy code - but it is done, and right where you do a lot of your work. This will quickly open a Finder window for the directory containing the following items:
* Active 4D Folder
* Data Folder
* HTML Root Folder
* Licenses Folder
* Resources Folder
* **Developer Methods**: This opens a selection List dialog that lets you select from any number of developer utilities method you have written (or that we wrote for our use). This is most useful when starting the development of a new system. There is no data, you need to test with data, change data, add data retest etc. Lots of good tools in this one. The scrollable dialog’s shows the choices of methods. Note that this is the standard Selection List Dialog in the ADE. The user can find what they want using different ways: scroll to find, Click on the Letter tab of what the item starts with and selection is limited to those items, Type ahead using find anywhere in the line (not starts with), this limits what is shown on screen. Also note, that in our ADE, you can set the number of items, where a popup menu will either show a popup menu, or after the count of items is beyond your preference it will show a Selection List Dialog.
* **My Favorites**: As we created the Developer Methods screen, we quickly realized it was so very useful. With multiple developers, we quickly wrote many little tools we used when developing. Being speed and efficiency freaks like developers tend to be, we wanted to find some of these tools even faster. Therefore we created a popup Menu that will show the signed in developer’s favorite Developer Methods. Each Developer can select from the Developer Methods their favorites and place in this popup menu. We even made it easy to modify this menu by having an item that opens the code that modifies it.
* **Logs Button**: The ADE will create multiple types of logs. Some of these logs are only created during development, others when ever logging is turned on, and an event to be logged occurs. This opens a finder window to where the logs for the application you are working on are stored. No clicking through directories to get what you need. No trying to remember where are the logs stored on this OS.
* **Declare Button:** This button gets used with almost every method I write. As we very rarely use process variables, and all interprocess variables are declared elsewhere we use a lot of local variables. When creating a new method, adding new locals, or removing locals it can be a real pain to add or remove compiler declarations. When in a method, Clicking on this button will go through the whole method and determine what local variables are used, and rewrite the compiler declarations for this method. Done. Saves so much time! The code is clean, and precise[[4]](#footnote-4).
* **Reports Button**: In Dev Control area there are many reports that we can generate. Clicking on this button opens the directory in the finder where all these reports are stored for the application you are working on. It is likely that with the next update to the ADE, this button will move down into the Dev Control dialog.
* **Hierarchal Listbox of Code Chest**: As this is a hierarchal Listbox it can be collapsed or expanded in whole or in part. I typically work with it all collapsed, find the heading, right click on the heading and select the code block I want[[5]](#footnote-5). Doing this will paste a simple consistent comment line or lines, or an extensive method into the open window of the Development Environment at the position of the cursor. I have worked on multiple projects for different clients. When I need to produce something I know I have written once, or many times before the first question is where did I write that. It is usually better to copy proven code from somewhere than to write it all over. This is especially true when using a ADE that has a tightly integrated shell in it. The problem was always where did I write that, so I can go and copy it over. With the DevMacro Palette I can look through the headings, Right Click the category and select the method / comment I want. I then look at the code and see if there are any highlighted areas I need to write specific code for this method. I then replace the place holders with the code that is needed. The code is written very generic since it is a shell that we use over and over again within each application, and across applications. This makes the code more robust, and less coding needed. There are currently well over 100 of these methods that you get with the ADE. Of course, within these code blocks are calls made to even more methods to do specific tasks. Using consistent proven code means consistent behavior for the end users, and higher confidence in the code.
* **Version**: This is an object on screen that shows the version of the application that you are working on. This is determined by the ADE when you load up the application. If you work like us where you can have the current running version, past versions, a Bug fix version, and a longer term version that you could be working on, this is a quick visual check that you can make to ensure you are in the version you are expecting.

