Facilitating Abstraction and Reuse with ExpectTK

by Navid Sabbaghi

Software engineering is a discipline of programming aimed at increasing individual as well as group programming productivity. Developing small programs to solve small engineering problems is relatively simple, because those projects usually require only one programmer who can ``hack" the code to work around any troublesome special cases that may arise. But large scale projects demand more careful attention since usually more than one person works on the project. Thus one programmer cannot look at the source and easily change a few lines to fix a bug. As Stroustrup points out, "You can make a small program work through brute force even if you break every rule of good style. For a larger program this is simply not so"[1]. Large projects require good programming style that enables everyone in the group to understand clearly defined input/output interfaces. Two of the key techniques to good programming style are abstraction and re-use.

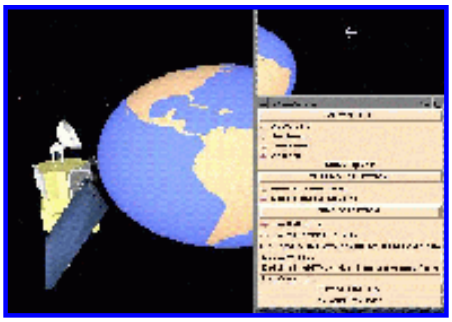
Abstraction shortens project completion time. It embodies the idea of developing subprograms that other programmers can use easily; they don't have to know about the details of the subprogram and the domain of the subproblem. They only need to know what problem the subprogram solves. In building this abstraction barrier, programmers don't waste company time learning about the intrinsic domains of the subproblems. Instead, programming tasks are distributed based on knowledge of the subproblem's domain. Similarly if programmers can reuse a program they used for another project, the completion time for the project will shorten and the programmers will be working more efficiently.

ExpectTK brings the efficiency of abstraction and re-use to life. ExpectTK, a scripting language with a graphical user interface (GUI), reads output from a process and acts appropriately for the given output. Unlike other scripting languages ExpectTK can navigate within processes requiring interaction. The programmer does not have to worry about what internal calculations each of the called subprograms performs. Instead, the focus can be placed on the logic of the overall program and purpose of each subprogram. ExpectTK makes abstraction and re-use possible.

This article will discuss core ideas of Software Engineering as well as features of the "ExpectTK" scripting language which combines the power of the "Expect" scripting language and the TK graphics library; Expect functions like ExpectTK, but it offers no graphical user interface (GUI).

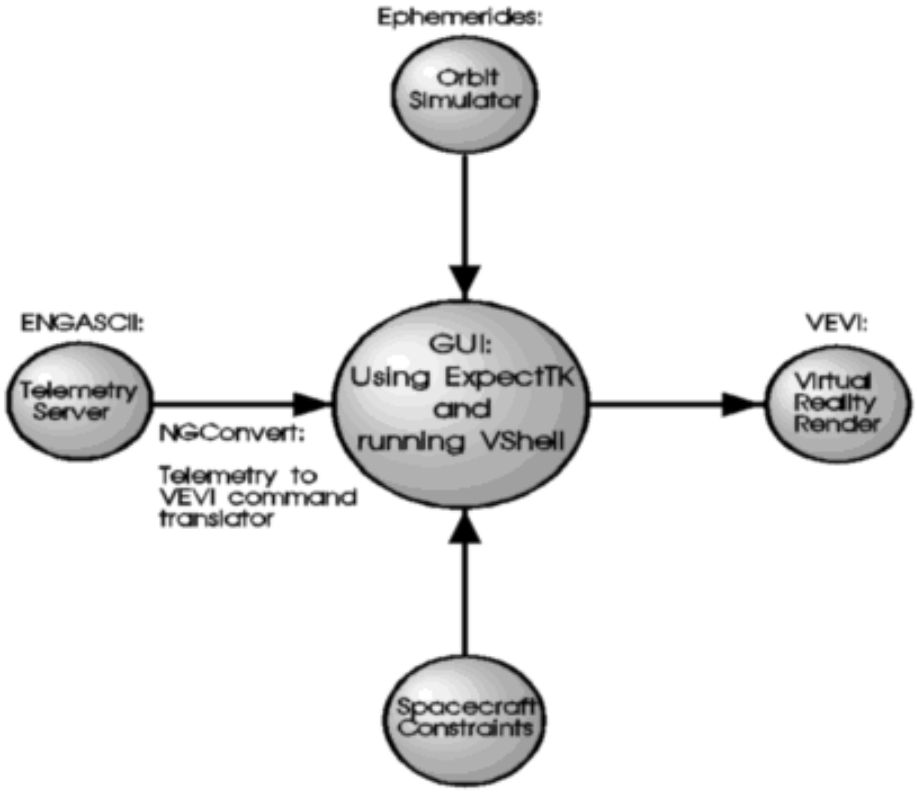
The EUVE Virtual Environment Project

The EUVE Virtual Environment (EVE) Project depends on abstraction and re-use. EVE offers an interactive virtual reality viewing tool for the Extreme Ultraviolet Explorer (EUVE) satellite. It serves as a predictive tool for forecasting spacecraft constraints and resolving anomalies, thereby reducing personnel requirements for payload monitoring, analysis, and science planning. This tool is currently being developed by three groups: the Applied Research and Technology (ART) division of the Center for EUV Astrophysics (CEA) at the University of California at Berkeley, NASA Ames Research Center (ARC), and Goddard Space Flight Center (GSFC). At first EVE seemed like a lengthy project. However, by splitting the work amongst these three groups, the project's chance for completion in the appropriate time frame increased dramatically.



