NATIONAL INSTRUMENTS™
LabVIEW™

Development Guidelines

NATIONAL INSTRUMENTS™

July 2000 Edition
Part Number 321393C-01

Worldwide Technical Support and Product Information

www.ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 794 0100

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011,
Canada (Calgary) 403 274 9391, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521,
China 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24,
Germany 089 741 31 30, Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406,
Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico (D.F.) 5 280 7625,
Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00,
Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085,
Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, see the *Technical Support Resources* appendix. To comment on the documentation, send e-mail to techpubs@ni.com

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

About This Manual

Chapter 1
Development Models

Chapter 2
Incorporating Quality into the Development Process

Chapter 6
LabVIEW Style Guide

Appendix A
References

Appendix B
Technical Support Resources

Glossary

Index

Figures

Table

About This Manual

The LabVIEW Development Guidelines describe many of the issues that arise when developing large applications. The guidelines are based on the advice of LabVIEW developers, and provide a basic survey of software engineering techniques you might find useful when developing your own projects.

There is also a discussion of style for creating VIs. Developers who have used LabVIEW and are comfortable in the LabVIEW environment can use the LabVIEW Development Guidelines to maintain a consistent and effective style in their projects.

Conventions

The following conventions appear in this manual:

» The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

 This icon denotes a tip, which alerts you to advisory information.

bold Bold text denotes items that you must select or click on in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace` Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

Platform Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Related Documentation

The following documents contain information that you might find helpful as you read this manual:

- *LabVIEW Help*, available by selecting **Help»Contents and Index**.

1 Development Models

This chapter provides examples of some common development pitfalls and describes a number of software engineering lifecycle models.

LabVIEW makes it easy to assemble components of data acquisition, test, and control systems. Because it is so easy to program in LabVIEW, you might be tempted to begin developing VIs immediately with relatively little planning. For simple applications, such as quick lab tests or monitoring applications, this approach might be appropriate. However, for larger development projects, good planning becomes vital.

Common Development Pitfalls

If you have developed large applications before, you probably have heard some of the following statements. Most of these approaches start out with good intentions and seem quite reasonable. However, these approaches are often unrealistic and can lead to delays, quality problems, and poor morale among team members.

- "I haven't really thought it through, but I'd guess that the project you are requesting can be completed in…"

  Off-the-cuff estimates rarely are correct because they usually are based on an incomplete understanding of the problem. When developing for someone else, you might each have different ideas about requirements. To estimate accurately, you both must clearly understand the requirements and work through at least a preliminary high-level design so you understand the components you need to develop.

- "I think I understand the problem the customer wants to solve, so I'm ready to dive into development."

  There are two problems with this statement. First, lack of consensus on project goals results in schedule delays. Your idea of what a customer wants might be based on inadequate communication. Developing a requirements document and prototyping a system, both described in the *Lifecycle Models* section later in this chapter, can be useful tools to clarify goals. A second problem with this statement is that diving into development might mean writing code without a detailed design. Just as builders do not construct a building without architectural plans,

developers should not begin building an application without a detailed design. Refer to the *Code and Fix Model* section later in this chapter for more information about development models to follow.

- "We don't have time to write detailed plans. We're under a tight schedule, so we need to start developing right away."

  This situation is similar to the previous example but is such a common mistake that it is worth emphasizing. Software developers frequently skip important planning because it does not seem as productive as developing code. As a result, you develop VIs without a clear idea of how they all fit together, and you might have to rework sections as you discover mistakes. Taking the time to develop a plan can prevent costly rework at the development stage. Refer to the *Lifecycle Models* section later in this chapter and Chapter 3, *Prototyping and Design Techniques*, for better approaches to developing software.

- "Let's try for the whole ball of wax in the first release. If it doesn't do everything, it won't be useful."

  In some cases, this might be correct. However, in most applications, developing in stages is a better approach. When you analyze the requirements for a project, prioritize features. You might be able to develop an initial VI that provides useful functionality in a shorter time at a lower cost. Then, you can add features incrementally. The more you try to accomplish in a single stage, the greater the risk of falling behind schedule. Releasing software incrementally reduces schedule pressures and ensures timely software release. Refer to the *Lifecycle Models* section later in this chapter for more information about using development models.

- "If I can just get all the features in within the next month, I should be able to fix any problems before the software is released."

  To release high-quality products on time, maintain quality standards throughout development. Do not build new features on an unstable foundation and rely on correcting problems later. This exacerbates problems and increases cost. Although you might complete all the features on time, the time required to correct the problems in the existing and the new code can delay the release of the product. Prioritize features and implement the most important ones first. Once the most important features are tested thoroughly, you can choose to work on lower priority features or defer them to a future release. Refer to Chapter 2, *Incorporating Quality into the Development Process*, for more information about techniques for producing high-quality software.

- "We're behind in our project. Let's throw more developers onto the problem."

  In many cases, doing this actually can delay the project. Adding developers to a project requires time for training, which can take away time originally scheduled for development. Add resources earlier in the project rather than later. Also, there is a limit to the number of people who can work on a project effectively. With a few people, there is less overlap. You can partition the project so each person works on a particular section. The more people you add, the more difficult it becomes to avoid overlap. Chapter 3, *Prototyping and Design Techniques*, describes methods for partitioning software for multiple developers. Chapter 2, *Incorporating Quality into the Development Process*, describes *configuration management* techniques that can help minimize overlap.
- "We're behind in our project, but we still think we can get all the features in by the specified date."

  When you are behind in a project, it is important to recognize that fact and deal with it. Assuming you can make up lost time can postpone choices until it becomes costly to deal with them. For example, if you realize in the first month of a six-month project that you are behind, you might sacrifice planned features or add time to the overall schedule. If you do not realize you are behind schedule until the fifth month, other groups might have made decisions that are costly to change.

  When you realize you are behind, adjust the schedule or consider features you can drop or postpone to subsequent releases. Do not ignore the delay or sacrifice testing scheduled for later in the process.

Numerous other problems can arise when developing software. The following list includes some of the fundamental elements of developing quality software on time:

- Spend sufficient time planning.
- Make sure the whole team thoroughly understands the problems that must be solved.
- Have a flexible development strategy that minimizes risk and accommodates changes.

Lifecycle Models

Software development projects are complex. To deal with these complexities, developers have collected a core set of development principles. These principles define the field of software engineering. A major component of this field is the lifecycle model. The lifecycle model describes the steps you follow to develop software—from the initial concept stage to the release, maintenance, and subsequent upgrading of the software.

Currently, there are many different lifecycle models. Each has advantages and disadvantages in terms of time-to-release, quality, and risk management. This section describes some of the most common models used in software engineering. Many hybrids of these models exist, so use the parts you believe will work for your project.

Although this section is theoretical in its discussion, in practice consider all the steps these models encompass. Consider when and how you decide that the requirements and specifications are complete and how you deal with changes to them. The lifecycle model serves as a foundation for the entire development process. Good choices in this area can improve the quality of the software you develop and decrease the time it takes to develop it.

Code and Fix Model

The code and fix model probably is the most frequently used development methodology in software engineering. It starts with little or no initial planning. You immediately start developing, fixing problems as you find them, until the project is complete.

Code and fix is a tempting choice when you are faced with a tight development schedule because you begin developing code right away and see immediate results.

Unfortunately, if you find major architectural problems late in the process, you might have to rewrite large parts of the application. Alternative development models can help you catch these problems in the early concept stages when it is easier and much less expensive to make changes.

The code and fix model is appropriate only for small projects that are not intended to serve as the basis for future development.

Waterfall Model

The waterfall model is the classic model of software engineering. It has deficiencies, but it serves as a baseline for many other lifecycle models.

The pure waterfall lifecycle consists of several non-overlapping stages, as shown in Figure 1-1. It begins with the software concept and continues through requirements analysis, architectural design, detailed design, coding, testing, and maintenance.

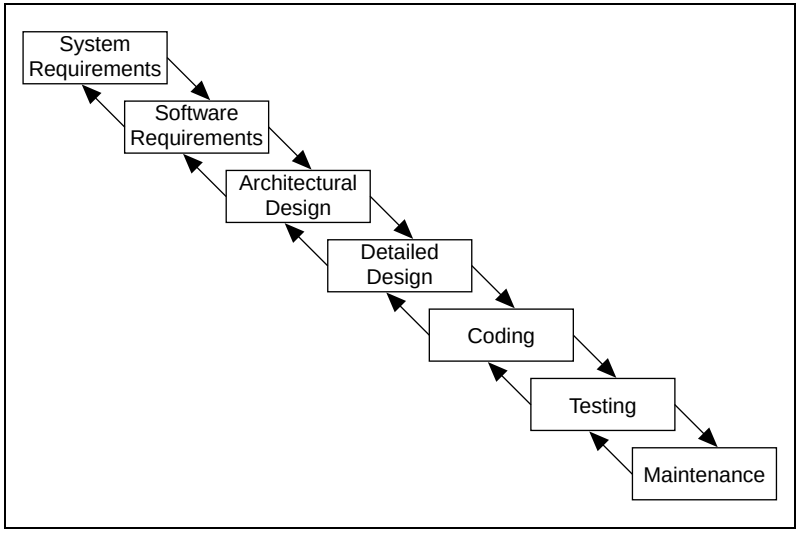

Figure 1-1. Waterfall Lifecycle Model

- System requirements—Establishes the components for building the system. This includes the hardware requirements (number of channels, acquisition speed, and so on), software tools, and other necessary components.
- Software requirements—Concentrates on the expectations for software functionality. You identify which of the system requirements the software affects. Requirements analysis might include determining interaction needed with other applications and databases, performance requirements, user interface requirements, and so on.
- Architectural design—Determines the software framework of a system to meet the specified requirements. The design defines the major components and the interaction of those components, but it does not define the structure of each component. You also determine the external interfaces and tools to use in the project. Examples include

decisions on hardware, such as plug-in boards, and external pieces of software, such as databases or other libraries.

- Detailed design—Examines the software components defined in the architectural design stage and produces a specification for how each component is implemented.
- Coding—Implements the detailed design specification.
- Testing—Determines whether the software meets the specified requirements and finds any errors present in the code.
- Maintenance—Perform as needed to deal with problems and enhancement requests after the software is released.

  In some organizations, each change is reviewed by a change control board to ensure that quality is maintained. You also can apply the full waterfall development cycle model when you implement these change requests.

In each stage, you create documents that explain your objectives and describe the requirements for that phase. At the end of each stage, you hold a review to determine whether the project can proceed to the next stage. Also, you can incorporate prototyping into any stage from the architectural design and after. Refer to the *Prototyping* section later in this chapter for more information about using prototyping in projects.

The waterfall lifecycle model is one of the oldest models and is widely used in government projects and in many major companies. Because it emphasizes planning in the early stages, it helps catch design flaws before they are developed. Also, because it is document and planning intensive, it works well for projects in which quality control is a major concern.

Many people believe you should not apply this model to all situations. For example, with the pure waterfall model, you must state the requirements before you begin the design, and you must state the complete design before you begin coding. There is no overlap between stages. In real-world development, however, you might discover issues during the design or coding stages that point out errors or gaps in the requirements.

The waterfall method does not prohibit returning to an earlier phase, for example, from the design phase to the requirements phase. However, this involves costly rework. Each completed phase requires formal review and extensive documentation development. Thus, oversights made in the requirements phase are expensive to correct later.

Because the actual development comes late in the process, you do not see results for a long time. This can be disconcerting to management and to customers. Many people also think the amount of documentation is excessive and inflexible.

Although the waterfall model has its weaknesses, it is instructive because it emphasizes important stages of project development. Even if you do not apply this model, consider each of these stages and its relationship to your own project.

Modified Waterfall Model

Many engineers recommend modified versions of the waterfall lifecycle. These modifications tend to focus on allowing some of the stages to overlap, reducing the documentation requirements, and reducing the cost of returning to earlier stages to revise them. Another common modification is to incorporate prototyping into the requirements phases, as described in the following section.

Overlapping stages such as requirements and design make it possible to feed information from the design phase back into the requirements. However, this can make it more difficult to know when you are finished with a given stage. Consequently, it is more difficult to track progress. Without distinct stages, problems might cause you to defer important decisions until late in the process when they are more expensive to correct.

Prototyping

One of the main problems with the waterfall model is that the requirements often are not completely understood in the early development stages. When you reach the design or coding stages, you begin to see how everything works together, and you might discover you need to adjust requirements.

Prototyping can be an effective tool for demonstrating how a design might deal with a set of requirements. You can build a prototype, adjust the requirements, and revise the prototype several times until you have a clear picture of your overall objectives. In addition to clarifying the requirements, the prototype also defines many areas of the design simultaneously.

The pure waterfall model allows for prototyping in the later architectural design stage and subsequent stages but not in the early requirements stages.

Prototyping has drawbacks, however. Because it appears that you have a working system quickly, customers might expect a complete system sooner than is possible. In most cases, the prototype is built on compromises that allow it to come together quickly but that might prevent the prototype from being an effective basis for future development. You need to decide early if you will use the prototype as a basis for future development. All parties need to agree to this decision before development begins.

Be careful that prototyping does not become a disguise for a code and fix development cycle. Before you begin prototyping, gather clear requirements and create a design plan. Limit the amount of time you spend prototyping before you begin. This helps to avoid overdoing the prototyping phase. As you incorporate changes, update the requirements and the current design. After you finish prototyping, you might consider returning to one of the other development models. For example, you might consider prototyping as part of the requirements or design phases of the waterfall model.

LabVIEW Prototyping Methods

There are a number of ways to prototype a system.