Object Viewer Palette

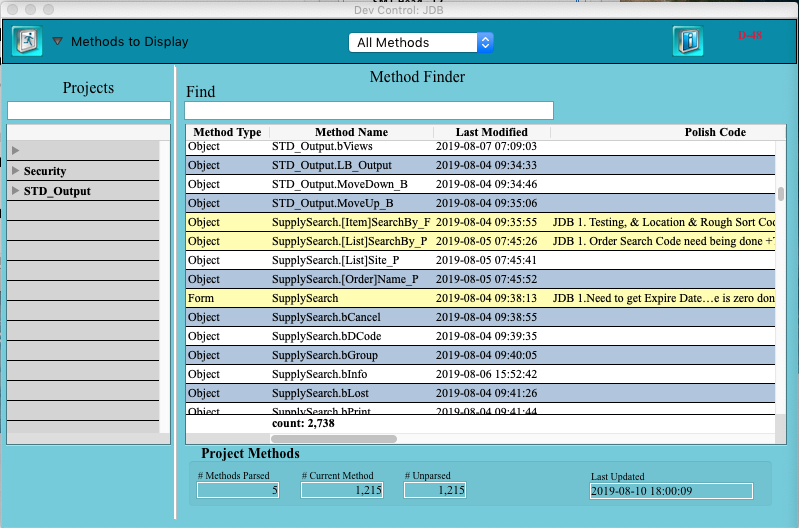
When we started using ObjectTools[[6]](#footnote-6) from Aparajita in the early 90s we quickly came to depend on it. It freed up our creative minds to do what we needed to do in so many ways. I cannot imagine programming without objects. The one thing that was frustrating was to be able to see was what in an object. I learned that right away when an object was not working. What was in the object. Therefore back in the 90s we wrote an object viewer. That was a simple viewer, that with this version much has been enhanced, and we have utilized language commands that became available in.

This is not really an replacement, or alternative to the debugger display of an object’s values which was a welcome addition to 4D. The display of the object values is much cleaner, structured, and can display more than the debugger can show. This shows a set of predetermined objects. This is can be displayed in interpreted or compiled mode. Being able to see these values in compiled mode can be a great tool for assisting to track down the problem in compiled mode. These objects can also be sent back to the developer so they can see what these values are when a problem is occurring. As these objects control a lot of the functionality in the ADE developed application knowing the values contained in specific objects is key to eliminating an item as the source of the problem, or identifying the problem.

* **C-Indexes**: Clicking on this button will procedurally determine what are the Compound Indexes in your current structure. They will then be listed here. The code used to get these are SQL commands. Not hard code, but SQL is not my ‘native tongue’ shall I say.
* **Menu Save Button**: When in the object that the Menus Access is stored this button will appear. It will save to the Menu Object, and to the record for the currently signed in user the preferences as shown on screen. This lets you update the menu options when you are adding new menus[[7]](#footnote-7).
* **Modify**: This button permits you to modify a specific Object Pair (Tab and value), or a whole set of values that are displayed in the Object Content List. This is typically used only when creating a new database, where you are adding in new objects. Once you have the database going, there are tools in the Dev Macro that are more powerful and do many tasks including this.
* **Object List**: This Listbox shows all of the objects that are displayed in the Object Viewer. With the ADE we have this set up to display all the ‘working objects’[[8]](#footnote-8). When you click on an object all of the top level tags are loaded into the Object Content List array. Other information as displayed is shown as well. This is an array based Listbox. Notice as well, that the Sub Object List will have the Object list Top level item that you clicked on in its list. At this level you can then look down and see the full set of tags and values displayed in the Object JSON view. In many cases this is as deep as I go, because I can see what I need at this level in the Object JSON view quickly.
* **Object Content List**: This is the Listbox mentioned in the Object List point above. Clicking on a row will load the Object JSON View below with all the tag – value pairs contained in the sub object you clicked on. You can right click on a row in this Listbox and have further options (some specific to the type of object). Standard options are:
* Add Item
* Delete Item
* Open Object: This permits you to dive deeper into the object.
* **Modify Button**: This button lets you modify the values stored in the Label and the paired value to go with the label. It does this for all objects contained in the Object Content List Listbox.

Dev Control Palette

The Dev Control Palette is where we have placed a set of tools that is related to the monitoring, management of developers. This is intended to be used by both the developers themselves, and their managers. The reports and data generated is stored within the specific applications ADE datastore. This means that you can have multiple applications each with their own Dev Control Data. An obvious enhancement would be if you have multiple data bases and a manager needing to see this overview & detail, this data could be exported from each to another database, or write an application that would let the manager view the data within each projects ADE from the manager application.