Snapshot of the EVE Project

There are five major programming requirements for EVE: the virtual reality renderer, the telemetry server, the orbit simulator for satellites and planets, the EUVE spacecraft constraints simulator, and a control panel for human interaction. Since the coding was split among three groups, the concept of abstraction became a necessity to maintain an efficient agenda; all the groups needed to work in parallel, and one group needed to be able to easily combine all the elements.



ARC's Responsibilities. Ames' modular approach to the virtual reality renderer, VEVI, was to develop a program that read a configuration file to set up the initial universe. VEVI reacted to spaceball or mouse movement for navigation in the virtual universe, and offered miscellaneous graphics support such as various rendering and viewpoint options. In addition, Ames developed a shell, Vshell, that facilitated message passing to the renderer, so that programmers unfamiliar with the intrinsic details of VEVI could change the state of the virtual universe. For example, it is possible to send messages to VEVI for the thermal shading of certain objects in the universe without having to know the representation of the desired object. Typing `VMC_CHANGECOLOR {objectA} {colorA}` at the shell's prompt, the programmer changes `objectA`'s color to `colorA`. Similary, the outside programmer doesn't have to know how VEVI stores position information; typing `VMC_SETPOSE {objectA} frame {objectB} x {c} y {d} z {e}`, the outside programmer places `objectA` at `(x,y,z)` with respect to `objectB`. The ability to use a tool easily without knowledge of its internal workings is a goal of *abstraction*. The shell abstraction allowed the other groups to develop their own elements without having to worry about how to interface with the renderer.

CEA's Responsibilities. The Center for EUV Astrophysics at the University of California at Berkeley was responsible for designing the telemetry server that computed relevant thermal and position values for the satellite as well as designing the control panel GUI. In addition, they were responsible for developing a spacecraft constraints simulator. However, one of the elements, the telemetry server, was partially completed because it had been developed for another project, Selmon, that performs anomaly detection. Therefore CEA did not have to start from scratch. To save time, CEA *reused* the telemetry server and modified it for the EVE project.

GSFC's Responsibilities. Goddard's major task was to develop orbit propagation software that would simulate where the planets and satellites should be at any point in time. GSFC was also in charge of designing the the 3D model of the EUVE satellite and setting up the universe's configuration file. Again, none of the other groups worried about how Goddard was developing their code because the code could be interfaced with EVE abstractly. For example, feeding the orbit propagation routine the object's name and the date and time, an outside programmer could easily obtain the object's coordinates. This abstraction made it simple for someone who had no background in orbit propagation to feed Vshell `VMC_SETPOSE` messages.

Design Lesson. Clearly defined interfaces promoted efficiency; the groups were able to work in parallel because they didn't have to worry about the details of each other's code. Each of the generic subprograms was written with a high level of abstraction for easy interaction. The groups also saved time because they could reuse modules from past projects. The obvious remaining task was to combine all these independent pieces of code with a GUI that could interact with each module without the programmer having to know how each of the subprograms work, "only what it does and how to use it" [2]. This idea necessitated the graphical scripting language ExpectTK, especially since one of the subprograms, Vshell, needed to be run as an interactive program in order to be efficient. The control panel GUI in EVE was the ExpectTK script that connected all the EVE programming elements. For example, the ExpectTK module let us take any executable, such as orbit propagation routines and a spacecraft constraints simulator, and use the output to make decisions, such as which `VMC_SETPOSE` messages to send to VEVI's shell. Using the ExpectTK interface, the programmer writing the script only has to know about the high level commands for each of the subprograms; this allows the programmer combining all the subprograms together to concentrate on the actual problem and not the coding problem.

The major bottleneck that hinders success in large scale projects is unnecessary complexity. Abstraction and re-use are the keys to decreasing complexity in software engineering. Success on large-scale projects requires the problem to be broken into solvable subproblems that can be easily combined.

