The Significance of Style

1

LabVIEW is a graphical programming language for developing diverse applications in a multitude of industries. The block diagram provides a unique form of source code expression that is dissimilar to most programming languages and development environments. The data flow paradigm represents the program as wires, terminals, structures, and nodes rich with functionality and innovation. LabVIEW extends this innovation to the developer, providing tremendous freedom of expression and creativity. As such, there are many means to an end, or possible development styles, with LabVIEW.

1.1 Style Significance

A given software task might have numerous possible implementations that appear functionally equivalent. What on the surface seems a matter of personal development preference, creative license, or style has significant implications. Developer tendencies have direct effects on the outcome of applications. Throughout my career as a professional LabVIEW consultant, manager, and trainer, I have observed many different development styles and have enjoyed debating the pros and cons of each. I can tell you that style is a sensitive issue for many developers. However, it is extremely important to recognize that good development style is not merely a matter of personal preference. Some styles lend themselves to better performance, source code that is easier to read and maintain, and applications that are more reliable and robust. Hence, I present Theorem 1.1:

Theorem 1.1: *A direct relationship exists between LabVIEW development style and the ease of use, efficiency, readability, maintainability, robustness, simplicity, and performance of the completed application.*

Theorem 1.1 is the foundation upon which this book is based. LabVIEW development style really *is* significant. Few would argue that clean and neat diagrams are easier to read than sloppy ones. Organized user interfaces are more intuitive and easier to operate. But did you know that VIs

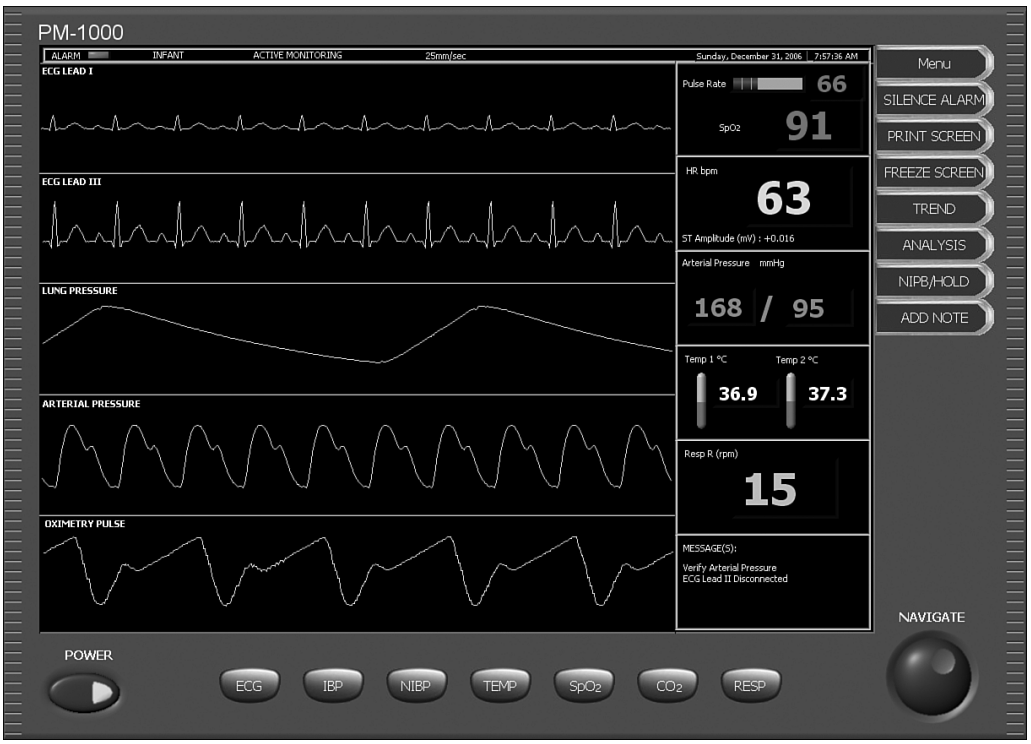


Figure 1-1
Meticulous VI—the front panel and block diagram for a complex application, developed with meticulous attention to detail

containing neat panels and diagrams normally execute more efficiently and with fewer bugs? Do you consider that your applications might need to be operated and maintained by people who are unfamiliar with your coding style? Ironically, this person might even be you 6 months, a year, or several years from now, after you have forgotten your own work. Indeed, I have seen developers confess their inability to explain source code that they developed last week, never mind last year. By contrast, many multideveloper teams that reside in remote locations can productively work together, share, and apply LabVIEW code seamlessly and without explanation. The difference is style.

Consider a few examples. Figures 1-1, 1-2, and 1-3 illustrate top-level VIs created by different developers for very dissimilar application types and complexity levels. However, each reflects the developer's distinctive programming style. Which front panel would you prefer to operate? Which diagram would you rather modify and maintain? Which VI executes efficiently? Which VI is more likely to have bugs, race conditions, or memory leaks?