* **Method Finder**: 4Ds method Finder is a better method Finder from 4D’s built in tool. Now that I have been using this for months, I have using not been using 4D without it. I now find the 4D Method Finder too frustrating. Our Method Finder has a few advantages:
* It finds methods using a contains search, rather than begins with. I know there are some methods that I keep forgetting the first word in the name, yet there is another part I always remember. When the search is performed, the Listbox hides those that do not match. This makes the finding much easier to deal with.
* With the 4D Method Finder you need to ensure that if a method is highlighted, that what you are looking for is below this point. Otherwise it will not find what you are looking for. With ADE method finder each time you type a letter the search is conducted on all values.
* With the ADE, you can type slowly and it will work the same as if you type fast.
* It searches through Project Methods, Form Methods, and Object Methods. This therefore permits you to bring up a listing of all Form Object Methods, or the Form Method, as well as the project methods.
* The Listbox footer shows the count of methods found. OK – I’m a stats geek and like to know these things.
* There are multiple columns of information. Type, Name, Last Modified, Polish Code[[9]](#footnote-9) comments, # of lines of code in the method, just to name a few of the columns. Depending on your needs and what you are doing with the database the other columns will be more or less interesting to you.
* The Project Methods area shows some interesting statistics. An important few to know are: # Unparsed, and Last Parsing date. Of course if you have the auto parse turned on for say each night the last Parsing date is less of an issue.
* Like the 4D Method finder, just double click on the method name and the method is opened up on screen for you.
* **Polish Code Column**: When the Method Finder is filled, it parses all methods for specific information. In all methods created within the ADE, we have a section called: ‘Polish Code’. When I write a method I may not be able to, or want to write all I have planned for the method right away. This may be a time restraint, or I know I want to add features at some time in the future. Instead of making the developer enter this in some other area, or application (thus it will not get done), the developer just makes notes in the ‘Polish Code’ area of the method. The Method Finder will get this and put this text into the Polish Code column of the Listbox. That row will have a different background colour to clearly show it is a method that still has outstanding work to be done on it. These can be viewed by developer making the Polish Code comment, or for the whole project. This way it helps to keep track of outstanding tasks that need to be done. As a programmer, or development manager you can have a high confidence level that the expected code has been completed. When using this area, I set the dialog to display just my Polish Code comments. It can then double click on the row, bring up the method and enhance the method thus reducing the Polish Code comments, or removing them totally. When I see that a project has zero Polish Code comments I have a much higher confidence that all code pieces have been implemented.
* **Method Compare**: This is the tool we developed to compare previous versions of the same method. ADE automatically saves a method to disk in our specialized directories when the method is modified and closed. This will bring up the current and selected historical version of the method of interest. It will highlight differences between the methods. This part of the tool does not have all the features of other commercial tools. We need to decide if we will continue to develop features for this within ADE, or have it launch a selected tool and do the dif with the other tool. There are pros can cons of either way. This module was done in v16 of 4D, so well before 4D announced their intention of moving to all methods being stored as text rather than binary. Therefore this tool gives you many of the features of the project way of developing while keeping all of the binary way of developing an application. We had been doing this with 4D v12, but with the limitation of 4D, it was only for project methods. The language supporting form and object methods is a welcome addition to the language.
* **Quality Control**: The QC module should be in every developer’s tool kit. This tool has a display that shows each method that does not meet various standards you have created for your code. Of course this is filled with a lot of our standards for our shell code. When the method is run, it examines all object methods, form methods, and project methods. It checks for various aspects of the code that our standard needs to be written a certain way. For code to check your code, it means you need to write your code in a structured consistent way. This includes comments (as they can help find specific areas of code). We use this to go through all our code and highlight anything that does not match our standard. Once I wrote this and ran it on the project I was doing, it was the first time that I could really say that I looked at every method to check that all the code met our latest standard of doing things. When the QC method is run, it fills in the Listbox with all the problems it finds. To fix a method just double click on the row in the QC Listbox. The method opens. The QC listing will show the exact line number it found the problem in, or what it is associated with (i.e. when a certain line is missing). You then fix up the code, and move onto the next item to fix. When I am getting ready to deliver a product, I will go through this process. If this is a habit to run, it will not be onerous. The first time I did this on a project it took me about three days of going through the identified methods to fix them. This was on a project that I had developed while enhancing our Shell, and modifying our standards. Therefore I expected there would be many. The ability though to scroll through and only deal with the identified methods, double click on the row to open the method makes this process much smoother and faster. Of course work on it, run it, fix and repeat until they are all cleared up. The QC method indicates which developer is working on the method. It will then assign the QC task to that developer. Therefore in a multi-developer arrangement each developer can fix up their problem code. If someone is now away, another developer can go in and fix that code. As a Development Manager, this gives you a much better sense of the quality of the code in the application you are developing. Coming soon, you will be able to have the QC method run at a set time each day so that at a specific time of day you could have this report available for review and assignment.
* **Time Log**: The time log feature logs the developers time. This is outside of the planned demonstration at the 4D 2020 Summit, so I will not cover it anymore here.
* What Work has Someone Done: As my mind got thinking on what I was doing, I remembered back to overseeing development with multiple developers. For those that have done this, you have those you just trust (rightly or wrongly), but there are some you are sure are not putting in the time and effort. The problem when I was so busy with too many hats, is that I really could not monitor this everyday by just looking through code. I started to think about how I could generate a report that let me know who did what on every day of a month. It could be done for monitoring billing, work performance, productivity and tasks worked on. For the ADE I started to generate tab delimitated files that I could import into Excel. These files were generated from the comments created in our standardized code. I could quickly see how valuable this was from a billing perspective, developer management, and project management points of view. With these files I could have a very good idea as to who was doing what, how many hours they coded a day, what they worked on, and does it match their time sheet. All part of management.
* **Feature and Bug List**: This list all of the feature to be added, and bugs to be dealt with for a project. This is outside of the planned demonstration at the 4D 2020 Summit, so I will not cover it anymore here.