Implementing an ExpectTK Script

So the theory sounds great, but how does ExpectTK actually reuse code, employ abstraction, and combine the independent pieces of code? The following serves as a brief introduction to ExpectTK and gives an example of how the above is achieved. The first step in writing an ExpectTK script is to set up your GUI. The syntax for this part of the scripting language is very similar to TK's wish commands for setting up widgets, which means if you are already familiar with the widely used TK wish language, using ExpectTK will be simple. The following is an example taken from file ex1.etk.

```
#!/disk22/usr/local/bin/expectk

frame .view_ports -relief raised -bd 1
label .viewHeader -text "VIEWPORTS:" -relief raised -bd 7
radiobutton .startracker -text StarTracker -variable viewportCMD \
        -value { StarTrackerOn } \
        -anchor w -relief flat
radiobutton .sunsensor -text SunSensor -variable viewportCMD \
        -value { SunSensorOn } \
```

```
        -anchor w -relief flat
radiobutton .telescope -text Telescope -variable viewportCMD \
        -value { TelescopeOn } \
        -anchor w -relief flat
radiobutton .antenna -text Antenna -variable viewportCMD \
        -value { AntennaOn } \
        -anchor w -relief flat
button .takeSnapShot -text "TakeSnapShot" -command capture -width 0
pack .viewHeader .startracker .sunsensor .telescope .antenna \
        -in .view_ports -fill x
pack .takeSnapShot -in .view_ports
pack .view_ports -fill x
```

Each non-indented line is a command. The first line simply calls the ExpectTK interpreter, which the following lines are fed into. The second line declares a graphical frame, `.view_ports`, within which widgets (buttons, text entry windows, etc.) can be placed. The lines after that declare a label (title for the frame) and different types of buttons. Finally, the pack commands determine how to place each of the widgets `-in` the graphical frame; the order in which the arguments are fed to pack determines their position within the frame. The `-fill x` tag makes the buttons and the frame expand horizontally to fill the display, the `-bd 1` tag sets the border to a width of one pixel, and the `-anchor w` tag aligns the buttons on the west side of the frame. The `-variable viewportCMD -value { x }` tag signifies that when the buttons with that tag are selected, `viewportCMD` takes on the value `x`. Also the `-command takeSnapShot` tag for the `.takeSnapShot` button declares that if the button is pressed, the capture function is called.

After the GUI is completed we can implement the functions called by the buttons. For example, if we had a text entry widget that read the RA Dec coordinates the user desired for the EUVE satellite and the widget called `radec_set` when the Return button was pressed, the definition of the widget would look like:

```
#!/disk22/usr/local/bin/expectk

frame .position_selection -relief raised -bd 1       #instantiate another graphical
frame to put widgets in
label .radecprompt -text "RA Dec coordinates"
entry .radec -textvar radec -relief sunken -width 1  #this line declares a widget
that allows text entry
focus .radec                                         #this puts the keyboard focus
on .radec widget
bind .radec <Return> radec_set                       #this declares that procedure
```

```
radec_set is called when
                                                    # return is pressed in the .
radec widget
pack .positionHeader  .radecprompt .radec -in .position_selection -fill x \
       -padx 2 -pady 2
pack .position_selection -fill x
```

Then the procedure associated with the text *entry* widget, `.radec`, could look like:

```
proc radec_set {} \
{
spawn vshell
expect "VshellPrompt:"
send "VMC_SETPOSE EUVE frame SUN radec $radec\r"
}
```

The preceding procedure calls Vshell, which is the shell that is used to talk with VEVI and looks for the prompt "VshellPrompt:". This is where the power of Expect plays a role; it navigates within a spawning process. Once the prompt is obtained, abstraction is employed; the VMC_SETPOSE message is input to the shell by this automated process, and the renderer then does its own magic and moves the satellite accordingly. So the user writing the GUI has to know nothing about how the actual renderer or any other applications operate. ExpectTK can similarly connect outputs from a lot of independent pieces of abstracted code and do if-then looping to execute the appropriate logistics step.

The above is a very simplified example, but an analogous procedure can be used to pipe telemetry data (thermal data, position data, etc.) through a translator, which translates each line of telemetry into a Vshell command. Then each translated line can be fed to Vshell. Also ftp sessions can be automated to obtain new data files for EVE's subprograms. These scenarios illustrate the abstraction barrier, integration, and re-use capabilities offered by ExpectTK; unix commands as well as generic group generated code can be reused easily by simply spawning the program and feeding the appropriate data to it.

Conclusion

Abstraction barriers and re-use are saving the EVE project a vast amount of time. The ideas allow groups to forget about the programming linguistics of a problem and instead focus on the logistics of a problem. Abstraction and re-use can easily be implemented with ExpectTK, increasing efficiency in the group setting.

References

1.
Stroustrup B., *The Design and Evolution of C++*, Addison-Wesley, Reading Mass., 1994, p.6.

2.
Blum B., *Software Engineering: A Holistic View*, Oxford University Press, New York N.Y., 1992, p.286.

3.

Brooks F., *The Mythical Man Month: essays on software engineering*, Addison-Wesley, Reading Mass., 1995.

4.

Fateman R., *CS169: Software Engineering--lecture notes*, August 1995.

5.

Libes D. X Wrappers for Non-Graphic Interactive Programs, *Proceedings of Xhibition 94*, June 20 1994.

6.

Libes D. expect: Curing Those Uncontrollable Fits of Interactivity. *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June 11-15, 1990.

7.

Libes D. expect: Scripts for Controlling Interactive Processes. *Computing Systems*, Vol.4, No. 2, University of California Press, Berkeley, CA, November 1991.

This work has been supported by NASA contract NAS5-29298 and NASA AMES cooperative agreement NCC2-902. The author wishes to thank Tom Morgan for his assistance with this article.