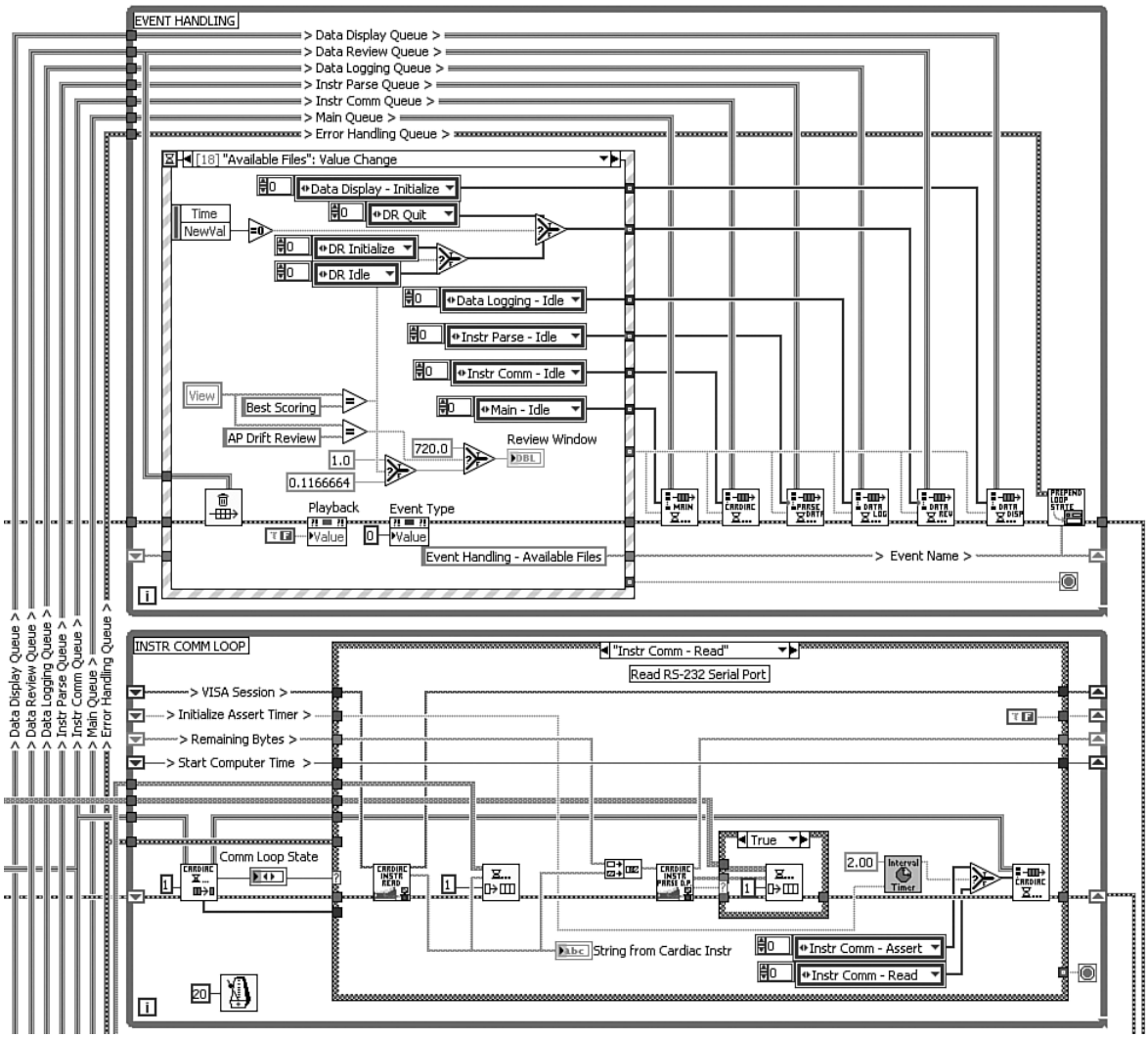


Figure 1-1 continued

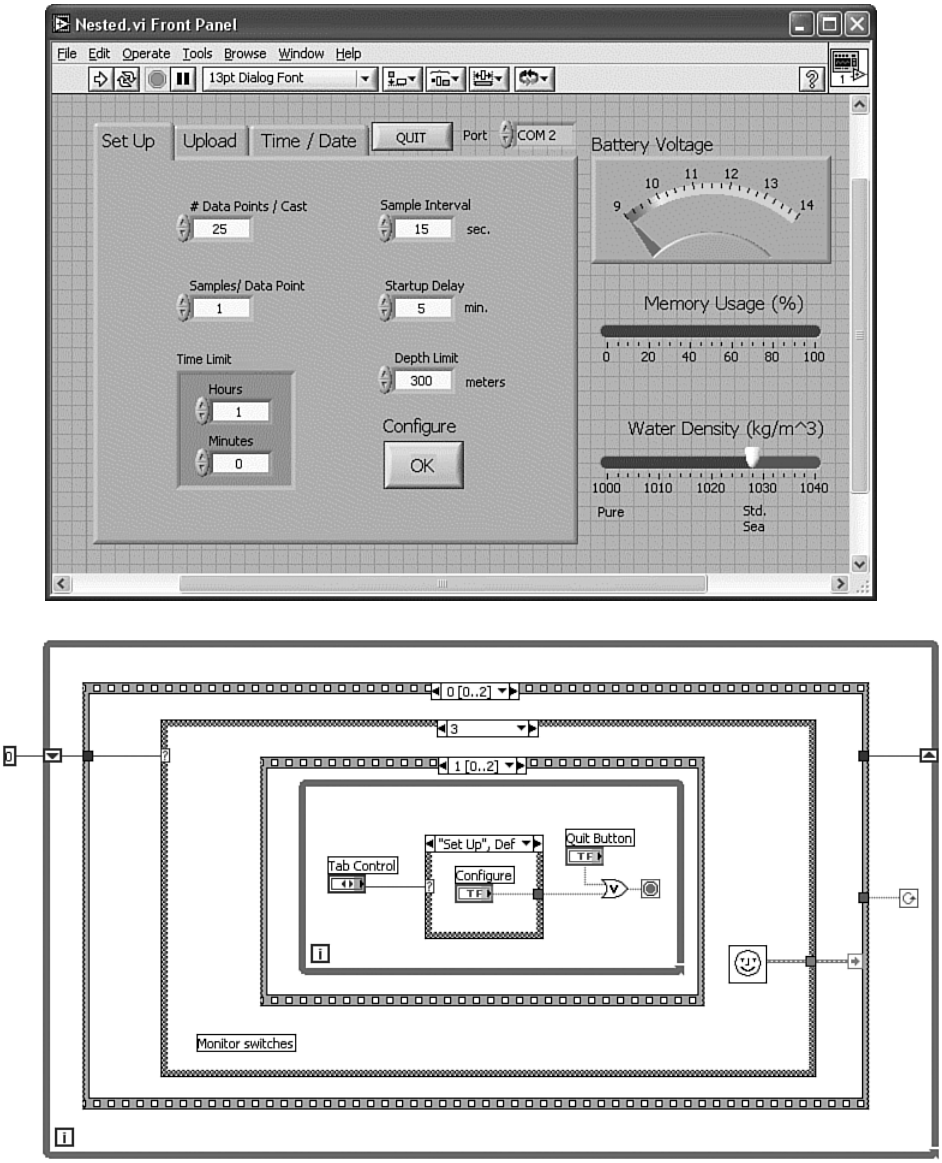
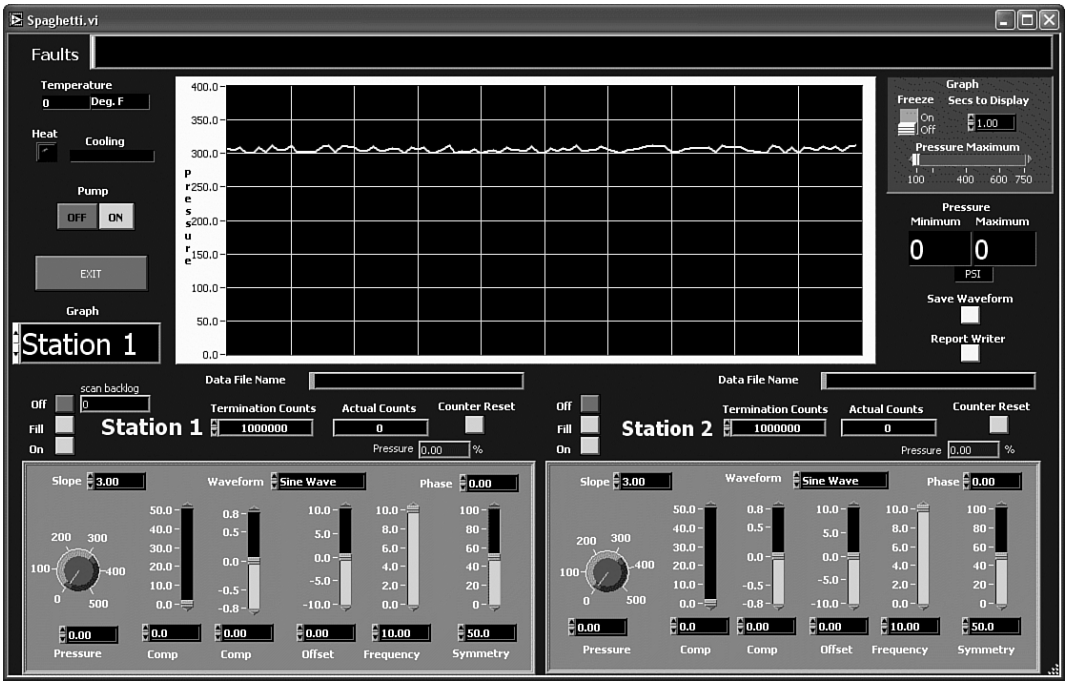


Figure 1-2
Nested VI—the front panel and block diagram for a simple application. The diagram has a highly nested architecture.