Demo Application Code Comments

In this section I am making further comments on the demo code. I am putting these comments here, so that the comments in the code are not ‘obstructing the view of the code’. The comments here are related to the Button Name on the ADE Code Demo dialog that comes up.

Declare

This code could be the code that every developer will quickly recognize as the biggest time saver. I know that when I get heavy into coding and thinking through the process, when I add or remove a local variable I will often forget to immediately update the compiler block at the top. I personally hate code that declares variables where they are first used for many reasons. That of course is my preference. Either way you still need to ensure you remove, or add the declaration statement no matter where you put those statements.

I have rewritten the Declaration code to a simpler version from what is in ADE. Our ADE Declaration code is more sophisticated in what it can handle in the declarations. The structure of your method code will make your work doing this easy, or harder. I did this as some of the code in there was to handle special cases as we moved code over, or as our standards changed. As I have thought about writing this to work with other coding standards I recognize that the code needs to be more sophisticated. We will do that, to make the ADE more attractive to other developers. The goal was to show how you can write code to declare all the local variables in your code. You can take this code and then adjust you’re your standards, and also to handle special cases you have in having multiple generations of coding ‘standards’.

The code was written with the understanding that the developer is writing code in a specific method. They know (or think they have) added and/or removed some local variables. Since they are working in the method, this window will be front most in the development environment. The developer then moves to the ADE Code Demo window and selects ‘Declare Locals’ in the Declare button. The whole method text is copied out, processes, and any old declarations in the ‘compiler declaration area are deleted, and the new declarations are put in.

Key parts of this code are:

* Getting the Window ID of the front most window in the development environment.
* Call Get Window Rect & Set Window Rect for the window otherwise Post Event does not work
* Using Post Event to select all, and then copy all the text from the method
* Processing the method, finding all the local variables (determining their type), declaring them
* Removing the old declarations, inserting the new declarations
* Copying the method text in memory of the method into the pasteboard
* Ensuring all the Method is selected, and pasting the new method text in.
* You must have a code recognizable variable naming convention. If not you have a lot of work to make is so. It will be well worth it in the end.

This is great piece of code for you to implement in your system(s). With this you learn how easy it is to read the code, and modify your code via code. From there, you like I have been, can get excited about the possibilities, and start to spend time automating your coding. I hope that team at 4D will get excited about this, and give us even more tools to do this. With version 17 they gave us some more, with version 18 they have given us more. I have not had enough time with version 18 with all I have on my plate to explore the new features that I know of.