In systems with I/O requirements that might be difficult to satisfy, you can develop a prototype to test the control and acquisition loops and rates. In I/O prototypes, random data can simulate data acquired in the real system.

Systems with many user interface requirements are perfect for prototyping. Determining the method you use to display data or prompt the user for settings can be difficult on paper. Instead, consider designing VI front panels with the controls and indicators you need. You might leave the block diagram empty and just talk through the way the controls work and how various actions lead to other front panels. For more extensive prototypes, tie the front panels together. However, be careful not to get too carried away with this process.

If you are bidding on a project for a client, using front panel prototypes can be an extremely effective way to discuss with the client how you might be able to satisfy his or her requirements. Because you can add and remove controls quickly, especially if you avoid developing block diagrams, you can help customers clarify requirements.

Spiral Model

The spiral model is a popular alternative to the waterfall model. It emphasizes risk management so you find major problems earlier in the development cycle. In the waterfall model, you have to complete the design before you begin coding. With the spiral model, you break up the project into a set of risks that need to be dealt with. You then begin a series of iterations in which you analyze the most important risk, evaluate options for resolving the risk, deal with the risk, assess the results, and plan for the next iteration. Figure 1-2 illustrates the spiral lifecycle model.

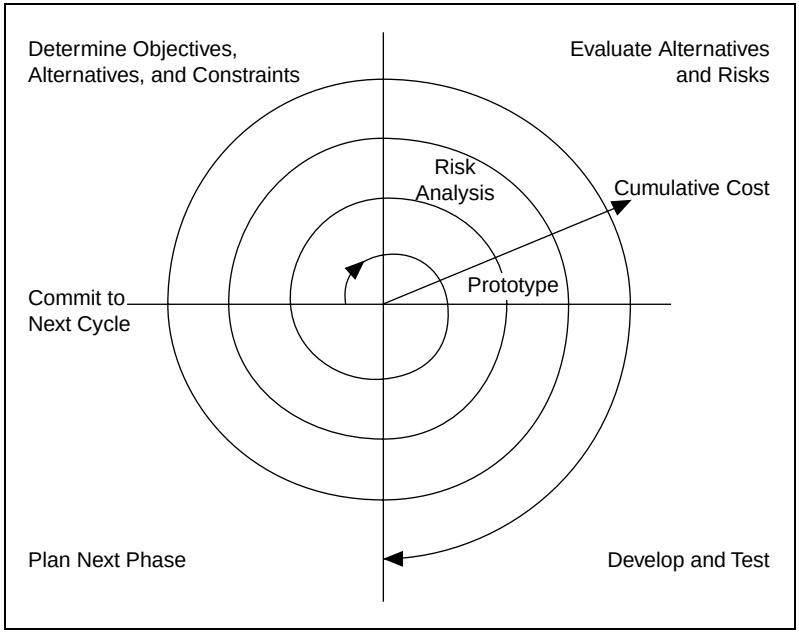

Figure 1-2. Spiral Lifecycle Model

Risks are any issues that are not clearly defined or have the potential to affect the project adversely. For each risk, you need to consider two things: How likely it is to occur (probability) and the severity of its effect on the project (loss). You might use a scale of 1 to 10 for each of these items, with 1 representing the lowest and 10 representing the highest. Risk exposure is the product of these two rankings.

You can use a table to keep track of the top risk items of the project. Table 1-1 gives an example of how to do this.

Table 1-1. Risk Exposure Analysis Example

ID	Risk	Probability	Loss	Risk Exposure	Risk Management Approach
1	Acquisition rates too high	5	9	45	Develop prototype to demonstrate feasibility
2	File format might not be efficient	5	3	15	Develop benchmarks to show speed of data manipulation
3	Uncertain user interface	2	5	10	Involve customer; develop prototype

In general, deal with the risks with the highest risk exposure first. In this example, the first spiral deals with the potential of the data acquisition rates being too high. After the first spiral, you might have demonstrated that the rates are not too high, or you might have to change to a different configuration of hardware to meet the acquisition requirements. Each iteration might identify new risks. In this example, using more powerful hardware might make higher cost a new, more likely risk.

For example, assume you are designing a data acquisition system with a plug-in data acquisition card. In this case, the risk is whether the system can acquire, analyze, and display data quickly enough. Some of the constraints in this case are system cost and requirements for a specific sampling rate and precision.

After determining the options and constraints, you evaluate the risks. In this example, create a prototype or benchmark to test acquisition rates. After you see the results, you can evaluate whether to continue with the approach or choose a different option. You do this by reassessing the risks based on the new knowledge you gained from building the prototype.

In the final phase, you evaluate the results with the customer. Based on customer input, you can reassess the situation, decide on the next highest risk, and start the cycle over. This process continues until the software is finished or you decide the risks are too great and terminate development. You might find that none of the options is viable because each is too expensive, time-consuming, or does not meet the requirements.

The advantage of the spiral model over the waterfall model is that you can evaluate which risks to take care of with each cycle. Because you can evaluate risks with prototypes much earlier than in the waterfall process, you can deal with major obstacles and select alternatives in the earlier stages, which is less expensive. With a standard waterfall model, you might have allowed assumptions about the risky components to spread throughout the design, which requires much more expensive rework when the problems are later discovered.

Summary

Lifecycle models are described as distinct choices from which you must select. In practice, however, you can apply more than one model to a single project. You might start a project with a spiral model to help refine the requirements and specifications over several iterations using prototyping. Once you have reduced the risk of a poorly stated set of requirements, you might apply a waterfall lifecycle model to the design, coding, testing, and maintenance stages.

Other lifecycle models exist. Appendix A, *References*, lists documents that contain information about other development methodologies.

2

Incorporating Quality into the Development Process

This chapter describes strategies for producing quality software.

Many developers who follow the code and fix style of programming described in Chapter 1, *Development Models*, mistakenly believe they do not need to deal with the issue of quality until the testing phase. This is simply not true. Design quality into a product from the start. Developing quality software begins by selecting a development model that helps you avoid problems in the first place. Consider quality during all stages of development: requirements and specification, design, coding, testing, release, and maintenance.

Do not regard quality controls as tedious requirements that impede development. Most of them help streamline development so problems are found before they are in the software, when it is inexpensive to fix them.

Quality Requirements

Set the quality standards for a product during the requirements stage. The desired quality level is treated as a requirement, just like other requirements. Weigh the merits and costs of various options you have for applying quality measures to the project. Some of the trade-offs to consider include speed versus robustness, and ease of use versus power and complexity.

For short projects, used only in-house as tools or quick prototypes, you do not need to emphasize robustness. For example, if you decide to develop a VI to benchmark I/O and graphing speeds, error checking is not as crucial.

However, with more complicated projects that must be reliable, such as applications for monitoring and controlling a factory process, the software must deal with invalid input gracefully. For example, if an operator mistakenly selects invalid voltage or current settings, the application must deal with it appropriately. Institute as many safeguards as possible to prevent problems. Select a lifecycle development model that helps you find

problems as early as possible and allows time for formal reviews and thorough testing.

Configuration Management

Configuration management is the process of controlling changes and ensuring they are reviewed before they are made. Chapter 1, *Development Models*, outlines development models, such as the waterfall model. A central focus of these models is to convert software development from a chaotic, unplanned activity to a controlled process. These models improve software development by establishing specific, measurable goals at each stage of development.

Regardless of how well development proceeds, changes that occur later in the process need to be implemented. It is common for customers introduce new requirements in the design stage. Performance problems discovered during development prompt reevaluation of the design. You might need to rewrite a section of code to correct a problem found in testing. Changes can affect any component of the project from the requirements and specification to the design, code, and tests. If these changes are not made carefully, you might introduce problems that can delay development or degrade quality.

Source Code Control

After setting the project quality requirements, develop a process to deal with changes. This is important for projects with multiple developers. As the developers work on VIs, they need a method for collecting and sharing work. A simple method to deal with this is to establish a central source repository. If each of the developer's computers is networked, you can create a shared location that serves as a central source for development. When developers need to modify files, they can retrieve them from this location. When they are finished with the changes and the system is working, they can return the files to this location.

Common files and areas of overlap introduce the potential for accidental loss of work. If two developers decide to work on the same VI at the same time, only one developer can easily merge changes into the project. The other developer has to use the Compare VIs tool, available in the LabVIEW Professional Development System, to determine the differences and merge the changes into a new version. You might avoid this with good communication, if each developer notifies the others when he or she needs to work on a specific VI. Inevitably, however, mistakes occur, and work is lost.

Source code control tools deal with the problems of sharing VIs and controlling access to avoid accidental loss of data. Source code control tools make it easy to set up shared projects and to retrieve the latest files from the server. Once you have created a project, you can check out a file for development. Checking out a file marks it with your name so that no other developer can modify the file. Other developers can, however, retrieve the current version from the server. A developer can check out the file, make modifications, test the changes, and check in the file to the source code system. After the file is checked in, it is accessible to the whole development team again. Another developer can then check out the file to make further modifications.

Managing Project-Related Files

Source code control tools can manage more than just VIs. You can use them to manage all aspects of the project—requirements, specifications, illustrations, reviews, and other documents related to the project. This ensures that you can control access to these documents and share them as needed. You can use the tools to track changes and access older versions of files.

As described in Chapter 3, *Prototyping and Design Techniques*, source management of all project-related files is extremely important for developing quality software. In fact, source management is a requirement for certification under existing quality standards such as ISO 9000.

Retrieving Old Versions of Files

There are times when you need to retrieve an old version of a file or project. This might happen if you make a change to a file and check it in, only to realize you made a mistake. Another reason you might need to retrieve an old version of a file or project is if you send a beta version of the software to a customer and continue development. If the customer reports a problem, you might need to access a copy of the beta version of the software.

One way to access an old version of files or project is to back up files periodically. However, unless you back up the VI after every change, you might not have access to every version.

Source code control tools provide a way to check in new versions of a file and make a backup copy of the old version. Depending on how you configure the system, the tools can maintain multiple backup copies of a file.

You can use source code control tools to label versions of files with descriptive names like `beta`, `v1.0`, and so on. You can label any number of files and later retrieve all versions of a file with a specific label. When you release a version of the software, label the files specifically for that version.

Tracking Changes

If you are managing a software project, it is important to monitor changes and track progress toward specific milestone objectives. You also can use this information to determine problem areas of a project by identifying which components required a lot of changes.

Source code control tools maintain a log of all changes made to files and projects. When checking in a file, the developer is prompted to enter a summary of the changes made. This summary information is added to the log for that file.

You can view the history information for a file or for the system and generate reports that contain that information.

In addition, if you back up the project at specific checkpoints, you can use the Compare VIs tool to compare the latest version of a project with another version to verify the changes in the project.

Change Control

Large projects might require a formal process for evaluation and approval of each change request. A formal evaluation system like this might be too restrictive, so be selective when choosing the control mechanisms you introduce into the system.

Changes to specific components, such as documents related to user requirements, must be dealt with cautiously because they generally are worked out through several iterations with the customer. In this case, the word *customer* is used in a general sense. You might be your own customer, other departments in your company might be your target audience, or you might develop the software under contract for a third party. When you are your own customer, it is much easier to adjust requirements as you move through the specification and even the design stage. If you are developing for someone else, changing requirements can be extremely difficult.

Source code control tools give you a degree of control when making changes. You can track all changes, and you can configure the system to maintain previous versions so you can back out of changes if necessary.

Some source code control systems give you more options for controlling software change. For example, with Microsoft Visual SourceSafe, Rational Software ClearCase, or Perforce Software Perforce, you can control access to files so some users have access to specific files but others do not. You also can specify that anyone can retrieve files but only certain users can make modifications.

With this kind of access control, you might limit change privileges for requirement documents to specific team members. Or, you might control access so a user has privileges to modify a file only when the change request is approved.

The amount of control you apply can vary throughout the development process. In the early stages of the project, before formal evaluation of the requirements, you do not necessarily need to restrict change access to files nor do you need to follow formal change request processes. Once the requirements are approved, however, you can institute stronger controls. You can apply the same concept of varying the level of control before and after a project phase is complete to specifications, test plans, and code.

Testing Guidelines

Decide up front what level of testing is expected. Engineers under deadline pressure frequently give short attention to testing, devoting more time to other development. Most software engineers suggest a certain level of testing, that is guaranteed to save you time.

Developers must clearly understand the degree to which you expect testing. Also, testing methodologies must be standardized, and results of tests must be tracked. As you develop the requirements and design specifications, also develop a test plan to help you verify that the system and all its components work. Testing reflects the quality goals you want to achieve. For example, if performance is more critical than robustness, develop more tests for performance and fewer that attempt incorrect input, low-memory situations, and so on.

Testing is not an afterthought. Consider testing as part of the initial design phases and test throughout development to find and fix problems as soon as possible.

There are a variety of testing methodologies you can use to help increase the quality of VI projects. The following sections describe some testing methodologies.

Black Box and White Box Testing

The method of black box testing is based on the expected functionality of software, without knowledge of how it works. It is called black box testing because you cannot see the internal workings. You can perform black box testing based largely on a knowledge of the requirements and the interface of a module. For a subVI, you can perform black box tests on the interface of a subVI to evaluate results for various input values. If robustness is a quality goal, include erroneous input data to see if the subVI deals with it well. For example, for numeric inputs, see how the subVI deals with Infinity, Not A Number, and other out-of-range values. Refer to the *Unit Testing* section later in this chapter for more examples.

The method of white box testing is designed with knowledge of the internal workings of the software. Use white box testing to check that all the major paths of execution are exercised. By examining a block diagram and looking at the conditions of Case Structures and the values controlling loops, you can design tests that check those paths. White box testing on a large scale is impractical because it is difficult to test all possible paths.