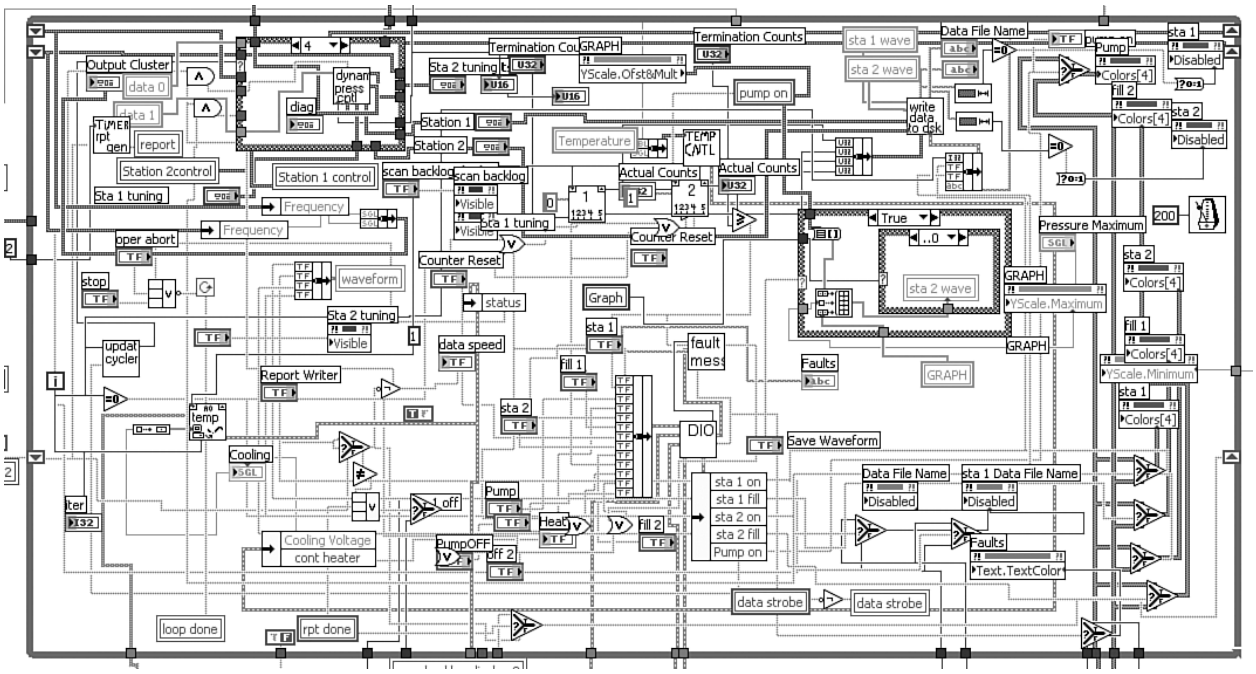


Figure 1-3
Spaghetti VI—the front panel and block diagram (which resembles spaghetti) for a medium-complexity application.

Meticulous VI in Figure 1-1 is a highly complex application used in the medical industry for patient physiologic data monitoring. It has a very professional and highly organized appearance. The front panel is densely functional yet appears neat and intuitive. The diagram is impressive, containing multiple parallel loops, networks of labeled wires, advanced constructs, and liberal documentation. Nested VI in Figure 1-2 is a relatively simple application that controls a scientific instrument. It has a clean, uncluttered front panel and block diagram. The most noteworthy characteristic is the diagram's multiple layers of structures nested within structures. Spaghetti VI in Figure 1-3 is an automated test application with medium complexity. The front panel contains many different colors and font styles, which tend to clash and distract the user. The diagram is a labyrinth of wires and nodes compacted together within a single loop. So dissimilar are these three programming styles that it is worth noting that all three examples are commercial applications developed by experienced LabVIEW professionals. Style is the main differentiator.

Let us break down Theorem 1.1 into seven different parts and evaluate each one individually: ease of use, efficiency, readability, maintainability, robustness, simplicity, and performance. Several of these terms are general, subjective, ambiguous, and overlapping. Therefore, each section begins with a definition of how the term is used here and then discusses how that term relates to development style.

1.1.1 Ease of Use

Ease of use is the ease with which the end user operates the software and accomplishes her objectives. This involves interacting with the application's graphical user interface (GUI). Ease of use ranges in importance from *less* important, for a one-time experiment to be run only by the application's developer, who is intimately familiar with the software; to *very* important, for a production application to be used by a variety of semiskilled operators within a single organization; to *extremely* important, for a mission- or safety-critical application, or a commercial application intended for distribution and resale. If you are designing a commercial application, read a GUI style reference for your application's target operating system. If your application is mission or safety critical, read an appropriate text on human factors. *The LabVIEW Style Book* contains general style rules that can be applied to most LabVIEW applications, from the lab to production, on most operating systems, from desktop PCs running Windows to embedded processors running a real-time or hand-held operating system.

Ease of use is related to readability and responsiveness of the GUI. In Figure 1-1, the front panel for Meticulous VI is intuitively laid out. The principal physiologic data is displayed in stacked waveform charts and large numeric indicators in the center of the panel. Ancillary data is contained on separate display screens. Boolean controls facilitate navigation in a conventional manner. The overall appearance resembles and behaves as a virtual instrument. The application is fast and responds quickly to operations that the user performs.

The front panel of Nested VI in Figure 1-2 uses a tab control to logically organize the front panel objects. The controls are clearly labeled and evenly spaced. However, the purpose of each tab and the order by which the user navigates the GUI are not immediately clear. The diagram contains Sequence structures that execute each frame to completion, regardless of any operations the user performs on the front panel. Its capability to respond to the user clicking **Quit** depends upon which frame of the Sequence structure the application is executing when the button's value changes.

The front panel of Spaghetti VI, shown in Figure 1-3, organizes controls and indicators using clusters. It is logical and organized, but it contains too many colors and font types, and not enough empty

areas or "white space." More important, the performance is not reliable. No matter how attractive a GUI appears, if it does not perform its intended function, it will not provide a positive user experience.

Chapter 3, "Front Panel Style," contains style rules for front panel design that promote consistency and ease of use for your GUIs. Chapter 6, "Data Structures," and Chapter 8, "Design Patterns," provide additional style rules that help optimize the responsiveness of the GUI.

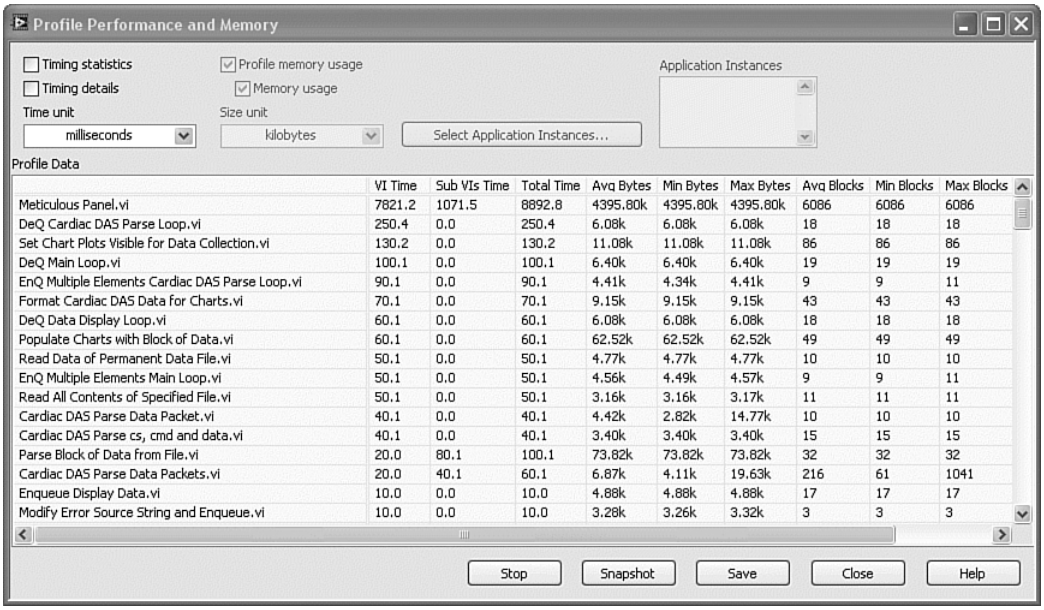
1.1.2 Efficiency

Efficiency pertains to the use of processor, memory, and input/output (I/O) resources. An efficient LabVIEW application executes quickly, without performing unnecessary operations, particularly ones that are performed repeatedly within looping structures. An efficient application also conserves memory by limiting the size of the four LabVIEW memory components: the front panel, block diagram, data space, and code. **Front panel** and **block diagram** memory store the graphical objects and images that comprise the front panel and block diagram, respectively. **Data space memory** contains all the data that flows through the diagram, as well as the diagram constants, default values for front panel controls, and the data that is copied when written to variables and front panel indicators. **Code** is the portion of memory that contains the compiled source code. Finally, efficient applications minimize I/O operations, such as GUI updates, instrument and network communications, and data acquisition (DAQ) calls. Execution speed and memory use are related. Memory and disk operations are a principal source of latencies in all modern computing devices. As memory consumption grows during the execution of an application, LabVIEW's memory manager is called upon to allocate new and larger memory blocks. This causes a delay while the memory manager runs and results in fragmented memory, for which some of the previous blocks are not efficiently used. The memory manager is a wonderful aspect of LabVIEW that handles memory allocation automatically. However, developers should be aware of the types of operations that might cause it to run and avoid these situations from occurring unnecessarily.