* DeclareLocals\_TestMethod: In this method you get to see more of what our standards for writing our method code has been. We have very structured locations in our code, that are identified by the means stated below. By doing this we can find what we need, and insert code. In this case it is simply where we need to insert code, and where we can ignore local variable text.
  + Comment Headings
  + Code with specific comment
* In this method when you trace through the code, you will see that 4D puts line of text in the actual 1st line of code. Their method editor hides this. That is fair, it is information they need to handle the method. In doing this work, we delete the line, write the code, and 4D automatically adds the line back in, just as if the method was being written for the first time. Works well.
* Each of our heading areas are used in our automated areas. Therefore we have a purpose for them. From my work with 4D v18 thus far, the only thing I seen us changing is that the start of the method comments that we write, until the end of the Assumptions area we will use the new comment delimiters: /\* and \*/. This will let us hide comments, especially long ones. This can remove one of the arguments used against well documented code.
* Any variable that was previously declared, but not in the method now is removed – well actually not declared in the new declaration block.
* The declaration block is sorted alphabetically so that you can find the variables in the block easier.
* All comments are retained in the method.

In longer methods this can take a few more seconds. If you are like me, you will need to have a progress indicator to let you know that things are progressing.

Paste Code

This demo shows in a very simple interface how you can paste blocks of code into other methods where your cursor is. As you build up your standard code that is generic, this becomes more valuable. Why should you have to rethink the writing of the wonder generic code each time, or have to find where you used it before to find it, copy it, and paste it. Use this type of a tool to just select it and it is done.

When I use our DEV\_Macro tool I find that my mind can stay concentrated on the code path I am writing. I need a Popup Menu, an Confirmation Alert, Dialog to let the User select a built in report, etc – I do not need to switch to that code writing. I just select it from our Dev Macro Tool and it is pasted in.

With this it is extremely quick, I do not need to remember what my macro starts with (Macros in 4D). When the code is pasted in, it is designed to highlight what parts needs to have me enter something specific (i.e. Alert Text).

With our ADE, we have designed our DEV Macro interface that presents this in a compact, but informative fashion. Adding a new method, or changing an existing one is very easy. In fact there is a Dev\_Macro for creating a new Dev\_Macro. Why shouldn’t we write code that writes code! This helps us to concentrate on the new stuff, and let the ADE create the standard stuff.

Key parts of this code are:

* Having a standard storage place for your Code to paste in. For this simple example I have created a method that puts this all in memory in arrays. This may or may not be the best solution for your needs. There are several possibilities that you can use for this:
  + Stored in an object
  + Stored in arrays
  + Stored in text files on Disk
* The code to get the Window you are working in, in the Design process
* Get the code you want from your storage into the pasteboard
* Using Post Event to paste the code into your method where your cursor is

Object to JSON (Making Objects Easier to Read)

In our ADE we store several objects that hold many different settings as to how we want the application to work, where in the application a form is add, Security Settings, Backup settings, and many other things. By doing this we can control 4D and our application in ways that are not native to 4D. By having all of these in an object instead of say hundreds of variables, it is easy for us to have a dialog that will let us find the specific setting we want. This can be done in a compiled application, we can have the application send all the settings to us (for trouble shooting), a sophisticated System Admin can see the values too. Each object we create is like a theme. Within the theme are groups of settings. Objects that we can build and extract information using dot notation is perfect. We have been doing this since the 1990s.

We use these objects for User preferences, Screen Use Counters, Help Info, Listbox setting for the current user, a Virtual Structure, SuperSearch Settings, and saved searches for the user, and many more. Once one uses this way of accessing configuration data it is just too easy to do it this way. Some of these objects are filled at startup and stay there. Others are switched out depending on who is signed in. Others are unique depending on the computer the application is running on.

In the Demo database I am presenting a very simple interface, with simple examples. This lets you see the critical code to accomplish this. Once you have this tool in your application, if you use objects (doesn’t everybody?) you will not be able to work without it.

In the demo I created only one object. The object is created in the Database Method: On Startup. This method calls the method Startup\_CreateObjects. It creates a bunch of configuration settings we could have in a database. The point was just to create and object that has nested values.

In the form I created a Listbox that just holds all the object names. This form is the foundation for an Object Viewer that you can have open while developing, and while in compiled mode. Beside this is a text object that will hold the JSON formatted text that 4D can produce quickly. Clicking on the object name in the Listbox will generate this text.

The view of objects (nested or not) is so much easier to read and go through than what is displayed in the debugger.