Although white box testing is difficult to fully implement for large programs, you can choose to test the most important or complex paths. You can combine white box testing with black box testing for more thorough testing of software.

Unit, Integration, and System Testing

Use black box and white box testing to test any component of software, regardless of whether it is an individual VI or the complete application. Unit testing, integration testing, and system testing are phases of the project at which you can apply black box and white box tests.

Unit Testing

You can use unit testing to concentrate on testing individual software components. For example, you might test an individual VI to see that it works correctly, deals with out-of-range data, has acceptable performance, and that all major execution paths in its block diagram are executed and performed correctly. Individual developers can perform unit tests as they work on the modules.

Some examples of common problems unit tests might account for include the following:

- Boundary conditions for each input, such as empty arrays and empty strings, or 0 for a size input. Be sure floating point parameters deal with Infinity and Not A Number.
- Invalid values for each input, such as –3 for a size input.
- Strange combinations of inputs.
- Missing files and bad pathnames.
- What happens when the user clicks the **Cancel** button in a file dialog box?
- What happens if the user aborts the VI?

Define various sets of inputs that thoroughly test the VI and write a test VI that calls the VI with each combination of inputs and checks the results. You can use interactive data logging to create input sets, or test vectors, and replay them interactively to re-test the VI or automatically from a test VI that uses programmatic data retrieval. Refer to the *Unit Testing Validation Procedure* application notes for more information about testing VIs.

To perform unit testing, you might need to *stub out* some components that have not been implemented yet or that are being developed. For example, if you are developing a VI that communicates with an instrument and writes information to a file, another developer can work on a file I/O driver that writes the information in a specific format. To test the components early, you might choose to stub out the file I/O driver by creating a VI with the same interface. This VI can write the data in a format that is easy for you to check. You can test the driver with the real file I/O driver later during the integration phase as described in the following *Integration Testing* section.

Regardless of how you test VIs, record exactly how, when, and what you tested and keep any test VIs you created. This test documentation is especially important if you are creating VIs for paying customers, and it is also useful for yourself. When you revise the VIs, run the existing tests to make sure you have not broken anything. Also update the tests for any new functionality you have added.

Refer to the *LabVIEW Unit Validation Test Procedure* application note for more information about unit testing.

Integration Testing

You perform integration testing on a combination of units. Unit testing usually finds most bugs, but integration testing might reveal unanticipated problems. Modules might not work together as expected. They might interact in unexpected ways because of the way they manipulate shared data. Refer to the *LabVIEW Performance* application note for more information about possible problems that are discovered during testing.

You also can perform integration testing in earlier stages before you put the whole system together. For example, if a developer creates a set of VIs that communicates with an instrument, he or she can develop unit tests to verify that each subVI correctly sends the appropriate commands. He or she also can develop integration tests that use several of the subVIs in conjunction with each other to verify that there is not any unexpected interaction.

Do not perform integration testing as a comprehensive test in which you combine all the components and try to test the top-level program. Doing this can be expensive because it is difficult to determine the specific source of problems within a large set of VIs. Instead, consider testing incrementally with a top-down or bottom-up testing approach.

With a top-down approach, you gradually integrate major components, testing the system with the lower level components of the system disabled, or stubbed out, as described in the *Unit Testing* section earlier in this chapter. Once you have verified that the existing components work together within the existing framework, you can enable additional components.

With a bottom-up approach, you test low-level modules first and gradually work up toward the high-level modules. Begin by testing a small number of components combined into a simple system, such as the driver test described in the *Unit Testing* section earlier in this chapter. After you have combined a set of modules and verified that they work together, add components and test them with the already-debugged subsystem.

The bottom-up approach consists of tests that gradually increase in scope, while the top-down approach consists of tests that are gradually refined as new components are added.

Regardless of the approach you take, you must perform regression testing at each step to verify that the features that already have been tested still work. Regression testing consists of repeating some or all previous tests. Because you might need to perform the same tests numerous times, you might want to develop representative subsets of tests to use for frequent regression tests. You can run these components at each stage, while the

more detailed tests can be run to test an individual set of modules if problems are encountered or as part of a more detailed regression test that is applied periodically during development.

System Testing

System testing happens after integration to determine if the product meets customer expectations and to make sure the software works as expected within the hardware system. You can do this as a set of black box tests to verify that the requirements have been met. Most LabVIEW applications perform some kind of I/O. The application also might communicate with other applications. With system testing, you test the software to make sure it fits into the overall system as expected. When testing the system, ask and answer questions such as the following:

- Are performance requirements met?
- If my application communicates with another application, does it deal with an unexpected failure of that application well?

You can complete this testing with alpha and beta testing. Alpha and beta testing serve to catch test cases that might not have been considered or completed by the developers. With alpha testing, a functionally complete product is tested in-house to see if any problems are found. When alpha testing is complete, the product is beta tested by customers in the field.

Alpha and beta testing are the only testing mechanisms for some companies. This is unfortunate because alpha and beta testing actually can be inexact. Alpha and beta testing are not a substitute for other forms of testing that rigorously test each component to verify that it meets stated objectives. Because this type of testing is done late in the development process, it is difficult and costly to incorporate changes suggested as a result.

Formal Methods of Verification

Some software engineers are proponents of formal verification of software. Other testing methodologies attempt to find problems by exploration, but formal methods attempt to *prove* the correctness of software mathematically.

The principal idea is to analyze each function of a program to determine if it does what you expect. You mathematically state the list of preconditions before the function and the postconditions that are present as a result of the function. You can perform this process either by starting at the beginning of the program and adding conditions as you work through each function or

by starting at the end and working backward, developing a set of weakest preconditions for each function. Refer to Appendix A, *References*, for a list of documents that include more information about the verification process.

This type of testing becomes more complex as more and more possible paths of execution are added to a program through the use of loops and conditions. Many people believe that formal testing presents interesting ideas for looking at software that can help in small cases but that it is impractical for most programs.

Style Guidelines

Inconsistent approaches to development and to user interfaces can be a problem when multiple developers work on a project. Each developer has his or her own style of development, color preferences, display techniques, documentation practices, and block diagram methodologies. One developer might make extensive use of global variables and Sequence Structures while another might prefer to make more use of data flow.

Inconsistent style techniques can create software that, at a minimum, looks bad. Users might become confused and find the user interface VIs difficult to use if the VIs have different behaviors, such as some expecting a user to click a button when he or she is finished and others expecting the user to use a keyboard function key.

Inconsistent style also makes software difficult to maintain. For example, if one developer does not like to use subVIs and decides to develop all features within a single large VI, that VI is difficult to modify.

Establish a set of guidelines for your VI development team. Establish an initial set of guidelines and add additional rules as the project progresses. You can use these style guidelines in future projects.

Chapter 6, *LabVIEW Style Guide*, provides some style recommendations. Use these guidelines as a basis for developing your own style guide. A single standard for programming style in any language really cannot exist because what one group prefers, another group might disagree with. Select a set of guidelines that works for you and your development team.

Design Reviews

Design reviews are a great way to identify and fix problems during development. When the design of a feature is complete, set up a design review with at least one other developer. Discuss quality goals, asking questions such as the following:

- Does the design incorporate testing?
- Is error handling built-in?
- Are there any assumptions in the system that might be invalid?

Also, look at the design with an eye for features that are essential as opposed to features that are extras. There is nothing wrong with building in extra features. If quality and schedule are important, however, ensure that these extra features are scheduled for late in the development process, so they can be dropped, or moved to the list of features for subsequent releases. Document the results of the design review and any recommended changes.

Code Walkthroughs

A code walkthrough is similar to a design review except that it analyzes the code instead of the design. To perform a code review, give one or more developers printouts of the VIs to review. You might want to perform the review online because VIs are easier to read and navigate online. It is wise for the designer talk through the design. The reviewers compare the description to the actual implementation. The reviewers must consider many of the same issues included in a design review. During a code walkthrough, many of the following questions might be asked and answered:

- What happens if a specific VI or function returns an error? Are errors dealt with and/or reported correctly?
- Are there any race conditions? An example of a race condition is a block diagram that reads from and writes to a global variable. There is the potential that a parallel block diagram simultaneously attempts to manipulate the same global variable, resulting in loss of data.

- Is the block diagram implemented well? Are the algorithms efficient in terms of speed and/or memory usage? Refer to the *LabVIEW Performance* application note for more information about creating effective code.
- Is the block diagram easy to maintain? Has the developer made good use of hierarchy, or is he or she placing too much functionality in a single VI? Does the developer adhere to established guidelines?

There are a number of other features you can look for in a code walkthrough. Take notes on the problems you encounter and add them to a list you can use as a guideline for other walkthroughs.

Focus on technical issues when doing a code walkthrough. Remember to review only the code, not the developer who produced it. Try not to focus only on the negative and be sure to raise positive points.

Refer to Appendix A, *References*, for a list of documents that include more information about walkthrough techniques.

Postmortem Evaluation

At the end of each stage in the development process, consider having a postmortem meeting to discuss what has gone well and what has not. Each developer must evaluate the project honestly and discuss obstacles that reduce the quality level of the project. Each developer must consider the following questions:

- What are we doing right? What is working well?
- What are we doing wrong? What can we improve?
- Are there specific areas of the design/code that need a lot of work? Is a design review or code walkthrough of that section necessary?
- Are the quality systems working? Can we catch more problems if we changed the quality requirements? Are there better ways to get the same results?

Postmortem meetings at major milestones can help to correct problems mid-schedule instead of waiting until the release is complete.

Software Quality Standards

As software has become a more critical component in systems, concerns about software quality have increased. Consequently, a number of organizations have developed quality standards that are specific to software or that can be applied to software. When developing software for some large organizations, especially government organizations, you might be required to follow one of these standards.

The following sections include a brief overview of the most popular standards. Refer to Appendix A, *References*, for a list of documents that include more information about these standards.

International Organization for Standardization ISO 9000

The International Organization for Standardization (ISO) developed the ISO 9000 family of standards for quality management and assurance. Many countries have adopted these standards. In some cases, governmental bodies require compliance with this ISO standard. Compliance generally is measured by certification performed by a third-party auditor. The ISO 9000 family is widely used within Europe and Asia. It has not been widely adopted within the United States, although many companies and some government agencies are begining to use it.

In each country, the ISO family of standards are referred to by slightly different names. For example, in the United States it has been adopted as the ANSI/American Society for Quality Control (ASQC) Q90 Series. In Europe, it has been adopted by the European Committee for Standardization (CEN) and the European Committee for Electrotechnical Standardization (CENELEC) as the European Norm (EN) 29000 Series. In Canada, it has been adopted by the Canadian Standards Association (CSA) as the Q 9000 series. However, it is most commonly referred to as ISO 9000 in all countries.

ISO 9000 is an introduction to the ISO 9000 family of standards. ISO 9001 is a model for quality assurance in design, development, production, installation, and servicing. Its focus on design and development makes it the most appropriate standard for software products.

Because the ISO 9000 family is designed to apply to any industry, it is somewhat difficult to apply to software development. ISO 9000.3 is a set of guidelines designed to explain how to apply ISO 9001 specifically to software development.

ISO 9001 does not dictate software development procedures. Instead, it requires documentation of development procedures and adherence to the standards you set. Conformance with ISO 9001 does not guarantee quality. Instead, the idea behind ISO 9001 is that companies that emphasize quality and follow their documented practices produce higher quality products than companies that do not.

U.S. Food and Drug Administration Standards

The U.S. Food and Drug Administration (FDA) requires all software used in medical applications to meet its Current Good Manufacturing Practices (CGMP). One of the goals of the standard is to make it as consistent as possible with ISO 9001 and a supplement to ISO 9001, ISO/CD 13485. These FDA standards are largely consistent with ISO 9001, but there are some differences. Specifically, the FDA did not think ISO 9001 was specific enough about certain requirements, so the FDA clearly outlined them in its rules.

Refer to the FDA web site at `http://www.fda.gov` for more information about the CGMP rules and how they compare to ISO 9001.

Capability Maturity Model (CMM)

In 1984, the United States Department of Defense created the Software Engineering Institute (SEI) to establish standards for software quality. The SEI developed a model for software quality called the Capability Maturity Model (CMM). The CMM focuses on improving the maturity of an organization's processes.

Whereas ISO establishes only two levels of conformance, pass or fail, the CMM ranks an organization into one of five categories.

Level 1. Initial—The organization has few defined processes; quality and schedules are unpredictable.

Level 2. Repeatable—The organization establishes policies based on software engineering techniques and previous projects that allow repeated success. Groups use configuration management tools to manage projects. Also, they track software costs, features, and schedules. Project standards are defined and followed. Although the groups can deal with similar projects based on this experience, their processes might not be mature enough to deal with significantly different types of projects.

Level 3. Defined—The organization establishes a baseline set of policies for all projects. Groups are well trained and know how to customize this set of policies for specific projects. Each project has well-defined characteristics that make it possible to accurately measure progress.

Level 4. Managed—The organization sets quality goals for projects and processes and measures progress toward those goals.

Level 5. Optimizing—The organization emphasizes continuous process improvement across all projects. The organization evaluates the software engineering techniques it uses in different groups and applies them throughout the organization.

Figure 2-1 illustrates the five levels of the CMM and the processes necessary for advancement to the next level.