LabVIEW contains a tool called the **Profile Performance and Memory** window that directly measures the execution speed and data memory of all VIs loaded in memory. This tool, shown in Figure 1-4, is accessed by selecting **Tools»Profile»Performance and Memory**. This is an excellent tool for helping to improve the efficiency of an application. Use the Profile Performance and Memory window to determine which VIs consume the most time and memory, and examine those more closely. You can optimize your VI's efficiency by iteratively making improvements and checking the profile metrics.

Note that the efficiency of distinctly different applications, such as Meticulous VI, Nested VI, and Spaghetti VI, cannot be compared in any meaningful way. This is because execution time, memory consumption, and I/O depend on the application's requirements as well as the developer's programming style. For example, metrics from the Profile Performance and Memory window might indicate that Nested VI executes the fastest and uses the least memory. However, Nested VI is the least resource-demanding of the three applications and might even have plenty of room for further optimization. The Profile Performance and Memory window is best used for measuring the change in efficiency of a single application when incremental modifications have been made.

An alternate method of evaluating efficiency is simply to inspect the application for sources of inefficiency. Look for unnecessary operations within loops and for operations that create new data buffers. The diagram of Spaghetti VI (see Figure 1-3), for example, contains many nodes within the main While Loop. This includes read local variables, which make copies of their data when read from, and Property Nodes that are written within every iteration, regardless of whether their value has changed.



	VI Time	Sub VIs Time	Total Time	Avg Bytes	Min Bytes	Max Bytes	Avg Blocks	Min Blocks	Max Blocks
Meticulous Panel.vi	7821.2	1071.5	8892.8	4395.80k	4395.80k	4395.80k	6086	6086	6086
DeQ Cardiac DAS Parse Loop.vi	250.4	0.0	250.4	6.08k	6.08k	6.08k	18	18	18
Set Chart Plots Visible for Data Collection.vi	130.2	0.0	130.2	11.08k	11.08k	11.08k	86	86	86
DeQ Main Loop.vi	100.1	0.0	100.1	6.40k	6.40k	6.40k	19	19	19
EnQ Multiple Elements Cardiac DAS Parse Loop.vi	90.1	0.0	90.1	4.41k	4.34k	4.41k	9	9	11
Format Cardiac DAS Data for Charts.vi	70.1	0.0	70.1	9.15k	9.15k	9.15k	43	43	43
DeQ Data Display Loop.vi	60.1	0.0	60.1	6.08k	6.08k	6.08k	18	18	18
Populate Charts with Block of Data.vi	60.1	0.0	60.1	62.52k	62.52k	62.52k	49	49	49
Read Data of Permanent Data File.vi	50.1	0.0	50.1	4.77k	4.77k	4.77k	10	10	10
EnQ Multiple Elements Main Loop.vi	50.1	0.0	50.1	4.56k	4.49k	4.57k	9	9	11
Read All Contents of Specified File.vi	50.1	0.0	50.1	3.16k	3.16k	3.17k	11	11	11
Cardiac DAS Parse Data Packet.vi	40.1	0.0	40.1	4.42k	2.82k	14.77k	10	10	10
Cardiac DAS Parse cs, cmd and data.vi	40.1	0.0	40.1	3.40k	3.40k	3.40k	15	15	15
Parse Block of Data from File.vi	20.0	80.1	100.1	73.82k	73.82k	73.82k	32	32	32
Cardiac DAS Parse Data Packets.vi	20.0	40.1	60.1	6.87k	4.11k	19.63k	216	61	1041
Enqueue Display Data.vi	10.0	0.0	10.0	4.88k	4.88k	4.88k	17	17	17
Modify Error Source String and Enqueue.vi	10.0	0.0	10.0	3.28k	3.26k	3.32k	3	3	3

Figure 1-4
The LabVIEW Profile Performance and Memory window displays the data memory of Meticulous VI.

Inspecting the diagram of Nested VI (see Figure 1-2) reveals a **tight loop**, in which several control terminals are polled within the innermost While Loop as fast as the loop can run. High-speed GUI polling is not an efficient use of the processor. Most humans cannot distinguish between a 1 millisecond (ms) and a 100ms GUI response. More efficient alternatives include adding a delay using the Wait (ms) function or using an Event structure to service all user interface activity. These alternatives free the processor to work on parallel tasks and applications.

Meticulous VI (see Figure 1-1) uses multiple parallel loops on one diagram, many shift registers, and an Event structure. The following tasks execute in their own separate parallel loop: GUI event handling, instrument communications, data parsing, data logging, data review, data display, and error handling. A total of eight parallel loops are used. Each loop is finely tuned for optimum performance. However, the top-level diagram is extremely large, resulting in very large data and diagram memory components. The four components of memory use are provided in the VI Properties window, shown in Figure 1-5. You access the VI Properties window by selecting **File»VI Properties** and then selecting **Memory Usage** for the category.

Event structures are the most efficient method of capturing user interface activity. The GUI event handling loop sleeps, using no processor time, until it receives a registered event from the operating system. This is both more efficient and much more responsive than polling the control terminals in a loop. Hence, Meticulous VI is maximally responsive to GUI events. Finally, Meticulous VI makes extensive use of shift registers for passing data between loop iterations, and queues for passing data between parallel loops. Shift registers are much more efficient alternatives to variables, and queues are much more functional.

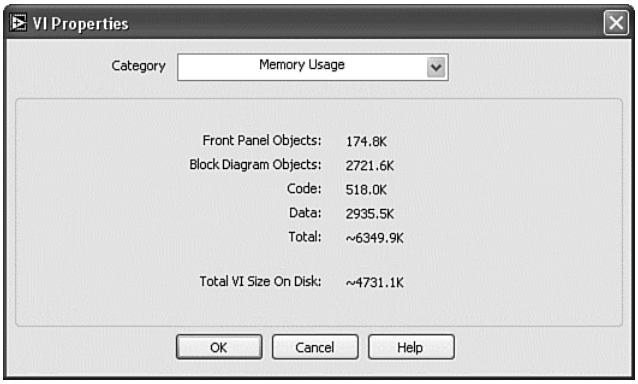


Figure 1-5
The VI Properties window indicates the four categories of memory use for Meticulous VI.

Nested VI makes good use of subVIs, which increase memory efficiency, but uses Sequence structures, which can decrease processing efficiency. SubVIs reduce the quantity of nodes on the diagram, which reduces block diagram memory. In addition, LabVIEW can reclaim memory buffers used by subVIs when the subVIs are not executing, thereby reducing data memory use. However, the nested structures, particularly the Sequence structures, are a potential source of inefficiency. When a Sequence structure is called, all frames must execute through completion, in consecutive order. In many situations, it is preferable to reorder or abort the sequence, to improve its efficiency.

The style rules presented throughout this book promote maximum efficiency. Chapter 4, "Block Diagram," and Chapter 8, "Design Patterns," discuss rules that influence processing efficiency. Chapter 6, "Data Structures," presents rules for efficient memory use.

1.1.3 Readability

Readability refers to how easily the developer can comprehend the source code. This includes both the front panel and the block diagram. The objects on the front panel should be clearly labeled and easily identified. The diagram should be neat, orderly, and easy to follow. The application should be well documented throughout.

The front panel of Meticulous VI appears both simple and perhaps a little mysterious. Figure 1-6 shows the front panel in edit mode. The menu controls along the bottom and right perimeters are Boolean controls with customized appearance and embedded text. A knob is used for rapid menu and waveform navigation. The main display area resembles a collage of indicators suspended on a black background. It is divided into two sections, a large subpanel control on the left and multiple overlapping indicators on the right. Twelve preset display configurations can be programmed by dynamically loading any one of twelve subVIs into the subpanel and making a subset of the overlapping indicators visible. The indicators are customized with a transparent background overlaid on a black decoration, giving them the appearance of floating in space. The diagram (see Figure 1-1) is easy to follow because all data flows through wires from left to right, with no more than one bend to wrap around structures and objects. The diagram is extensively documented, including visible terminal labels, free labels on long wires and within each structure, and enumerated case selectors.