Method Explorer

Our Method Finder tool came about because I wanted to display information that we had stored within the method (PolishCode[[10]](#footnote-10)). As is typical starting with one feature and then using it makes for adding more features. PolishCode is fantastic reason for this tool for a single developer, or a group of developers. What I am showing here though is another feature that every developer can use. Essentially it can take over the use of the Method Explorer window totally.

The Method finder gets a Listbox of all of the Methods (Project / Form / Object). These are displayed with a column indicating the type of method, the method name (just the specific name minus the full path), full pathname of the method, Polish Code, and several more nice to haves.

There is a search object above the Listbox. In this object you can type as slowly as you want. As well, the find is a contains find rather than a starts with find. Though we have rather strict naming conventions for our methods, I sometimes just cannot remember what the method starts with. I do however know the intent of the method. I just type that into the Method Finder search and often get a nice short list. No rushing to type so that it will find what you want.

Once I have the method listed I just double click on the method and 4D opens the window up on screen. Of course 4D opens the window up on the screen where it last was. With our Method Finder you can set up all sorts of parameters so that you can have the newly opened method window open in specified locations, or specified size. This can include ensuring it opens on the right ‘screen’, and is not off the screen.

With the method finder you can have all sorts of other valuable searches. For example, finding all methods that I wrote, that still have PolishCode comments in them. This helps me as a developer find what I need to get done for the project. The nice footer that tells you how many methods are found also gives you an indication of the amount of work ahead of you.

To see a simple version of the method finder just click on the ‘Method Finder’ button. This will open the dialog, and load all the names of methods contained in the Demo application. The only column I have put in this Method Finder is the method name and the full path name. As we did in the ADE, you can add more columns, and more search features so that you can accelerate your development, and quality.

In ADE, we have a way for the developer to add notes into a method of what they are planning to have the method do. That way they can add to the notes at the start of writing it, and anytime in the future. As they finish features they remove those notes. Any feature notes that are not removed, cause the line in the Method Finder to be highlight in yellow, as well as displaying who it is assigned to, and what is outstanding. We have the ability to search for all outstanding features, and all outstanding features assigned to a specific developer.

Quality Control for Coding

When you have thousands of methods, and hundreds of thousands (or millions) of lines of code how can you know that all your methods are meeting minimum standards. The compiler of course helps with syntax – a critical part of the code.

You or your company can have standards as to documentation, that specific snippets of code should be there, have you missed any lines of code that are to be in a type of method. If your company changes code standards, how do you know that all the methods have been updated to the new standard? With this type of tool you can validate that the types of items identified below are being met. This raises the bar on the quality of code, and thus quality of application.

At ARGUS we have been creating methods that write code. For example, when needing a new table all we do is present a dialog for the developer to enter the table name and fields. From there the ADE, creates the table with the ARGUS required fields, the fields specific to the table, and writes code that needs to be placed into various other standard methods. To do this successfully you need to have excellent coding standards. This includes code style, documentation, and standards. When you have this it makes writing Code to Write Code (CWC) much easier, and more reliable. This is were we are really pushing ADE as this will make our custom work costs go down greatly and reduce bugs. Being able to do this makes us much more competitive in the market place against all types of developers.

This is what this module is aimed at. For the 4D Summit 2020 Chicago I wrote just a small example of doing this. There are more things that you can do with your method review than I have placed in here. This just 3 types of validation.

* **Style**: Programmers are a fuzzy group. Each of us has our own styles. When we hit code that does not follow our style it slows us down, and frustrates us. We are less efficient. This can be many things. It can something (that I think) is small - # of blank lines. I discovered that I made many methods that would have 2 or 4 blank lines between blocks of code. Likely I put more than one or two just to give me a clear white space showing a different logic block. Well that frustrates some people. Having an agreed upon style helps everyone working on the project read and understand the code, and to get to the pieces of code they need to look at.
* **Documentation**: I admit, I am fussy about documentation. I think that I am fussy because I have only worked with one programmer that did documentation well. The rest – well they frustrated the heck out of me. I would go into their code and find zero documentation. Now instead of just reading 3-10 lines of documentation, I had to read through the code to get an idea as to what the method did. I am usually too busy to ‘waste’ time like that. Therefore we developed standards for documentation. I admit, this is not a cure, but it helps educate the worst offenders.
* **Standards**:Depending on the type of method, you can have different standards of what needs to be in the code. For example we have logging, log level settings, timing, Dev Asserts, logging, and more. This type of requirement can be validated in the code to ensure it is there in the appropriate type of method. With this one, you can get very sophisticated in what you check for. If a specific type of code is encountered, you could look for other things that should be there. This is really an ongoing effort to add standards.