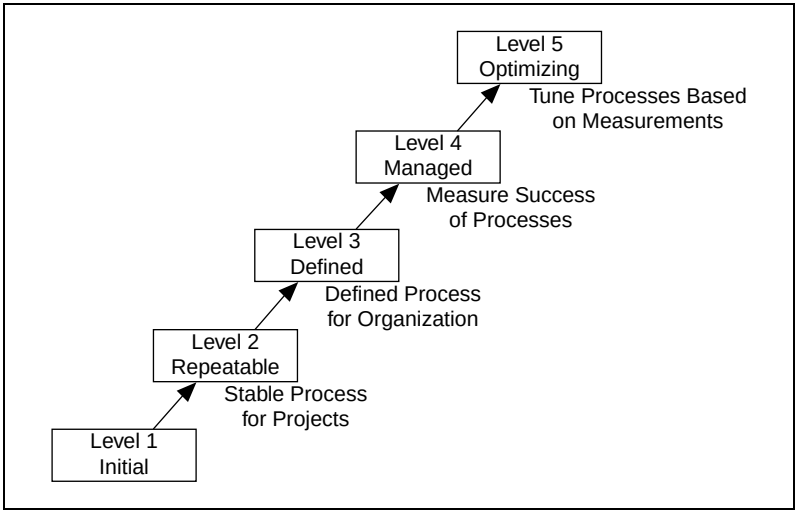

Figure 2-1. Capability Maturity Model

Most companies are at Level 1 or 2. The U.S. Department of Defense prefers a Level 3 or higher CMM assessment in bids on new government software development. Some commercial companies, mainly in the United States, also use the CMM.

The CMM differs from ISO 9001 in that it is software specific. Also, the ISO specifications are fairly high-level documents. ISO 9001 is only a few pages. CMM is very detailed, with more than 500 pages.

Institute of Electrical and Electronic Engineers (IEEE) Standards

IEEE defined a number of standards for software engineering. IEEE Standard 730, first published in 1980, is a standard for software quality assurance plans. This standard serves as a foundation for several other IEEE standards and gives a brief description of the minimum requirements for a quality plan in the following areas:

- Purpose
- Reference documents
- Management
- Documentation
- Standards, practices, conventions, and metrics
- Reviews and audits
- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Code control
- Media control
- Supplier control
- Records collection, maintenance, and retention
- Training
- Risk management

As with the ISO standards, IEEE 730 is fairly short. It does not dictate how to meet the requirements but requires documentation for these practices to a specified minimum level of detail.

In addition to IEEE 730, several other IEEE standards related to software engineering exist, including the following:

- IEEE 610—Defines standard software engineering terminology.
- IEEE 829—Establishes standards for software test documentation.
- IEEE 830—Explains the content of good software requirements specifications.
- IEEE 1074—Describes the activities performed as part of a software lifecycle without requiring a specific lifecycle model.
- IEEE 1298—Details the components of a software quality management system; similar to ISO 9001.

Your projects might be required to meet some or all these standards. Even if you are not required to develop to any of these specifications, they can be helpful in developing your own requirements, specifications, and quality plans.

3

Prototyping and Design Techniques

This chapter gives you pointers for project design, including programming approaches, prototyping, and benchmarking.

When you first begin a programming project, deciding how to start can be intimidating. Many LabVIEW developers start immediately with a code and fix development process, building some of the VIs they think are needed. Then they realize they actually need something different from what they have built already. Consequently, a lot of code is developed, reworked, or thrown away unnecessarily.

Clearly Define the Requirements of the Application

Before you develop a detailed design of a system, define goals as clearly as possible. Begin by making a list of requirements. Some requirements are specific, such as the types of I/O, sampling rates, or the need for real-time analysis. You need to do some research at this early stage to be sure you can meet the specifications. Other requirements depend on user preferences, such as file formats or graph styles.

Try to distinguish between absolute requirements and desires. You might be able to satisfy all requests, but it is best to have an idea about what you can sacrifice if you run out of time.

Also, be careful that the requirements are not so detailed that they constrain the design. For example, when you design an I/O system, the customer probably has certain sampling rate and precision requirements. He or she also is constrained by cost. Include those issues in the requirements. However, if you can avoid specifying the operating system and hardware, you can adjust the design after you begin prototyping and benchmarking various components. As long as the costs are within budget and the timing and precision issues are met, the customer might not care whether the system uses a particular type of plug-in card or other hardware.

Another example of overly constraining a design is to be too specific about the format for display used in various screens with which the customer interacts. A picture of a display might be useful to explain requirements, but be clear about whether the picture is a requirement or a guideline. Some designers go through significant difficulties trying to produce a system that behaves in a specific way because a certain behavior was a requirement. In this case, there might be a simpler solution that produces the same results at a much lower cost in a shorter time.

Top-Down Design

The block diagram programming metaphor LabVIEW uses was designed to be easy to understand. Most engineers already use block diagrams to describe systems. The goal of the block diagram is to make it easier for you to move from the system block diagrams you create to executable code.

The basic concept is to divide the task into manageable pieces at logical places. Begin with a high-level block diagram that describes the main components of the VI. For example, you might have a block diagram that consists of a block for configuration, a block for acquisition, a block for analysis of the acquired data, a block for displaying the results, a block for saving the data to disk, and a block to clean up at the end of the VI.

After you determine the high-level blocks, create a block diagram that uses those blocks. For each block, create a new *stub* VI, which is a non-functional prototype that represents a future subVI. Create an icon for this stub VI and create a front panel with the necessary inputs and outputs. You do not have to create a block diagram for this VI yet. Instead, define the interface and see if this stub VI is a useful part of the top-level block diagram.

After you assemble a group of these stub VIs, determine the function of each block and how it works. Ask yourself whether any given block generates information that a subsequent VI needs. If so, make sure the top-level block diagram sketch contains wires to pass the data between the VIs. You can document the functionality of the VI and the inputs and outputs using the **Documentation** page of the **VI Properties** dialog box in LabVIEW.

In analyzing the transfer of data from one block to another, try to avoid global variables because they hide the data dependency among VIs and might introduce race conditions. Refer to the *LabVIEW Performance* application notes for more information about issues that might arise when

creating VIs. As the VI becomes larger, it becomes difficult to debug if you use global variables as the method of transferring information among VIs.

Continue to refine the design by breaking down each of the component blocks into more detailed outlines. You can do this by going to the block diagram of what was once a stub VI and filling out its block diagram, placing lower level stub VIs on the block diagram that represent each of the major actions the VI must perform.

Be careful not to jump too quickly into implementing the system at this point. One of the objectives here is to gradually refine the design so you can determine if you have left out any necessary components at higher levels. For example, when refining the acquisition phase, you might realize there is more information you need from the configuration phase. If you completely implement one block before you analyze a subsequent block, you might need to redesign the first block significantly. It is better to try to refine the system gradually on several fronts, with particular attention to sections that have more risk because of the complexity.

Data Acquisition System Example

This example describes how you might apply top-down design techniques to a data acquisition system. This system must let the user provide some configuration of the acquisition, such as rates, channels, and so on; acquire data; process the data; and save the data to disk.

Start to design the VI hierarchy by breaking the problem into logical pieces. The flowchart in Figure 3-1 shows several major blocks you can expect to see in one form or another in every data acquisition system.

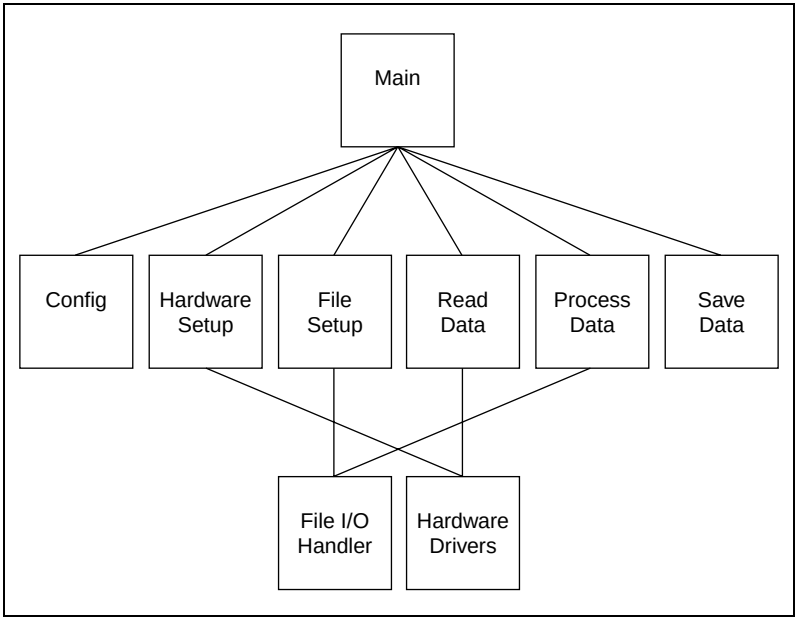

Figure 3-1. Flowchart of a Data Acquisition System

Think about the data structures needed, asking questions such as "What information needs to accompany the raw data values from the Read Data VI to the Save Data VI?" This might imply a cluster array, which is an array of many channels, each element of which is a cluster that contains the value, the channel name, scale factors, and so on. A method that performs some action on such a data structure is called an algorithm. Algorithms and data structures are intertwined. This is reflected in modern structured programming, and it works well in LabVIEW. If you like to use pseudocode, try that technique as well. Figures 3-2 and 3-3 show a relationship between pseudocode and LabVIEW structures.

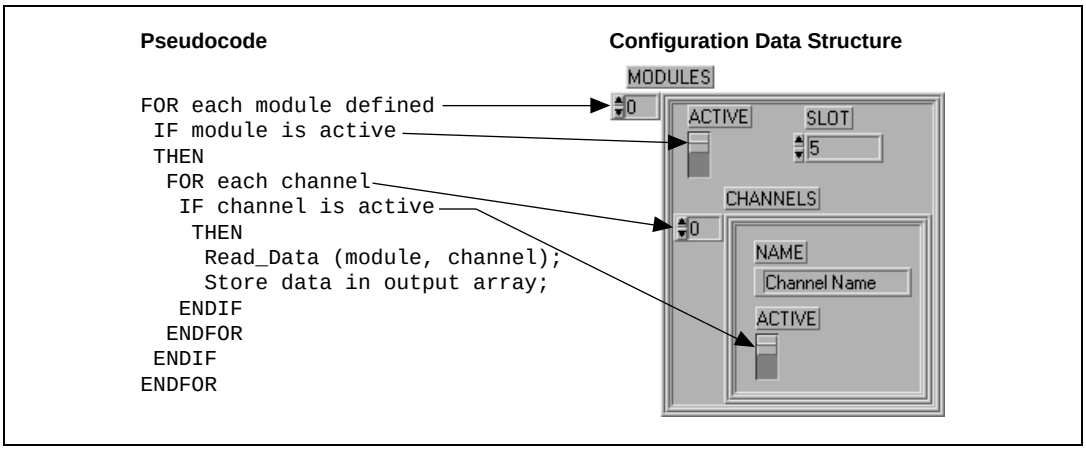

Figure 3-2. Mapping Pseudocode into a LabVIEW Data Structure

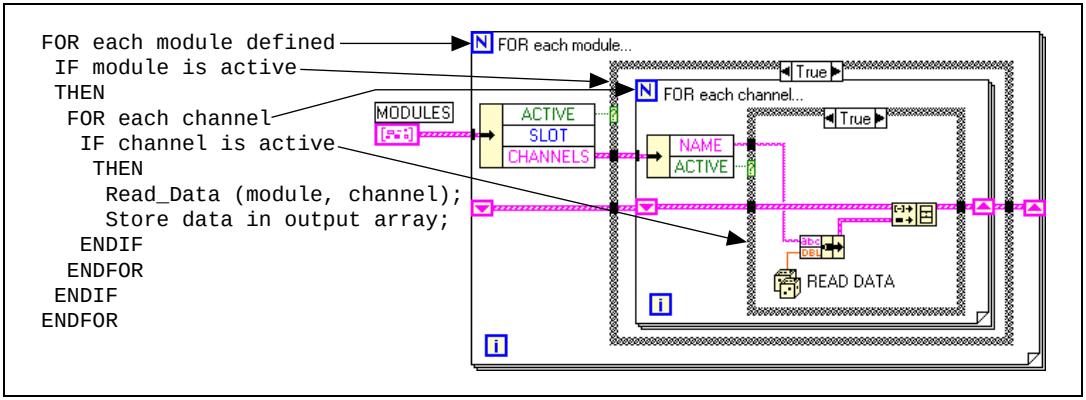

Figure 3-3. Mapping Pseudocode into Actual LabVIEW Code

Notice that the program and the data structure correspond in Figure 3-2.

Many experienced LabVIEW users prefer to use sketches of LabVIEW code. You can draw caricatures of the familiar structures and wire them together on paper. This is a good way to think things through, sometimes with the help of other LabVIEW programmers.

If you are not sure how a certain function will work, prototype it in a simple test VI, as shown in Figure 3-4. Artificial data dependency between the VIs and the main While Loop in Figure 3-4 eliminates the need for a Sequence Structure.

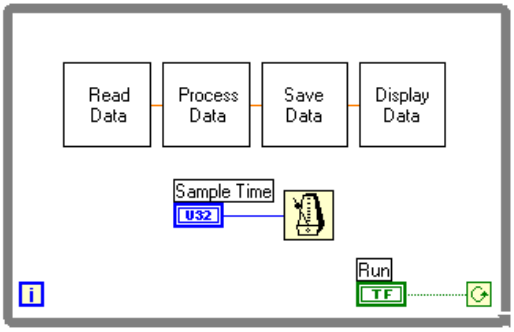

Figure 3-4. Data Flow for a Generic Data Acquisition Program

Finally, you are ready to write the program in LabVIEW. Remember to make the code modular, building subVIs when there is a logical division of labor or the potential for code reuse. Solve the more general problems along with the specific ones. Test the subVIs as you write them. This might involve constructing higher level test routines. It is much easier to catch the problems in one small module than in a large hierarchy of VIs.