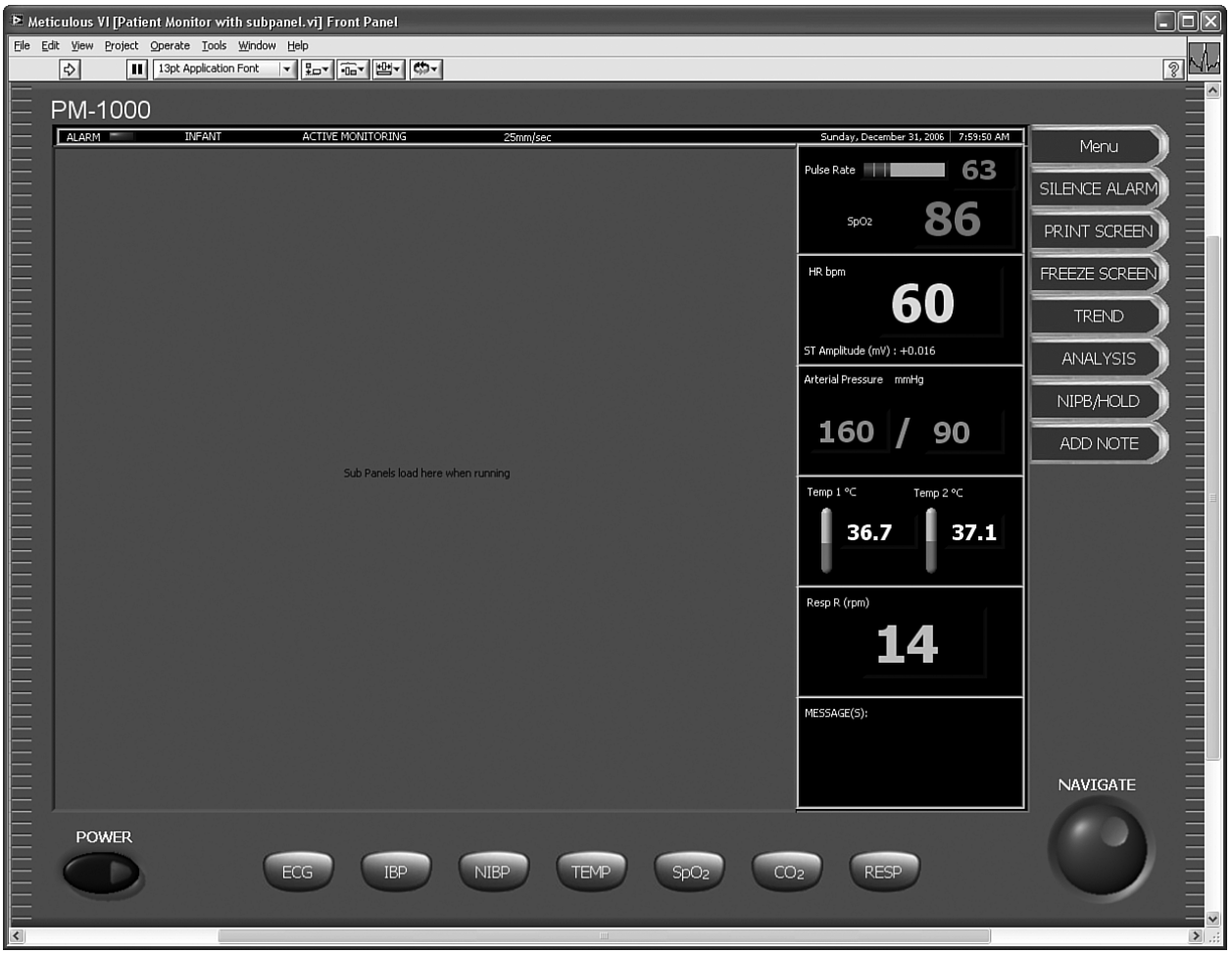


Figure 1-6
The front panel of Meticulous VI in edit mode. It contains customized Boolean controls along the bottom and right perimeters, a knob, a subpanel control, and multiple overlapping indicators.

The front panel of Nested VI (see Figure 1-2) is very simple and readable. The objects on the front panel and diagram are intuitively labeled and evenly spaced. The diagram contains free labels in every frame of every structure. However, the VI, control, and indicator descriptions are absent throughout the application. Descriptions are an important source of documentation, as discussed in Chapter 9, "Documentation." Figure 1-7 shows the Context Help window appearance for a VI and control, without descriptions.

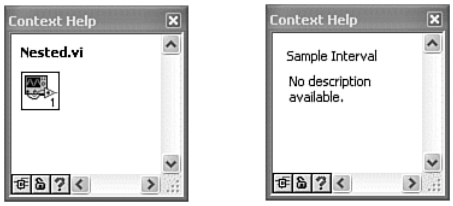


Figure 1-7
The Context Help window reveals missing VI and control descriptions on the panel of Nested VI.

The front panel of Spaghetti VI (see Figure 1-3) contains simple controls, indicators, clusters, and decorations. Intuitive labels exist for most controls, but not all of them. For example, each station cluster contains two vertical fill slides that have the same abbreviated name, **Comp.** These controls are ambiguous. Each control should be uniquely named, and the abbreviation should be replaced with a more intuitive and meaningful term. Chapter 3 presents rules regarding front panel text. The diagram looks like, well, spaghetti. This is because the wiring scheme is haphazard, with data flowing in all directions, and the architecture is not adequate for the application's complexity. It is extremely difficult to visually trace most wires from source terminal to destination terminal. Chapter 4 presents rules for proper wiring and data flow that facilitate readability.

1.1.4 Maintainability

A LabVIEW application is **maintainable** if other LabVIEW developers besides the author understand the source code and if it can be easily modified and expanded to change or add new functionality. Hence, the source code must be readable to be maintainable. Additionally, the source code must use constructs that are modular and scalable, thereby allowing for future expansion of functionality.

The front panel of Meticulous VI contains the subpanel control and multiple overlapping indicators, as shown in Figure 1-6. The subpanel control promotes maintenance because new displays are created and existing displays are modified via component subVIs that are loaded into the subpanel control. This provides tremendous flexibility in the display appearance, along with the capability to modularize different display configurations as subVI panels. Hence, editing a display configuration involves editing a specific component subVI panel. However, changes to the overlapping indicators on the right require showing, hiding, and careful sizing and positioning of the indicators. Maintaining this portion of the front panel is tedious.

The diagram of Meticulous VI (see Figure 1-1) uses a complex application framework that consists of multiple parallel loops. Most loops, including the loop labeled **INSTR COMM LOOP,** use a variation of the State Machine design pattern. This consists of a Case structure with a separate case for each state of the application, and an enumeration wired to the case selector. The State Machine design pattern is readable because each frame has an intuitive label in the selector area that corresponds to the labels of the enumerated type definition and describes the function of each state. It is also scalable because states can be added and removed simply by adding and removing cases to the Case structure, and items to the enumerated type definition. Chapter 8 discusses the State Machine design pattern and Multiple Loop Application Framework.