For this demo the Quality Control button is used. It opens up a simple dialog with a Listbox on it. To see it work, just click on the Process button. This will run against one method made specifically for this part of the demo. The method is called: QC\_DemoMethod[[11]](#footnote-11). I typically have a method that I call for each general type of method. From there I can have further subtypes, which will be determined and another method call. In this case all the checking is done in the method AC\_Project\_Methods.

In the demo, I have the problems found, immediately placed into the Listbox. You should plan to write this to storage location, and then load from there when you open up your QC Observer dialog. You will likely want to have the QC\_Method run over night for your applications.

Open Design

This is the button you need to click on to get the Design Environment open. Once you have done this, then you can use the File Menu to look at specific code.

Quit

Used to Quit the ADE Code Demo application, and code resource.

1. Jonoke – Jonoke Software Development Inc. The company I started in 1988, and incorporated in 1992. I left Jonoke in Oct 2013. [↑](#footnote-ref-1)
2. Really that is what we all do in programming for our customers. In our case though, we were our customers for the shell we were creating, that evolved into an ADE. [↑](#footnote-ref-2)
3. There is an Argus Component with the v17 version of ADE. This component is very small, with some utility code moved into it, and our licensing code in it. The later, of course is the reason that this component exits. [↑](#footnote-ref-3)
4. Until we move ADE into a component all locals need to be named in our standard. When we move the ADE into a component, we will have a separate datastore so that you can set up your own preferences for the naming of the local variables. [↑](#footnote-ref-4)
5. The other way to work with this is to expand the group, then double click on the code block you want to insert into the open developer window at the cursor. [↑](#footnote-ref-5)
6. This is a 4D plug-in available at aparajita.com/shop/. In Aparajita’s comments on the OjbectTools product he states: “If you are a current user of ObjectTools, I will continue to support it for the foreseeable future. If you have never used ObjectTools before, then you are better off using 4D’s native **C\_OBJECT**.” That is good advice. [↑](#footnote-ref-6)
7. Menu Save: This is an example where a future release could enhance the ADE so that a Developer Tool in the Dev Macros could add the menu appropriately, add the menu access to the current developer in both the object, and in the Developers Menu preference records, and finally write the code to automatically add it to the options for all users. This is all about saving developer and implementation time. 4D now permits us to do all of this. [↑](#footnote-ref-7)
8. Working Objects are objects that we have that will hold temporary values that we need to manage the operation of the application in a specific time limited manner. These are limited only by your imagination and need. They could be values to indicate that a specific page within a multipage form has had its objects filled. Therefore when the user clicks to see that page from a different page of the form they are not filled again. This is a methodology we use on multi-page forms that would be too time consuming to load before displaying the form. The first page can be shown quickly, and then the additional pages only shown when the user clicks to see the page, or they could be filled in the background through a worker task. [↑](#footnote-ref-8)
9. Polish Code: See more on this lower down in this section. This is another fantastic feature for ensuring that all the planned code is written in every method, by everyone. [↑](#footnote-ref-9)
10. **PolishCode** is pure text at the top of the method that the developer writes to indicate what further changes to the method need to be done for the method to be completed. This could be for a need for other areas of the application, the current one, or some future idea. The challenge is how to manage this intended code at the micro level (the method). How to ensure that it gets done. Documentation is great, and external project management tools as well. The problem I have always encountered is to get developers to take the time to update the data in these external tools. I count myself in that group of developers. Therefore over the years I have always driven to have documentation, and management tools data be as close as possible to the developer’s work. Nothing can be closer than within the methods they write. [↑](#footnote-ref-10)
11. AC\_DemoMethod: If you go in and adjust this (as you might want to do), there is a method called: QC\_DemoMethod\_Backup. You can copy and paste from this method back into QC\_DemoMethod to get the ‘factory method’ back. [↑](#footnote-ref-11)