Bottom-Up Design

Usually, avoid bottom-up system design. It is sometimes useful when used in conjunction with top-down design, with bottom-up design, you start by building the lower level components and then progressing up the hierarchy, gradually putting pieces together until you have the complete system.

The problem with bottom-up design is that because you do not start with a clear idea of the big picture, you might build pieces that do not fit together the way you expect.

There are specific cases in which using bottom-up design is appropriate. If the design is constrained by low-level functionality, you might need to build that low-level functionality first to get an idea of how it can be used. This might be true of an instrument driver, where the command set for the instrument constrains you in terms of when you can do certain operations. For example, with a top-down design, you might break up the design so configuration of the instrument and reading a measurement from the instrument are done in distinct VIs. The instrument command set might turn out to be more constraining than you thought, requiring you to combine these operations. In this case, with a bottom-up strategy, you might start by building VIs that deal with the instrument command set.

In most cases, use a top-down design strategy. You might mix in some components of bottom-up design, if necessary. Thus, in the case of an instrument driver, you might use a risk-minimization strategy to understand the limitations of the instrument command set and develop the lower level components. Then you might use a top-down approach to develop the high-level blocks.

The following example shows in more detail how you can apply this technique to the process of designing a driver for a GPIB instrument.

Instrument Driver Example

A complex GPIB-controlled instrument can have hundreds of commands, many of which interact with each other. A bottom-up approach might be the most effective way to design a driver for such an instrument. The key here is that the problem is detail driven. You must learn the command set and design a front panel that is simple for the user yet gives full control of the instrument functionality. Design a preliminary VI hierarchy, preferably one based on similar instrument drivers. You must satisfy the user's needs. Designing a driver requires more than putting knobs on GPIB commands. The example chosen here is the Tektronix 370A Curve Tracer. It has about 100 GPIB commands if you include the read and write versions of each one.

Once you begin programming, the hierarchy fills out naturally, one subVI at a time. Add lower level support VIs as required, such as a communications handler, a routine to parse a complex header message, or an error handler. For instance, the 370A requires a complicated parser for the waveform preamble that contains information such as scale factors, offsets, sources, and units. It is much cleaner to bury this operation in a subVI than to let it obscure the function of a higher level VI. Also, a communications handler makes it simple to exchange messages with the instrument. Such a handler formats and sends the message, reads the response if required, and checks for errors.

Once the basic functions are ready, assemble them into a demonstration driver VI that makes the instrument do something useful. It quickly finds any fundamental flaws in earlier choices of data structures, terminal assignments, and default values.

Refer to the instrument drivers *LabVIEW Help* for more information about developing.

The top-level VI in Figure 3-5 is an automated test example. It calls nine of the major functions included in the driver package. Each function, in turn, calls subVIs to perform GPIB I/O, file I/O, or data conversion.

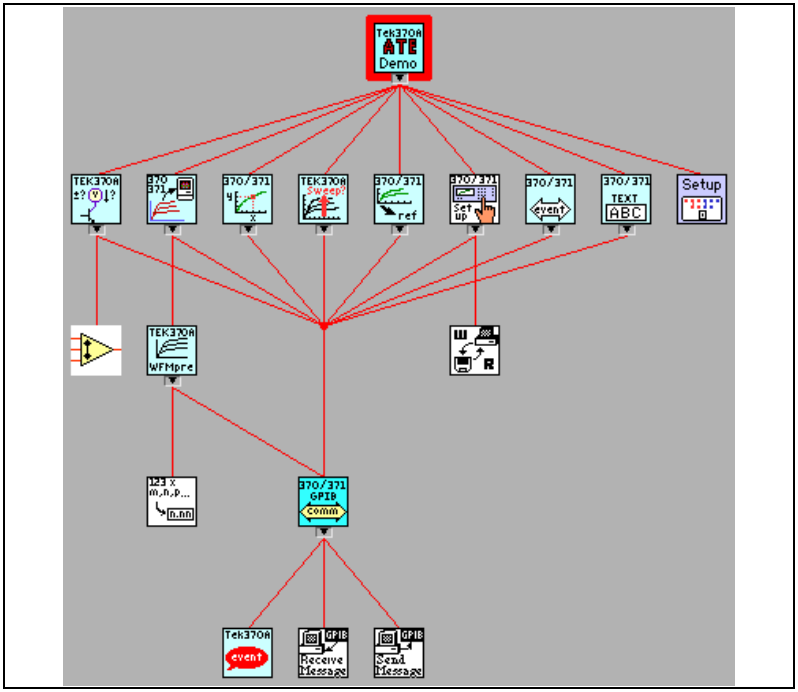

Figure 3-5. VI Hierarchy for the Tektronix 370A

Designing for Multiple Developers

One of the main challenges in the planning stage is to establish discrete project areas for each developer. As you design the specification and architectural design, begin to see areas that have a minimal amount of overlap. For example, a complicated data monitoring system might have one set of VIs to display and manipulate data and another set to acquire the information and transfer it to disk. These two modules are substantial, do not overlap, and can be assigned to different developers.

Inevitably, there is some interaction among the modules. One of the principal objectives of the early design work is to design how those modules interact with each other. The data display system must access the data it needs to display. The acquisition component needs to provide this information for the other module. At an early stage in development, you

might design the connector panes of VIs needed to transfer information between the two modules. Likewise, if there are global data structures that must be shared, analyze and define them early in the architectural design stage before the individual developers begin work on the components.

In the early stages, each developer can create stub VIs with the connector pane interface that was defined for the shared module. This stub VI can do nothing, or, if it is a VI that returns information, you might have it generate random data. This allows each member of the development team to continue development without having to wait for the other modules to be finished. It also makes it easy for the individuals to perform unit testing of modules as described in Chapter 2, *Incorporating Quality into the Development Process*.

As components near completion, you can integrate the modules by replacing the stub components with the real counterparts. At this point you can perform integration testing to verify the system works as a whole. Refer to the *Integration Testing* section in Chapter 2, *Incorporating Quality into the Development Process*, for more information about integration testing.

Front Panel Prototyping

As mentioned in Chapter 1, *Development Models*, front panel prototypes can provide insight into the organization of the program. Assuming the program is user-interface intensive, you can attempt to create a mock interface that represents what the user sees.

Avoid implementing block diagrams in the early stages of creating prototypes so you do not fall into the code and fix trap. Instead, create just the front panels. As you create buttons, listboxes, and rings, think about what needs to happen as the user makes selections. Ask yourself questions such as the following:

- Should the button lead to another front panel?
- Should some controls on the front panel be hidden and replaced by others?

If new options are presented, follow those ideas by creating new front panels to illustrate the results. This kind of prototyping can help solidify the requirements for a project and give you a better idea of its scope.

Prototyping cannot solve all development problems, however. You have to be careful how you present the prototype to customers. Prototypes can give

an overly inflated sense that you are rapidly making progress on the project. You have to be clear to the customer, whether it is an external customer or other members of your company, that this prototype is strictly for design purposes and that much of it is reworked in the development phase.

Another danger in prototyping is that you might overdo it. Consider setting strict time goals for the amount of time you prototype a system to prevent yourself from falling into the code and fix trap.

Of course, front panel prototyping deals only with user interface components. As described here, it does not deal with I/O constraints, data types, or algorithm issues in the design. The front panel issues might help you better define some of these areas because it gives you an idea of some of the major data structures you need to maintain, but it does not deal with all these issues. For those issues, you need to use one of the other methods described in this chapter, such as performance benchmarking and top-down design.

Performance Benchmarking

For I/O systems with a number of data points or high transfer rate requirements, test the performance-related components early because the test might prove that the design assumptions are incorrect.

For example, if you plan to use an instrument as the data acquisition system, you might want to build some simple tests that perform the type of I/O you plan to use. While the specifications might seem to indicate that the instrument can handle the application you are creating, you might find that triggering, for example, takes longer than you expected, that switching between channels with different gains cannot be done at the necessary rate without reducing the accuracy of the sampling, or that even though the instrument can handle the rates, you do not have enough time on the software side to perform the desired analysis.

A simple prototype of the time-critical sections of the application can help reveal this kind of problem. The timing template example in the `examples/general/structs.llb` directory illustrates how to time a process. Because timings can fluctuate from one run to another for a variety of reasons, put the operation in a loop and display the average execution time. You also can use a graph to display timing fluctuations. Causes of timing fluctuations can include system interrupts, screen updates, user interaction, and initial buffer allocation.

Identify Common Operations

As you design programs, you might find that certain operations are performed frequently. Depending on the situation, this might be a good place to use subVIs or loops to repeat an action.

For example, consider Figure 3-6, where three similar operations run independently.

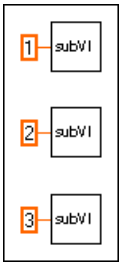

Figure 3-6. Operations Run Independently

An alternative to this design is a loop that performs the operation three times, as shown in Figure 3-7. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the loop.

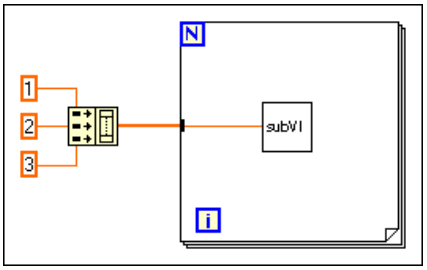

Figure 3-7. Loop Performs Operation Three Times

If the array elements are constant, you can use an array constant instead of building the array on the block diagram.

Some users mistakenly avoid using subVIs because they are afraid of the overhead it might add to the execution time. It is true that you probably do not want to create a subVI from a simple mathematical operation such as the Add function, especially if it must be repeated thousands of times.

However, the overhead for a subVI is fairly small and usually is dwarfed by any I/O you perform or by any memory management that might occur from complex manipulation of arrays.

4

Scheduling and Project Tracking

This chapter describes techniques for estimating development time and using those estimates to create schedules. This chapter also distinguishes between an estimate, which reflects the time required to implement a feature, and a schedule, which reflects how you fulfill that feature. Estimates are commonly expressed in ideal person-days, or 8 hours of work. In creating a schedule from estimates, you must consider dependencies—one project might have to be completed before another can begin—and other tasks, such as meetings, support for existing projects, and so on.

Estimation

One of the principle tasks of planning is to estimate the size of the project and fit it into the schedule because most projects are at least partially driven by a schedule. Schedule, resources, and critical requirements interact to determine what you can implement in a release.

Unfortunately, when it comes to estimating software schedules accurately, few people are successful. Major companies have had software projects exceed original estimates by a year or more. Poor planning or an incomplete idea of project goals often causes deadlines to be missed. Another major cause of missed schedules is *feature creep*: The design gradually grows to include features that were not part of the original requirements. In many cases, the delays in schedule are a result of using a code and fix development process rather than a more measurable development model.

Off-the-cuff estimates are almost never accurate for the following reasons:

- People are usually overly optimistic. An estimate of two months at first might seem like an infinite amount of time. During the last two weeks of the project, when developers find themselves working many overtime hours, it becomes clear that it is not.
- The objectives, implementation issues, and quality requirements are not understood clearly. When challenged with the task of creating a data monitoring system, an engineer might estimate two weeks. If the product is designed by the engineer and for the engineer, this estimate

might be right. However, if it is for other users, he or she probably is not considering requirements that might be assumed by a less knowledgeable user but never are specified clearly.

For example, VIs need to be reliable and easy to use because the engineer is not going to be there to correct them if a problem occurs. A considerable amount of testing and documentation is necessary. Also, the user needs to save results to disk, print reports, and view and manipulate the data on screen. If he or she has not discussed or considered the project in detail, the engineer is setting himself or herself up for failure.

- Day-to-day tasks are ignored. There are meetings and conferences to attend, holidays, reports to write, existing projects to maintain, and other tasks that make up a standard work week.

Accurate estimates are difficult because of the imprecise nature of most software projects. In the initial phase of a project, complete requirements are not known. The way you implement those requirements is even less clear. As you clarify the objectives and implementation plans, you can make more realistic estimates.

The following sections outline some of the current best-practice estimation techniques in software engineering. All these techniques require breaking the project down into more manageable components you can estimate individually. There are other methods of estimating development time. Refer to Appendix A, *References*, for a list of documents that describe these and other estimation techniques in more detail.

Source Lines of Code/Number of Nodes Estimation

Software engineering documentation frequently refers to source lines of code (SLOC) as a measurement, or metric, of software complexity. SLOC as a measurement of complexity is popular in part because the information is easy to gather. Numerous programs exist for analyzing text-based programming languages to measure complexity. In general, SLOC measurements include every line of source code developed for a project, excluding comments and blank lines.

The VI Metrics tool, included in the LabVIEW Professional Development System, provides a method for measuring a corresponding metric for LabVIEW code. The VI Metrics tool counts the *number of nodes* used within a VI or within a hierarchy of VIs. A node is almost any object on a block diagram excluding labels and graphics but including functions, VIs, and structures, such as loops and sequences. Refer to the *LabVIEW Help*

for more information about how to use this tool and the accounting mechanism it uses.

You can use number of nodes as a method for estimating future project development efforts. For this to work, you must build a base of knowledge about current and previous projects. You must have an idea of the amount of time it took to develop components of existing software products and associate that information with the number of nodes used in that component.

Armed with this historical information, you next need to estimate the number of nodes required for a new project. It is not possible to do this for an entire project at once. Instead, you must break the project down into subprojects you can compare to other tasks completed in the past. Once you have broken it down, you can estimate each component and produce a total estimate of the number of nodes and the time required for development.