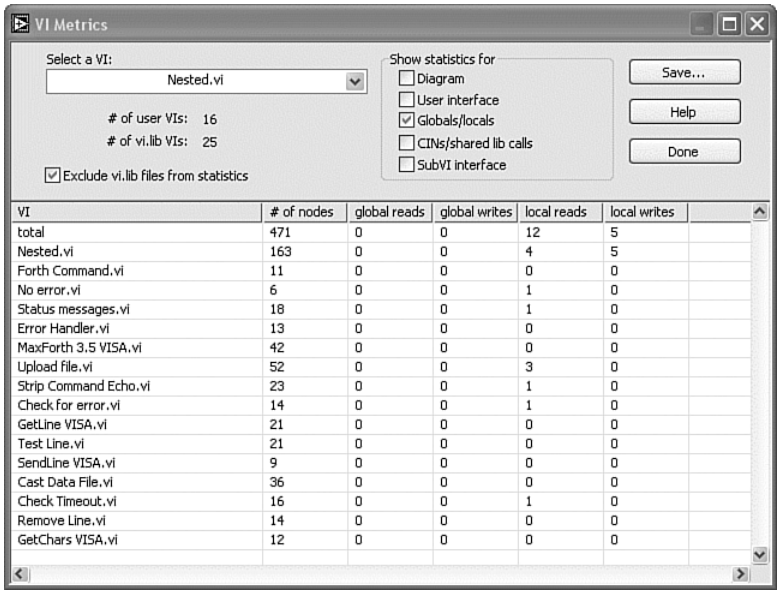
The front panel of Nested VI (see Figure 1-2) uses a simple tab control that can be readily expanded by adding new tabs. Each tab contains a different display page and can be selected by the user when needed. The diagram however, contains Sequence and Case structures nested together, which obstructs the developer's view of the source code by forcing her to navigate each frame of each structure to understand how it works. Moreover, it is not a standard design pattern. Rather, it is confusing compared to the State Machine design pattern, which is more recognizable and less nested.

Spaghetti VI (see Figure 1-3) is not maintainable because the architecture is not scalable, the diagram is not modular, and the wiring is sloppy. The architecture consists of a single While Loop with all nodes that execute more than once inside the loop, and wires, structures, and nodes intermixed and overlapping. Known as the Continuous Loop design pattern, this is discussed in Chapter 8. If the scope of the application expands, the user either increases the size of the While Loop, which is already larger than what can be displayed on one screen in high resolution, or adds to the confusion within the existing area. Creating new wires and nodes within this framework would be tortuous. The front panel also has maintenance considerations. Adding a new station, for example, involves replicating the station cluster and the station-specific controls located outside the cluster. However, the front panel cannot support an additional station in its current form. Fortunately, a tab control can be integrated to provide substantial improvements with minimal effort. You can dedicate a separate tab for each station and then add and remove tabs as needed. In addition, you can incorporate the station-specific controls outside the cluster into the station cluster and save the cluster as a type definition. This way, any changes to the type definition are applied to all stations that use the type definition. Chapter 3 discusses tab controls, and Chapter 6 covers type definitions. The style rules presented throughout this book help ensure maintainability.

1.1.5 Robustness

A LabVIEW application is **robust** if it is bug free and never crashes. LabVIEW provides fundamental constructs that promote robust applications, including subVIs and error handling, as well as fundamental elements such as local and global variables that can hinder applications when used improperly. Good LabVIEW developers always modularize their diagrams using subVIs. This involves identifying a portion of code that performs a specific task and creating a subVI to perform that task. Testing and debugging subVIs is relatively easy, as long as they are each dedicated to a specific task comprised of a limited number of nodes. Modular applications that use previously tested and debugged subVIs are generally higher quality from the outset. Any bugs that are discovered are easy to identify and isolate because of the modularity.

LabVIEW contains a tool called VI Metrics that reports how many user subVIs and write variables an application uses, as well as the overall number of nodes on the diagram. The VI Metrics utility is accessed by selecting **Tools»Profile»VI Metrics**. Figure 1-8 shows the VI Metrics window with Nested VI loaded.



VI	# of nodes	global reads	global writes	local reads	local writes
total	471	0	0	12	5
Nested.vi	163	0	0	4	5
Forth Command.vi	11	0	0	0	0
No error.vi	6	0	0	1	0
Status messages.vi	18	0	0	1	0
Error Handler.vi	13	0	0	0	0
MaxForth 3.5 VISA.vi	42	0	0	0	0
Upload file.vi	52	0	0	3	0
Strip Command Echo.vi	23	0	0	1	0
Check for error.vi	14	0	0	1	0
GetLine VISA.vi	21	0	0	0	0
Test Line.vi	21	0	0	0	0
SendLine VISA.vi	9	0	0	0	0
Cast Data File.vi	36	0	0	0	0
Check Timeout.vi	16	0	0	1	0
Remove Line.vi	14	0	0	0	0
GetChars VISA.vi	12	0	0	0	0

Figure 1-8
The number of nodes and the number of user VIs reported by the VI Metrics window help approximate the application's modularity.

For comparison purposes, we can define a modularity index as follows:

Equation 1.1

Modularity Index = (# user VIs / total # of nodes) × 100

I recommend a modularity index greater than 3.0. In our examples, Meticulous VI has 111 user VIs and 4,947 nodes, for a modularity index of 2.2. Nested VI has 16 user VIs and 471 nodes, for a modularity index of 3.3. Spaghetti VI contains 56 user VIs and 1,944 nodes, for a modularity index of 2.9. Chapter 4 discusses subVI modularization in detail.

Error handling consists of trapping errors by propagating the error cluster and reporting errors using a dialog or log file. Error handling is critical for robust application performance. Meticulous VI contains very thorough error handling, as can be seen by the error clusters that are propagated among all nodes that have error terminals. Any errors that occur within a loop are passed to a dedicated error handling loop via queue, where they are evaluated, reported, and logged. Figure 1-9 shows the **Error Handling Loop**.

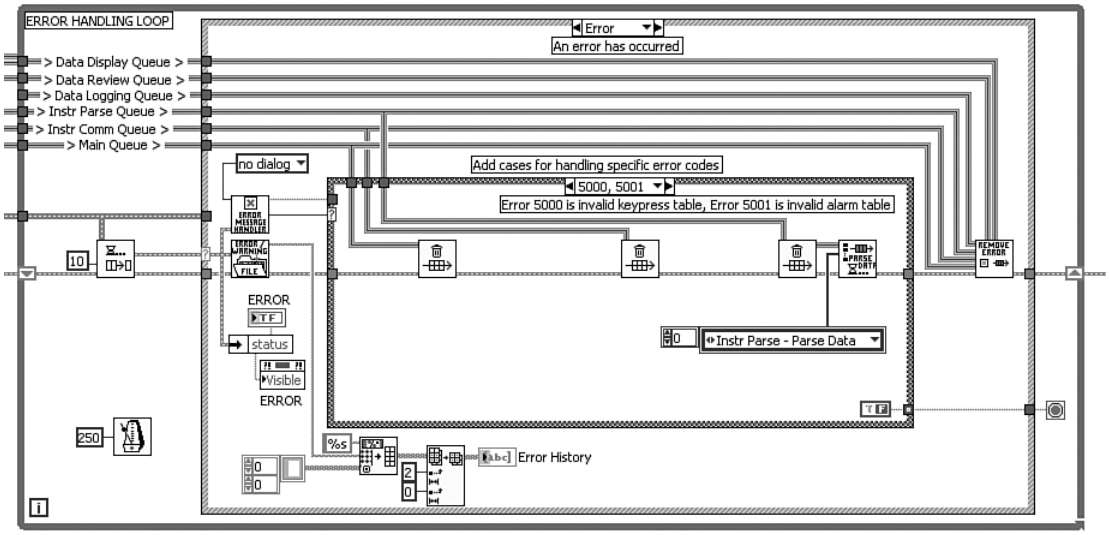


Figure 1-9
Meticulous VI contains a dedicated error handling loop that receives error information from the parallel loops via queue and evaluates, reports, and logs the errors.

The error handling scheme in Nested VI is less thorough and sophisticated than Meticulous VI, yet it is simple and functional. Referring to Figure 1-10, the first subVI called in each case of the inner Case structure has an unwired **error in** terminal, whereas the **error out** terminal is propagated between the nodes of its case, through the Case structure's output tunnel, to a Sequence Local. The Sequence Local passes the error information to the error handling subVI in frame 1 of the outermost Sequence structure. You can improve error handling style by propagating the error cluster between loop iterations and wiring it to the **error in** terminals of all nodes. Efficiency also can be improved if the error handling code is not called unless an error occurs.

Finally, error handling is both incomplete and ineffective in Spaghetti VI. As you can see in Figure 1-3, no apparent error handling scheme is used. Overall, Meticulous VI is the most robust in terms of error handling. Proper error handling is an essential ingredient for reliable and robust applications. Chapter 7, "Error Handling," covers this topic in detail.

Writing to local and global variables can cause unexpected results in an application. For example, a race condition occurs when two copies of a write variable are written to at the same time. As a general rule, the more write variables an application has, the greater the potential for unexpected and undesirable behavior. The quantity of local and global write variables is quickly determined from the VI Metrics window. Meticulous VI contains 92 write variables, Nested VI contains only 5, and Spaghetti VI contains 46. Hence, Nested VI is the most robust in terms of fewest write variables. Chapter 4 discusses variables in greater detail. The rules presented throughout this book ensure robust applications.

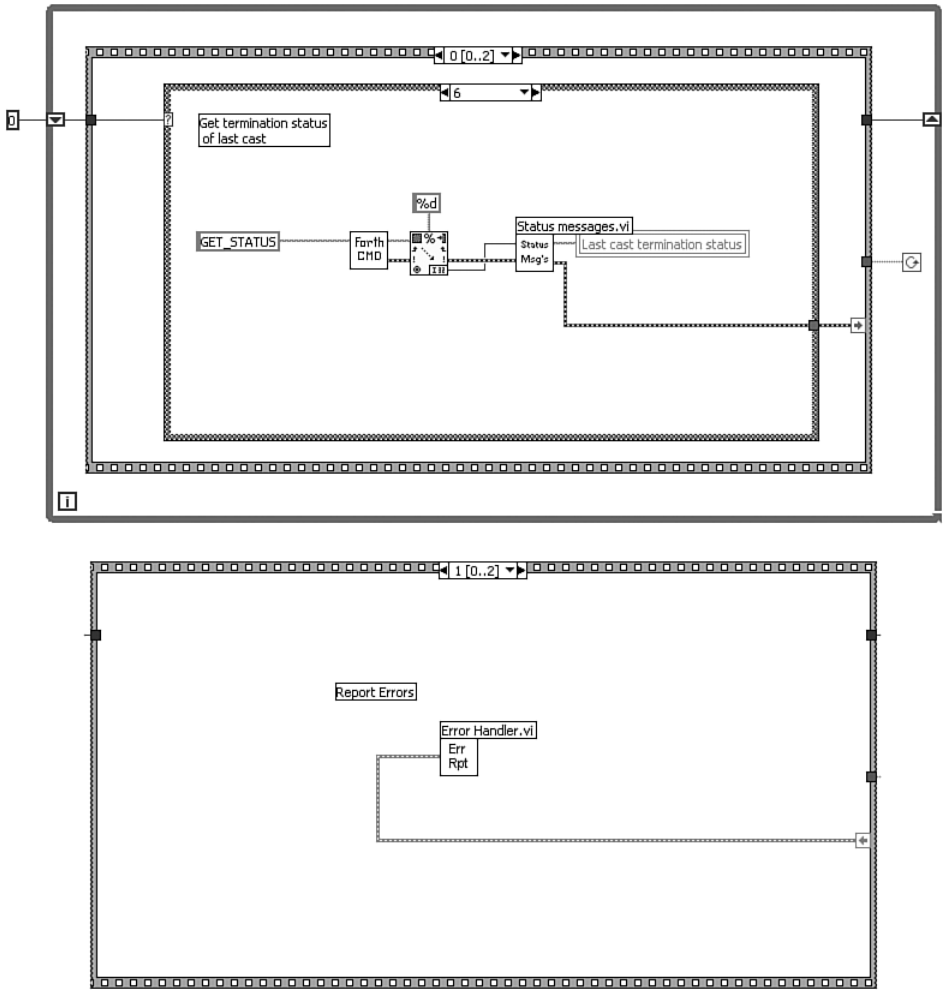


Figure 1-10
The error handling scheme of Nested VI is functional but not optimal. The error cluster propagates between subVIs, through the Case structure, to the sequence local. The error is processed using the Error Handler VI in frame 1 of the Sequence structure.

1.1.6 Simplicity

Simplicity inversely relates to the quantity of nodes and terminals that comprise an application. Simplicity is the opposite of complexity. The fewer front panel objects and block diagram nodes, the simpler the application. Simplicity affects readability and performance. It is generally desirable to minimize complexity by choosing the simplest implementations that require the fewest nodes.

The VI Metrics window provides a useful interface for evaluating simplicity. Choose a VI in the **Select a VI** control, such as the top-level VI for an application. The totals displayed in the top row of

the metrics table indicate the simplicity. Specifically, the total number of nodes indicates the total number of functions, subVIs, structures, front panel terminals, constants, global and local variables, and property nodes within the VI and all subVIs combined. The smaller the nodes total, the simpler the application. Note that VIs that are called by reference are not visible to the VI Metrics tool and are not included in the nodes total.

Simplicity is primarily dictated by the requirements of the application. The more functionality that is required, the more source code is required and the more complex the application is. The three example applications from Figures 1-1, 1-2, and 1-3 all have statically linked subVIs, with the exception of the component subVIs used by the subpanel of Meticulous VI. You can compare the complexity of each application using the total number of nodes reported by the VI Metrics window. With 4,947 nodes, Meticulous VI is by far the most complex, followed by Spaghetti VI, with 1,944. On the contrary, Nested VI is a simple application, with only 471 total nodes.

However, simplicity also relates to style. The total number of nodes required to perform a specific task varies based on how the developer chooses to implement the task. Implementations that contain fewer nodes are generally more efficient and readable than implementations that contain more nodes. For example, OpenG[1] held a public LabVIEW coding contest. The requirement was to develop a VI that removed all backspaces and their immediately preceding character from an input string. The VIs were informally judged in terms of icon style, performance, and diagram style. A wide variety of VIs were submitted, all performing the same task. Some of the submissions were implemented with only 12 or 13 nodes; others contained 25 or more nodes. Figure 1-11 illustrates three submissions. The implementation with the fewest nodes, Figure 1-11A, is easier to read and understand than implementations with more nodes, including Figure 1-11B and Figure 1-11C. We compare the performance of these examples next.

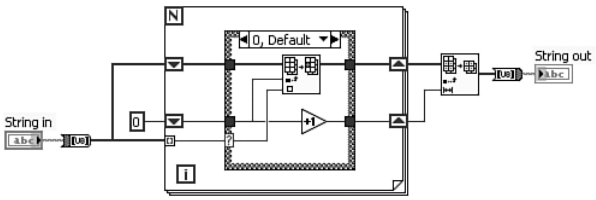


Figure 1-11A
This subVI implementation uses only 13 nodes to remove all backspaces from the input string.

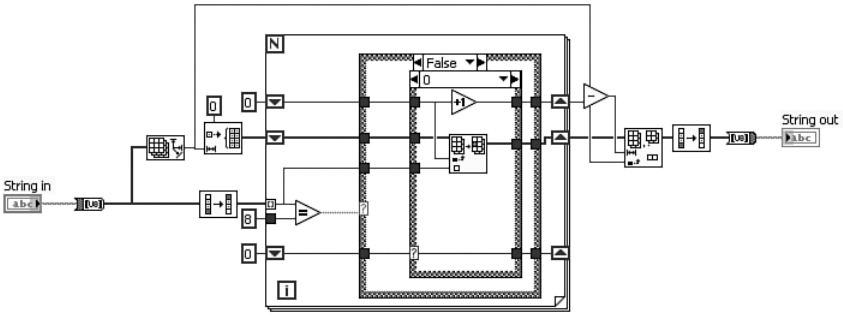


Figure 1-11B
This implementation uses 22 nodes, including nested Case structures, to achieve the same functionality.