Problems with Source Lines of Code and Number of Nodes

Size-based metrics are not uniformly accepted in software engineering. Many people favor them because it is a relatively easy metric to gather and because a lot of literature has been written about it. Detractors of size metrics point out the following flaws:

- Size-based metrics are dependent on the organization. Lines of code/numbers of nodes can be useful within an organization as long as you are dealing with the same group of people and they are following the same style guidelines. Trying to use size metrics from other companies/groups can be difficult because of differing levels of experience, different expectations for testing and development methodologies, and so on.
- Size-based metrics are also dependent on the programming language. Comparing a line of code in assembly language to one written in C can be like comparing apples to oranges. Statements in higher level languages can provide more functionality than those in lower level languages. Comparing numbers of nodes in LabVIEW to lines of code in a text-based programming language can be inexact for this reason.
- Not all code is created with the same level of quality. A VI that retrieves information from a user and writes it to a file can be written so efficiently that it involves a small number of nodes or it can be written poorly with a large number of nodes.

- Not all code is equal in complexity. An add function is much easier to use than an array index node. A block diagram that consists of 50 nested loops is much more difficult to understand than 50 subVIs connected together in a line.
- Size-based metrics rely on a solid base of information that associates productivity with various projects. To be accurate, have statistics for each member of a team because the experience level of team members varies.

Despite these problems, size metrics are used widely for estimating projects. A good technique is to estimate a project using size metrics in conjunction with one of the other methods described later in this chapter. The two different methods can complement each other. If you find differences between the two estimates, analyze the assumptions in each to determine the source of the discrepancy.

Effort Estimation

Effort estimation is similar in many ways to number of nodes estimation. You break down the project into components that can be more easily estimated. A good guideline is to break the project into tasks that take no more than a week to complete. More complicated tasks are difficult to estimate accurately.

Once you have broken down the project into tasks, you can estimate the time to complete each task and add the results to calculate an overall cost.

Wideband Delphi Estimation

You can use wideband delphi estimation in conjunction with any of the other estimation techniques this chapter describes to achieve more reliable estimates. For successful wideband delphi estimation, multiple developers must contribute to the estimation process.

First divide the project into separate tasks. Then meet with other developers to explain the list of tasks. Avoid discussing time estimates during this early discussion.

Once you have agreed on a set of tasks, each developer separately estimates the time it takes to complete each task using uninterrupted person-days as the unit of estimation. The developers need to list any assumptions made in forming estimates. The group then reconvenes to graph the overall estimates as a range of values. It is a good idea to keep the estimates

anonymous and to have a person outside the development team lead this meeting.

After graphing the original set of values, each developer reports any assumptions made in determining the estimate. For example, one developer might have assumed a certain VI project takes advantage of existing libraries. Another developer might point out that a specific VI is more complicated than expected because it involves communicating with another application or a shared library. Another team member might be aware of a task that involves an extensive amount of documentation and testing.

After stating assumptions, each developer reexamines and adjusts the estimates. The group then graphs and discusses the new estimates. This process might go on for three or four cycles.

In most cases, you converge to a small range of values. Absolute convergence is not required. After the meeting, the developer in charge of the project can use the average of the results, or he or she might ignore certain outlying values. If some tasks turn out to be too expensive for the time allowed, he or she might consider adding resources or scaling back the project.

Even if the estimate is incorrect, the discussion from the meetings gives a clear idea of the scope of a project. The discussion serves as an exploration tool during the specification and design part of the project so you can avoid problems later.

Refer to Appendix A, *References* for a list of documents that include more information about the wideband delphi estimation method.

Other Estimation Techniques

Several other techniques exist for estimating development cost. These are described in detail in some of the documents listed in Appendix A, *References*. The following list briefly describes some popular techniques:

- Function-Point Estimation—Function-point estimation differs considerably from the size-estimation techniques described so far. Rather than divide the project into tasks that are estimated separately, function points are based on a formula applied to a category breakdown of the project requirements. The requirements are analyzed for features such as inputs, outputs, user inquiries, files, and external interfaces. These features are tallied, and each is weighted. The results are added to produce a number that represents the complexity of the project. You can compare this number to function-point estimates of previous projects to determine an estimate.

Function-point estimates were designed primarily with database applications in mind but have been applied to other software areas as well. Function-point estimation is popular as a rough estimation method because it can be used early in the development process based on requirements documents. However, the accuracy of function points as an estimation method has not been thoroughly analyzed.

- COCOMO Estimation—COCOMO (COnstructive COst MOdel) is a formula-based estimation method for converting software size estimates to estimated development time. COCOMO is a set of methods that range from basic to advanced. Basic COCOMO makes a rough estimate based on a size estimate and a simple classification of the project type and experience level of a team. Advanced COCOMO takes into account reliability requirements, hardware features and constraints, programming experience in a variety of areas, and tools and methods used for developing and managing the project.

Mapping Estimates to Schedules

An estimate of the amount of effort required for a project can differ greatly from the calendar time needed to complete the project. You might accurately estimate that a VI takes only two weeks to develop. However, in implementation you must fit that development into your schedule. You might have other projects to complete first, or you might need to wait for another developer to complete his or her work before you can start the project. You might have meetings and other events during that time also.

Estimate project development time separately from scheduling it into your work calendar. Consider estimating tasks in *ideal* person-days, which correspond to 8 hours of development without interruption.

After estimating project time, try to develop a schedule that accounts for overhead estimates and project dependencies. Remember that you have weekly meetings to attend, existing projects to support, reports to write, and other responsibilities.

Record progress meeting time estimates and schedule estimates. Track project time and time spent on other tasks each week. This information might vary from week to week, but be able to determine an average that is a useful reference for future scheduling. Recording more information helps you plan future projects accurately.

Tracking Schedules Using Milestones

Milestones are a crucial technique for gauging progress on a project. If completing the project by a specific date is important, consider setting milestones for completion.

Set up a small number of major milestones for the project, making sure each one has clear requirements. To minimize risk, set milestones to complete the most important components first. If, after reaching a milestone, the project falls behind schedule and there is not enough time for another milestone, the most important components are already complete.

Throughout development, strive to keep the quality level high. If you defer problems until a milestone is reached, you are, in effect, deferring risks that might delay the schedule. Delaying problems can make it seem like you are making more progress than you actually are. Also, it can create a situation where you attempt to build new development on top of an unstable foundation.

When working toward a major milestone, set smaller goals to gauge progress. Derive minor milestones from the task list you created as part of your estimation.

Refer to Appendix A, *References*, for a list of documents that include more information about major and minor milestones.

Responding to Missed Milestones

One of the biggest mistakes people make is to miss a milestone and not reexamine the project as a consequence. After missing a milestone, many developers continue on the same schedule, assuming they can work harder and make up the time.

Instead, if you miss a milestone, evaluate the reasons you missed it. Is there a systematic problem that might affect subsequent milestones? Is the specification still changing? Are quality problems slowing down new development? Is the development team at risk of burning out from too much overtime?

Consider problems carefully. Discuss each problem or setback and have the entire team make suggestions on how to get back on track. Avoid accusations. You might have to stop development and return to design for a period of time. You might decide to cut back on certain features, stop adding new features until all the problems are fixed, or renegotiate the schedule.

Deal with problems as they arise and monitor progress to avoid repeating mistakes or making new ones. Do not wait until the end of the milestone or the end of the project to correct problems.

Missing a milestone should not come as a complete surprise. Schedule delays do not occur all at once. They happen little by little, day by day. Correct problems as they arise. If you do not realize you are behind schedule until the last two months of a year-long project, you probably can not get back on schedule.

5

Creating Documentation

Taking the time to create quality documentation can mean the difference between a usable and maintainable set of VIs, and VIs that confuse users and make future modifications needlessly difficult.

This chapter focuses on documentation that is specific to LabVIEW-based development. It does not address general documentation issues that apply to all software products.

Create several documents for software you develop. The two main categories for this documentation are as follows:

- Design-related documentation—Requirements, specifications, detailed design plans, test plans, and change history documents are examples of the kinds of design-related documents you might need to produce.
- User documentation—User documentation explains how to use the software.

The style of each of these documents is different. Design-related documentation generally is written for an audience with extensive knowledge of the tools that you are documenting, and that they are using. User documentation is written for an audience with a lesser degree of understanding and experience with the software.

The size and style of each document can vary according to the type of project. For simple tools that are used only in-house, you might not need to do much of either. If you plan to sell a product, you must allow a significant amount of time to develop detailed user-oriented documentation that describes the product. For products that must go through a quality certification process, such as a review by the U.S. Food and Drug Administration, you must ensure that the design-related documentation is as detailed as required.

Design and Development Documentation

The format and detail level of the documentation you develop for requirements, specifications, and other design-related documentation is determined by the quality goals of the project. If you are developing to meet a quality standard such as ISO 9000, the format and detail level of these documents are different from the format and detail level of an in-house project.

Refer to Appendix A, *References*, for a list of documents that include more information about the types of documents to prepare as part of the development process.

LabVIEW includes features that simplify the process of creating documentation for the VIs you design:

- **History** window—Use the **History** window to record changes to a VI as you make them.
- **Print** dialog box—Use the **Print** dialog box to create printouts of the front panel, block diagram, connector pane, and description of a VI. You can also use it to print the names and descriptions of controls and indicators for the VI and the names and paths of any subVIs. You can print this information, generate Web pages, create online help source files, or create word-processor documents.

Developing User Documentation

End users of VIs fall into two classes: end users of top-level VIs and end users of subVIs. Each of these users have different documentation needs. This section addresses techniques for creating documentation that helps both of these classes of users. The format of user documentation depends on the type of product you create.

Documentation for a Library of VIs

If the software you are creating is a library of VIs for use by other developers, such as an instrument driver or add-on package, create documents with a format similar to the *LabVIEW Help*. Because the audience is other developers, you can assume they have a working knowledge of LabVIEW. The documentation might consist of an overview of the contents of the package, examples of how to use the subVIs, and a detailed description of each subVI.

For each subVI, you might want to include the VI name and description, a picture of the connector pane, and the description and a picture of the data type for each of the controls and indicators on the connector pane.

You can generate much of this documentation easily if you use the **Documentation** page of the **VI Properties** dialog box for VIs and controls as described in the *VI and Control Descriptions* section later in this chapter. You can use **File»Print** to create a printout of a VI in a format almost identical to the format used in the VI and function reference information in *LabVIEW Help*. With **File»Print** you also can save the documentation to a file and create documentation for multiple VIs at once.

Documentation for an Application

If you are developing an application for users who are not familiar with LabVIEW, the documentation requires more introductory material. The documentation covers basic features such as installation and system requirements. It provides an overview of how the package works. If the package uses I/O, describe the necessary hardware and any configuration that must be done before the user starts the application.

For each front panel the user interacts with, provide a picture of the front panel and a description of the major controls and indicators. Organize the front panel descriptions in a top-down fashion, with the first front panels the user sees documented first. As described in the previous section, you can use the **Print** dialog box to create this documentation.

Creating Help Files

You can create online help or reference documents if you have the right development tools. Online help documents are based on formatted text documents. You can create these documents using a word-processing program, such as Microsoft Word, or using other help compiling tools. You can also create online help documents in HTML with an HTML help compiler. Special help features such as links and hotspots are created as hidden text.

You can use the **Print** dialog box to help you create the source material for the help documents.

Once you have created source documents, use a help compiler to create a help document. If you need help files on multiple platforms, you must use the help compiler for the specific platform on which the help files are used. Once you have created and compiled the help files, you can add them to the

Help menu of LabVIEW or your own custom application by placing them in the `help` directory. You also can link to them directly from a VI using one of two ways.

You can link to the **Help** menu using the **VI Properties»Documentation** dialog box. Refer to the **VI Properites** *LabVIEW Help* for more information about linking to the **Help** menu from VIs. You also can use the Help functions on the **Functions»Application Control»Help** palette to link to topics in specific help files programmatically.

VI and Control Descriptions

You can integrate information for the user in each VI you create by using the VI description feature, by placing instructions on the front panel, and by including descriptions for each control and indicator.

VI Description

The VI description in the **File»VI Properties** dialog box is often the only source of information about a VI accessible to a user. The **Context Help** window displays the VI description when the user moves the mouse cursor over the VI icon, either the connector pane or the icon used as a subVI on a block diagram.

Include the following important items in a VI description:

- An overview of the VI
- Instructions for use
- Descriptions of inputs and outputs

Self-Documenting Front Panels

One way of providing important instructions is to place a block of text prominently on the front panel. A concise list of important steps is valuable. You might even include a suggestion such as, "Select **File»VI Properties** for instructions" or "Select **Help»Show Help**." For long instructions, you can use a scrolling string control instead of a free label. Be sure to right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu to save the text when you finish entering the text.

If a text block requires too much space on the front panel, you can include a highly visible **Help** button on the front panel instead. Include the instruction string on the front panel that appears when the user clicks the

Help button. Use the **Window Appearance** page in the **VI Properties** dialog box to configure this help panel as either a dialog box that requires the user to click an **OK** button to close it and continue or as a window the user can move anywhere and close anytime.

Alternatively, you can use a **Help** button to open an entry in an online help file. You can use the Help functions on the **Functions»Application Control»Help** palette to open the **Context Help** window or to open a help file and link to a specific topic.

Control and Indicator Descriptions

Include a description for every control and indicator. You can enter this with the **Description and Tip** shortcut menu item. The **Context Help** window displays an object description when the user moves the mouse cursor over the object.

When confronted with a new VI, a user has no alternative but to guess the function of each control and indicator unless you include a description. Always remember to enter a description as soon as you create the object. Then, if you copy the object to another VI, the description is copied also. Also be sure to tell users about this behavior.