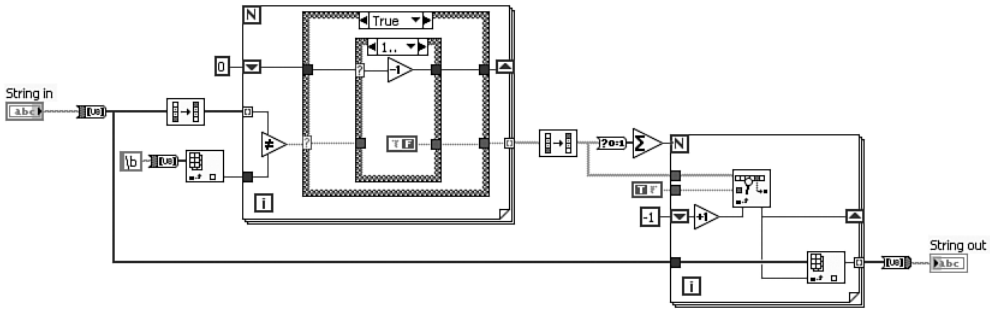


Figure 1-11C
This implementation uses 25 nodes including two For Loops and nested Case structures.

1.1.7 Performance

Performance is a particularly broad term that has many definitions. In this discussion, we consider application and subVI performance. **Application performance** refers to how well the application or VI completes its intended mission. Therefore, application performance measures are related to the requirements. For example, if an application is developed to reduce test time, test time is the primary performance measure. If improving product quality is the primary objective, a reduction in the number of defective products or customer returns might be the performance measure. Most applications have multiple objectives. The most common objectives include increasing the speed and efficiency by which a task is accomplished while improving ease of use. Therefore, application performance can be considered a combination of efficiency (see Section 1.1.2) and ease of use (see Section 1.1.1). Robustness (see Section 1.1.5) also is required. If the application is not reliable and robust, performance is meaningless.

Style affects application performance. Referring to the application examples, the primary objective of Meticulous VI (see Figure 1-1) is to perform reliable acquisition, logging, and display of physiologic data. This objective is achieved using the Multiple Loop Application Framework, which prioritizes the important tasks in dedicated loops. Nested VI (see Figure 1-2) performs configuration of a scientific instrument and then uploads data acquired from a variety of remote sample sites. In this case, the simplistic GUI aids the user in error-free operation. Spaghetti VI (see Figure 1-3), however, suffers from unreliable and sluggish performance because of the inefficiency of the single-loop architecture and other style-related factors previously discussed.

Consider **subVI performance** as its execution speed. Section 1.1.6 mentioned the relationship between simplicity and performance. Often the fewer nodes that are used to implement a subVI, the faster the subVI executes. For example, the three implementations of the Remove Backspace VI in Figure 1-11 have different execution speeds. Figure 1-12 shows the Profile Performance and Memory window containing the timing statistics after each VI was called 1,000 times within a loop, with a long string passed to the **string in** terminal. Thirteen Nodes VI consumed only 5.6ms, Twenty-Two Nodes VI consumed an average of 5.9ms per run, and Twenty-Five Nodes VI consumed 12.8ms. Although 5.6ms, 5.9ms, and 12.8ms might seem insignificant, if the three development styles are extended to every subVI throughout a large application, the performance difference becomes substantial.

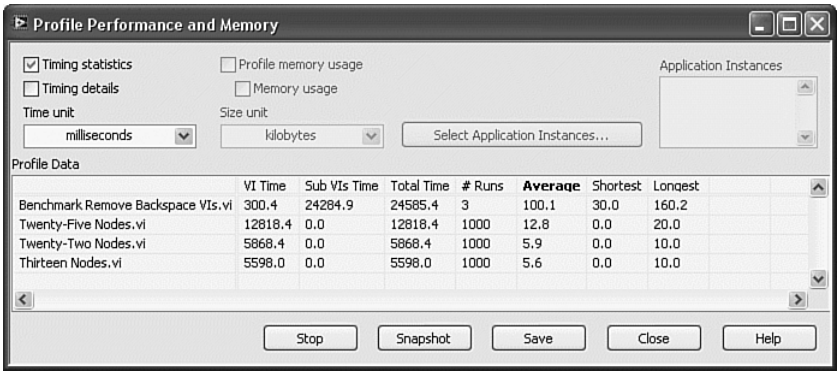


Figure 1-12
The Profile Performance and Memory window displays the timing statistics after each Remove Backspace subVI runs 1,000 times.

1.1.8 Style Tools

LabVIEW contains several built-in tools that help evaluate style. In the previous sections, you observed the Profile Performance and Memory window used to evaluate data memory use and execution speed. The Memory Usage VI properties page determines the memory allocated to each of the four memory components of a single VI. These tools provide metrics for efficiency (see Section 1.1.2) and performance (see Section 1.1.7). The VI Metrics window determines simplicity (see Section 1.1.6) and is used to calculate the modularity index to help evaluate robustness (see Section 1.1.5). Additionally, code reviews provide a more direct method of evaluating style. Code reviews are performed utilizing a combination of self reviews, automated inspection, and peer reviews. The Style Rules Summary in Appendix B can be used as a style checklist for performing self reviews. Additionally, many of the rules presented in *The LabVIEW Style Book* can be evaluated using the LabVIEW VI Analyzer Toolkit from National Instruments. Chapter 10, "Code Reviews," discusses these tools in greater detail.

1.2 Style Versus Time Tradeoff

Good LabVIEW developers are often in high demand within our organizations. The applications we develop are critical to our companies' research, development, or production endeavors. Intense time-to-market pressures encourage fast application development cycles. As such, LabVIEW developers often gravitate toward shortcut and fast development techniques for speeding their development time. After all, that is why we all chose LabVIEW in the first place—it substantially accelerates our development time versus traditional text-based programming languages.

Definition 1.1: Development time *includes the hours required to develop, document, test, modify, and maintain an application* ***throughout its entire life cycle****.*

On the surface, the pursuit of fast development time appears at odds with good programming style. The former might entail shortcuts that bypass the latter. At Bloomy Controls, we are familiar with this apparent paradox. Many of our customers contract us as a means of achieving tight deadlines. Any style rules that increase development time without an offsetting productivity benefit must be eliminated. However, it is important to look at the entire life cycle of the application when considering development time. Many developers mistakenly consider the time from conception until the first version of the software is released as the development time and strive to minimize this time. In actuality, development time includes the hours consumed for all testing, debugging, documentation, revisions, and upgrades that occur throughout the entire life cycle of the application. When taken in this context, it behooves us to use good style.

Theorem 1.2: *Good style* ***reduces*** *development time and effort.*

Good development style might require more up-front time and effort, but the quality of the software is higher from the start, and the software is more useful throughout the life cycle of the application. I cannot overemphasize this concept. In numerous instances, Bloomy Controls has been contracted to make minor functional modifications and bug fixes to customers' preexisting applications that required disproportional effort. In some instances, we were forced to rework the application as if starting over from scratch because bad style prevented modifications. Using proper style from the outset reduces development time throughout the application's life cycle and makes modifications easy.

Furthermore, this book provides only rules that are practical to implement in real-world situations, including projects with tight deadlines. I have eliminated rules that are not practical. One such rule that we had many years ago was a requirement for developers to edit the VI History *every* time *any* VI was saved, regardless of the project, development model, or stage of development. As a rule, all the Bloomy engineers had LabVIEW configured to automatically prompt for revision history comment. It immediately became clear that this was impractical and a nuisance for many applications during the early stages of development, prior to the first release. This rule was abandoned less than two weeks after it began.

An example of a rule that some consider a hindrance but that I adamantly require among all of my staff is writing VI descriptions for every VI. VI descriptions are a critical source of documentation. I have seen employees, customers, and colleagues try to save time by saving this simple snippet of documentation for *later* because they're trying to get a piece of production machinery online or meet some critical deadline. However, more often than not, *later* is really *never*. Even with the best intentions, the same time pressures that cause one to skip the descriptions in the first place usually linger around later. Alternatively, if we do return to our code and write the descriptions later, it is much more time-consuming because we have to locate the undocumented VIs one at a time and refresh our memories about their purpose. It is much less time-consuming and more reliable and efficient to write the description immediately after the VI is written or revised and it is fresh in our minds. This practice ensures that the descriptions are written promptly, accurately, and efficiently. Chapter 9 covers documentation, including VI descriptions. The style rules presented throughout this book are practical to implement after they are learned and applied.