Every control and indicator needs a description that includes the following information:

- Functionality
- Data type
- Valid range (for inputs)
- Default value (for inputs)
- Behavior for special values (0, empty array, empty string, and so on)
- Additional information, such as if the user must set this value always, often, or rarely

Alternatively, you can list the default value in parentheses as part of the control or indicator name. For controls and indicators on the VI connector pane, mark the inputs and outputs by right-clicking the connector pane and selecting **This Connection is**»**Required**, **Recommended**, or **Optional** from the shortcut menu.

6

LabVIEW Style Guide

This chapter describes recommended practices for good programming technique and style. Remember that these are only *recommendations*, not laws or strict rules. Several experienced LabVIEW programmers have contributed to this guide.

As mentioned in Chapter 2, *Incorporating Quality into the Development Process*, inconsistent style causes problems when multiple developers are working on the same project. The resulting VIs can confuse users and be difficult to maintain. To avoid these problems, establish a set of style guidelines for VI development. You can establish an initial set at the beginning of the project and add additional guidelines as the project progresses. The most important thing is often not the specific style you use, but the consistency of that style.

A style checklist is included at the end of this chapter to help you maintain quality as you develop VIs. To save time, review the list before and during development. To create a successful VI, you must consider who will be using it and for what reasons. Consider your audience:

- Users need a clear front panel
- Developers need an easy to read block diagram
- Everybody needs good documentation

Organization

Organize the VIs in the file system to reflect the hierarchical nature of the software. Make the top-level VIs directly accessible. Place subVIs in subdirectories and group them to reflect any modular components you have designed, such as instrument drivers, configuration utilities, and file I/O drivers.

Create a directory for all the VIs for one application and give it a meaningful name, as shown in Figure 6-1. Save the main VIs in this directory and the subVIs in a subdirectory. If the subVIs have subVIs, continue the directory hierarchy downward.

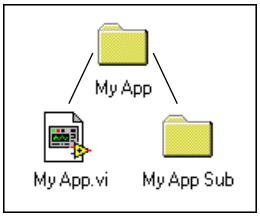

Figure 6-1. Directory Hierarchy

When naming VIs, VI libraries, and directories, avoid using characters that are not accepted by all file systems, such as slash (/), backslash (\), colon (:), tilde (~), and so on. Most operating systems accept long descriptive names for files, up to 31 characters on a Macintosh and 255 characters on other platforms.

Select **Tools»Options** to make sure the VI Search Path contains `<topvi>\*` and `<foundvi>\*`. The `*` causes all subdirectories to be searched. In Figure 6-1, `MyApp.vi` is the top-level VI. This means that the application searches for subVIs in the directory `MyApp`. Once a subVI is found in a directory, the application looks in that directory for subsequent subVIs.

Avoid creating files with the same name anywhere within the hierarchy. Only one VI of a given name can be in memory at a time. If you have a VI with a specific name in memory and you attempt to load another VI that references a subVI of the same name, the VI links to the VI in memory. If you make backup copies of files, be sure to save them into a directory outside the normal search hierarchy so that LabVIEW does not mistakenly load them into memory when you open development VIs.

Refer to the *Saving VIs* section of Chapter 7, *Creating VIs and SubVIs*, in the *LabVIEW User Manual* for more information about saving VIs individually and in VI libraries.

Front Panel Style

Remember that a user's first contact with your work is the front panel. Front panels should be well organized and easy to use.

Fonts and Text Styles

Do not be tempted to use all the fonts and styles available. Limit the VI to the three standard fonts—application, system, and dialog—unless you have a specific reason to use a different font. For example, monospace fonts, fonts that are proportionally spaced, are useful for string controls and indicators where the number of characters is critical. To set the default font, select it from the **Text Settings** pull-down menu in the toolbar without any text or objects selected. You can select all the labels you need to change and set the font in all of them at once using the **Text Settings** pull-down menu.

The actual font used for the three standard fonts varies depending on the platform. For example, when working under Windows, preferences and video driver settings will affect the size of the fonts. Text might appear larger or smaller on different systems, depending on these factors. To compensate for this, allow extra space for larger fonts and enable the **Size to Text** option on the shortcut menu. Use carriage returns to make multiline text instead of resizing the text frame.

You can prevent labels from overlapping because of font changes on multiple platforms by allowing extra space between controls. For example, make space to the right of text that has been left-justified. Fonts are the least portable aspect of the front panel, so always test them on all of the target platforms.

Color

Like fonts, it is easy to get carried away with color. The particular danger of color is that it distracts the operator from important information. For instance, a yellow, green, and bright orange background make it difficult to see a red danger light. Another problem is that other platforms might not have as many colors available. Also, some users have black-and-white monitors that cannot display certain color combinations well. For example, black-and-white monitors display black letters on a red background as all black. Use a minimal number of colors, emphasizing black, white, and gray. The following are some simple guidelines for using color:

- Never use color as the sole indicator of device state. People with some degree of color-blindness might not detect the change. Also, multiplot graphs and charts can lose meaning when displayed in black and white.

- Use line styles in addition to color.
- Use light gray, white, or pastel colors for backgrounds.
- Select bright, highlighting colors only when the item is important, such as an error notification.
- Always check the VI on other platforms and on a black-and-white monitor.
- Be consistent in use of color.

Graphics and Custom Controls

You can enhance the functionality of the front panel with imported graphics. You can import bitmaps, Macintosh PICTs, Windows Enhanced Metafiles, and text objects for use as backgrounds or in pict rings and custom controls, as shown in Figure 6-2.

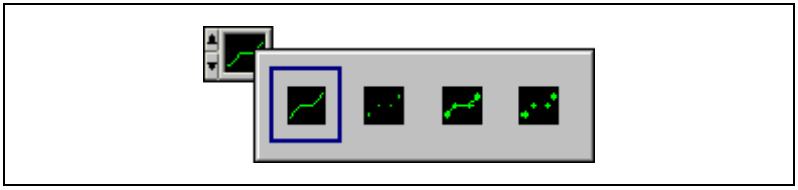

Figure 6-2. Example of Imported Graphics Used in a Pict Ring

Use a pict ring when a function or mode is conveniently described by a picture.

Check how the imported pictures look when the VI is loaded on another platform. For example, a Macintosh PICT file that has an irregular shape might convert to a rectangular bitmap with a white background on Windows or UNIX.

One disadvantage of imported graphics is that they slow down screen updates. Make sure indicators and controls are not placed on top of a graphic object. That way, the object does not have to be redrawn each time the indicator is updated.

Tip If you must use a large background picture with controls on top of it, try breaking it into several smaller objects and import them separately. Large graphics usually take longer to draw than small ones. For instance, import several pictures of valves and pipes individually instead of importing one large picture.

Use a custom Boolean control that is transparent in one state, and is visible in another to detect mouse clicks in specified regions of the screen.

Layout

Consider the arrangement of controls on front panels. Keep front panels simple to avoid confusing the user, and use menus to help reduce clutter. For top-level VIs that users see, place the most important controls in the most prominent positions. Use the **Align Objects** and the **Distribute Objects** pull-down menus to create a uniform layout. Use **Edit»Reorder Controls in Panel** to arrange controls in a logical sequence.

Do not overlap controls with other controls or with their own label, digital display, or other parts unless you are trying to achieve a special effect. Overlapped controls are much slower to draw and might flash.

Use decorations such as raised rectangles or recessed frames to visually group objects with related functions, as shown in the illustration below. Use clusters to group related data. However, do not use clusters for aesthetic purposes only. It makes connections to the VI more difficult to understand. Avoid importing graphic objects that are inanimate copies of real controls. For instance, do not use a copy of a cluster border to group controls that are not actually in a cluster.

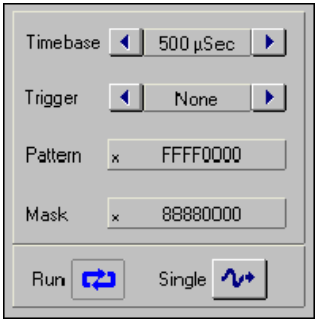

Figure 6-3. Example of Using Decorations to Visually Group Objects Together

For subVI front panels the user does not see, you can place the objects so they correspond to the connector pattern. Generally, place inputs on the left and outputs on the right.

Sizing and Positioning

Front panels should fit on a monitor that is the standard resolution for most intended users. Make the window as small as possible without crowding controls or sacrificing a good layout. If the VIs are intended for in-house use and everyone is using high-resolution display settings, design large

front panels. If you are doing commercial development, keep in mind that some displays might offer a limited resolution, especially LCD displays and touchscreens.

Front panels should open in the upper-left corner of the screen for the convenience of users with small screens. Place sets of VIs that are often opened together so the user can see at least a small part of each. Place front panels that open automatically in the center of the screen by selecting **VI Properties»Windows Appearance»Auto-Center**. Centering the front panels makes the the VI eaiser to read for users on monitors of various sizes.

Labels

The **Context Help** window displays labels as part of the connector. If the default value of a control is valid, add it to the name in parentheses. Include the units of the value, where applicable. The **Required**, **Recommended**, **Optional** setting for connector pane terminals affects the appearance of the inputs and outputs in the **Context Help** window.

The name of a control or indicator should describe its function. For example, for a ring or labeled slide with options for volts, ohms, or amperes, a name like “Select units for display” is better than “V/O/A” and is certainly an improvement over the generic “Mode.” If the control is going to be visible to the user, use captions to display a long description and add a short label to prevent using valuable space on the block diagram. Use property nodes to change captions programatically. For Boolean controls, the name should give an indication of which state corresponds to which function, while still indicating the default state. Free labels next to a Boolean can help clarify the meaning of each position on a switch, as shown in Figure 6-4.

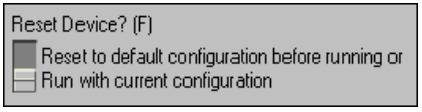

Figure 6-4. Free Labels on a Boolean Control

Enums versus Rings

Because the names are a part of the data type, you cannot change the names in an enumeration programmatically at run time. Also, you cannot compare two enumumerations of different types. If you wire an enumeration control to something that expects a standard numeric, you will see a coercion dot because the type is being converted.

Enumeration controls are useful for making code easier to read. Ring controls are useful for front panels the user interacts with, where you want to programmatically change the strings.

Default Values and Ranges

Expect the user to supply invalid values to every control. You can check for invalid values in the block diagram or right-click the control and select **Data Range** to set the control item to coerce values into the desired range: **Minimum**, **Maximum**, and **Increment**.

Set controls with reasonable default values. A VI should not fail when run with default values. Remember to show the default in parentheses in the control label. Do not set default values of indicators like graphs, arrays, and strings without a good reason because that wastes disk space when saving the VI.

Use default values intelligently. In the case of high-level file VIs such as the Write Characters to File VI, the default is an empty path that forces the VI to display a **File Selection** dialog box. This can save the use of a Boolean switch in many cases.

Other difficult situations must be dealt with programmatically. Many GPIB instruments limit the permissible settings of one control based on the settings of another. For example, a voltmeter might permit a range setting of 2,000 V for DC but only 1,000 V for AC. If the affected controls like Range and Mode reside in the same VI, place the interlock logic there. If one or more of the controls are not readily available, you can request the present settings from the instrument to ensure you do not try to set an invalid combination.

Property Nodes

Use property nodes to give the user more feedback on the front panel and to make the VI easier to use. The following are examples of using property nodes to enhance ease of use:

- Set the text focus to the main, most commonly used control.
- Disable or hide controls that are not currently relevant or valid.
- Guide the user through steps by highlighting controls.
- Change window colors to bring attention to error conditions.

Key Navigation

Some users prefer to use the keyboard instead of a mouse. In some environments, such as a manufacturing plant, only a keyboard is available. Even if a mouse is used, keyboard shortcuts, such as using the <Enter> key to select the default action of a dialog box, add convenience. For these reasons, consider including keyboard shortcuts for the VIs.

Select **Edit»Reorder Controls in Panel** to see the order of the front panel controls. In general, set the order to read left to right and top to bottom.

Pay attention to the key navigation options for buttons on the front panel. You can set key navigation options by right-clicking any control and selecting **Advanced»Key Navigation** from the shortcut menu. Set the <Enter> key to be the keyboard shortcut to the front panel default control. However, if you have a multiline string control on the front panel, you might not want to use the <Enter> key as a shortcut. Refer to the *Controlling Button Behavior with Key Navigation* section of the *LabVIEW User Manual* for more information about using key navigation options.

If the front panel has a **Cancel** button, assign a shortcut to the <Esc> key. You also can use function keys as navigation buttons to move from screen to screen. If you do this, be sure to use the shortcuts consistently. For controls that are offscreen, use the **Key Navigation** dialog box to skip over the controls when tabbing.

Also, you might consider using the **Key Focus** property to set the focus programmatically to a specific control when the front panel opens.

Dialog Boxes

Dialog boxes can be used effectively in LabVIEW applications as a way to group controls that are too numerous to place on one front panel. Consider using the tab control to group controls effectively and reduce clutter.

Many modern programs use dialog boxes to announce messages to the user, but quite often this is overused. Consider using a status text window to display less serious warnings. Refer to the *Designing Dialog Boxes* section of Chapter 4, *Building the Front Panel*, of the *LabVIEW User Manual* for more information about creating dialog boxes.

Block Diagram Style

Style is just as important on the block diagram of the VI as the front panel. Users may not see it, but other developers will. A well-planned, consistent block diagram is easier to understand and modify.

Good Wiring Techniques

The block diagram is the primary way for others to understand how a VI works, therefore it is often worth the effort to follow a few simple steps to make the block diagrams more organized and easier to read. The Align and Distribute feature in LabVIEW can make it easy to quickly arrange objects on the block diagram to make it easier to see and understand groupings of objects. Placing objects using symmetry and straight lines can make the block diagram easier to read.

Some other good wiring tips are:

- Avoid placing any wires under any block diagram objects such as subVIs or structures.
- Add as few bends in the wires as possible while trying to keep the wires short. Avoid creating wires with long complicated paths that can be confusing.
- Delete any extraneous wires to keep block diagram clean.
- Avoid the use of local variables when it is possible to pass the data by wire. Every local variable that reads the data makes a copy of it.
- Try not to pass wires though structures if the data in the wire itself is not used within the structure.
- Evenly space parallel wires in straight lines and around corners.

Memory and Speed Optimization

There are many things you can do to optimize usage and execution time of a LabVIEW VI. Generally an advanced topic, optimization quickly becomes a concern when a program has large arrays and/or critical timing problems. Refer to the *LabVIEW Performance* application note for more information about optimizing LabVIEW VIs. Even though optimization can be a very involved topic, you can take the following basic actions to optimize an application:

- If speed is not necessary to a While Loop, add a Wait function to avoid the problem of slowing down other tasks. Generally slowing down other tasks is only an issue with loops that are not as active between iterations, such as the user interface loops, because LabVIEW runs the

loop as quickly as possible and does not give many processor resources to any other tasks. The slowing of tasks outside of the loop results in the computer seeming sluggish even though it is running a simple loop. Adding a slight delay between iterations with the Wait (ms) function can dramatically help the computer run outside tasks normally without affecting the operation of the loop. Typically a delay of 50 to 100 MS is sufficient, but other factors in the application might affect the delay. The delay does not have to have any data dependencies, as shown in Figure 6-5.

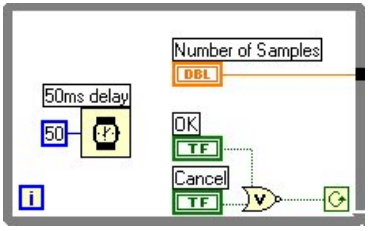

Figure 6-5. While Loop with 50 Second Delay

- If possible, do not build arrays using the Build Array function within a loop because the function makes repetitive calls to the memory manager. A more efficient method of building an array is to use auto-indexing or pre-size the array and use the Replace Array Element function to place values in it. Similar issues exist when dealing with strings because in memory, LabVIEW handles strings as arrays of characters.
- Use global and local variables as sparingly as possible. You can use both global and local variables to write VIs very efficiently. However, if you are misuse or abuse global and local variables, the memory usage of the VI increases and the performance is affected.
- Choosing the proper data type for the data to be handled can be very important in controlling the memory usage of the application. For example, imagine you have an extended-precision floating-point array of 100,000 values, but the actual values that are stored in the array are only between the values 0 to 10. You could have used an array of single-precision floating-point values for the following difference in memory usage in Windows:

Array Data Type	Memory Used
Array of 100,000 EXT values	1 MB

Array Data Type	Memory Used
Array of 100,000 SGL values	400 KB
Memory saved	600 KB

- Avoiding coercion dots also can help you reduce unnecessary memory usage and speed. Coercion dots indicate that a conversion is taking place from one data type to another, which means that a copy of the data must be made. The effects of this become much more dramatic when there is coercion on large arrays of data.
- Consider indicator overhead when designing the VI if performance is a concern for the application. Frequently updating front panel indicators with new data can effect the performance of the VI especially if you are displaying large amounts of data in graphs or charts. To optimize the performance of the VI, only display the necessary information on the front panel and only send data to indicators if the data is different from what is already displayed.

Sizing and Positioning

The size of the block diagram window can affect how readable your LabVIEW code is to others. In general, try to make the block diagram window no larger than the screen size. To ensure that the block diagram is readable, keep it smaller than 800×600 pixels. While it is good to be aware of the size of the block diagram window, it is also very important to ensure that the LabVIEW code that is displayed in it is not too large. Code that is much larger than the window displaying it can force others to scroll the window, sometimes making the code harder to read. Code that requires the user to scroll only horizontally or vertically is acceptable as long as the user does not have to scroll an unreasonable amount to view the entire code.

Left-to-Right Layouts

LabVIEW was designed to use a left-to-right and sometimes top-to-bottom layout. Block diagrams should follow this convention. While the positions of program elements do not determine execution order, avoid wiring from right to left. Only wires and structures determine execution order.

Block Diagram Comments

Developers who maintain and modify VIs need good documentation. Without it, modifying the code is more time-consuming and error-prone than necessary.

- Use comments on the block diagrams to explain what the code is doing. LabVIEW code is not self-documenting even though it is graphical. Free labels with a colored background work well for block diagram comments.
- Omit labels on function and subVI calls because they tend to be large and unwieldy. Someone looking at the blcok diagram can easily find out the name of a function or subVI call by using the **Context Help** window.
- Use free labels on wires to identify their use. This is particularly useful for wires coming from shift registers.
- Use labels on structures to specify the main functionality of that structure.
- Use labels on constants to specify the nature of the constant.
- Use labels on Call Library Nodes to specify what function the node is calling.
- Use the description of Code Interface Nodes (CINs) to record the following information:
  - Source code title and filename
  - Platform and O/S
  - Compiler version
  - Where to find the source code
  - What the code does
  - List of other files required by the CIN
  - Other critical information required to maintain the CIN
- Use comments to document algorithms that you use on the block diagrams. If you use an algorithm from a book or other reference, provide the reference information.

Icon and Connector Style

Using good style techniques when creating the icons and connectors for VIs can greatly benefit users of those VIs.

For examples of good icons and connector designs, the LabVIEW Queue VIs, available on **Functions»Advanced»Synchronization»Queue** palette.

Icon

The icon represents the VI on a palette and the block diagram. Use the follwoing suggestions when creating icons:

- Create a meaningful icon for every VI. The LabVIEW libraries are full of well-designed icons that try to show the functionality of the underlying program; use them as prototypes where applicable. When you don't have a picture, text is acceptable.

Tip 8pt Small Fonts is a good size and font choice for text icons

- Create a unified icon style for related VIs. This helps users visually understand what subVI calls are associated with the top-level VI.
- Do not use plays on words when making the icon. Plays on words usually do not work for users who speak a different language. For example, don't represent the data logging VI by a picture of a piece of a tree branch or a lumberjack. This sort of picture does not translate well.
- Always create a black and white icon for printing purposes. Probably not every user has access to a nice color printer.
- Always create standard size (32x32) icons. VIs with smaller icons can be awkward to select and wire. They also tend to look strange when wired.

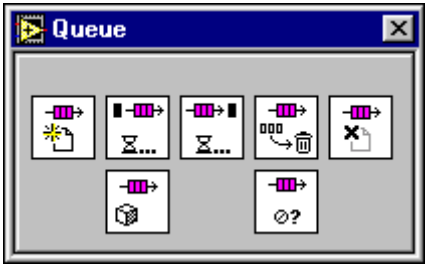

Figure 6-6. Example: Queue VIs

The Queue VIs use a picture (a graphical representation of a queue) as a common element in their icons to unify the icon designs. Notice that all of the icons use pictures only (and none that are plays on words), making these icons suitable for speakers of any language.

Refer to the *Creating an Icon* section of the *LabVIEW User Manual* for more information about using icons.

Connector

The connector has terminals that connect to the controls and indicators of the subVIs of the top-level VI. Use the following suggestions when creating connectors:

- Always select a connector pane pattern with extra terminals. Unforeseen additions to the VI may require more terminals. Changing patterns requires relinking to the subVI in all calling VIs and may result in wiring incompatibilities.
- Choose the same connector pane pattern for related VIs. Wire related inputs and outputs on opposite terminals .
- Wire inputs on the left and outputs on the right to follow the standard left-to-right data flow.
- Use the **Required**, **Recommended**, **Optional** setting for every terminal.
- Choose only connector pane patterns with 16 or fewer terminals. While patterns with more terminals might seem useful, they are very difficult to wire. If you need to pass more data, use clusters.
- Strive for similar arrangements between panel and connector pane layouts.

Style Checklist

Use the following checklist to help you maintain consistent style and quality. You might want to copy this checklist to use on all LabVIEW projects.

VI Checklist

- ❑ Organize VIs in a hierarchical directory with easily accessible top-level VIs and subVIs in subdirectories.
- ❑ Avoid putting too many VIs in one library because large LLBs take longer to save.
- ❑ With LLBs, use **Tools»Edit VI Library** to mark top-level VIs.
- ❑ If the VIs will be used as subVIs, create a `.mnu` file or edit the menu that is part of the LLB. Be sure to arrange the palettes, name the menus, and hide dependent subVIs.

- ❑ Give VI meaningful names without special characters, such as backslash (\), slash (/), colon (:), and tilde (~).
- ❑ Use standard extensions so Windows and UNIX can distinguish files (`.vi`, `.ctl`).
- ❑ Capitalize first letters of VI names.
- ❑ Distinguish example VIs, top-level VIs, subVIs, controls, and global variables by saving them in subdirectories, separate libraries in the same directory, or by giving them descriptive names such as `MainX.vi`, `Example of X.vi`, `Global X.vi`, and `TypeDef X.ctl`.
- ❑ Write a VI description. Proofread it. Check the **Context Help** window.
- ❑ Include your name and/or company and the date in the **VI Description** on the **Documentation** page of the **VI Properties** dialog box.
- ❑ When you modify a VI, use the **History** window to document the changes.
- ❑ Create a meaningful black-and-white icon in addition to a color icon.
- ❑ Choose a connector pane pattern to leave extra terminals for later development. Use consistent layout across related VIs.
- ❑ Avoid using connector panes with more than 16 terminals.
- ❑ Consider VI and window options carefully.
- ❑ Hiding menu bars and using dialog box style makes **Context Help** and VI descriptions inaccessible.
- ❑ Hiding **Abort** and debugging buttons increases performance slightly.
- ❑ Set print options to print attractive output in the most useful format.
- ❑ Make test VIs that check error conditions, invalid values, and **Cancel** buttons.
- ❑ Save test VIs in a separate directory so you can reuse them.
- ❑ Load and test VIs on multiple platforms, making sure labels fit and window size and position are correct.

Front Panel Checklist

- ❑ Give controls meaningful names. Use consistent capitalization.
- ❑ Make name label backgrounds transparent.
- ❑ Check for consistent placement of control names.
- ❑ Use standard, consistent fonts throughout all front panels.
- ❑ Use **Size to Text** for all text for portability and add carriage returns if necessary.
- ❑ Use **Required**, **Recommended**, and **Optional** settings on the connector pane.
- ❑ Put default values in parentheses after input names.
- ❑ Include unit information in names if applicable, for example, **Time Limit (10 Seconds)**.
- ❑ Write descriptions for controls, including array, cluster, and refnum elements. Remember that you might need to change the description if you copy the control.
- ❑ Arrange controls logically. For top-level VIs, put the most important controls in the most prominent positions. For subVIs, put inputs on the left and outputs on the right and follow connector pane terminals.
- ❑ Arrange controls attractively, using the **Align Objects** and the **Distribute Objects** pull-down menus.
- ❑ Do not overlap controls.
- ❑ Use color logically and sparingly, if at all.
- ❑ Use error in and error out clusters where appropriate.
- ❑ Consider other common thread controls, such as taskID, refnum, and name.
- ❑ Provide a **Stop** button if necessary. Do not use the **Abort** button to stop a VI. Hide the **Abort** button.
- ❑ Use ring controls and enumerated controls where appropriate. If you are using a Boolean controls for two options, consider using an enumerated control instead to allow for future expansion of options.

- ❑ Use custom controls or typedefs for common controls, especially for rings and enums.
- ❑ In system controls, label controls with the same name as the VI, for example, `Alarm Boolean.ctl` has the default name Alarm Boolean.

Block Diagram Checklist

- ❑ Avoid creating extremely large block diagrams. Limit the scrolling necessary to see in the entire block diagram to one direction.
- ❑ Label controls, important functions, constants, property nodes, local variables, global variables, and structures.
- ❑ Add comments. Use object labels instead of free labels where applicable and scrollable string constants for long comments.
- ❑ Place labels below objects when possible and right-justify text if label is placed to the left of an object.
- ❑ Use standard, consistent font conventions throughout.
- ❑ Use **Size to Text** for all text and add carriage returns if necessary.
- ❑ Reduce white space in smaller block diagrams but allow at least 3 or 4 pixels between objects.
- ❑ Flow data from left to right. Wires enter from the left and exit to the right, not the top or the bottom.
- ❑ Align and distribute functions, terminals, and constants.
- ❑ Label long wires with small labels with white backgrounds.
- ❑ Do not wire behind objects.
- ❑ Make good use of reusable, testable subVIs.
- ❑ Make sure the program can deal with error conditions and invalid values.
- ❑ Show name of source code or include source code for any CINs.
- ❑ Save with the most important or the first frame of structures showing.
- ❑ Review for efficiency, especially data copying, and accuracy, especially parts without data dependency.

A

References

This appendix provides a list of references for further information about software engineering concepts.

LabVIEW Technical Resource. Edited by Lynda P. Gruggett, LTR Publishing, phone (214) 706-0587, fax (214) 706-0506. `http://www.LTRPub.com`. A quarterly newsletter and disk of VIs that features technical articles about all aspects of LabVIEW.

Rapid Development: Taming Wild Software Schedules. Steve C. McConnell, Microsoft Press. Explanation of software engineering practices with many examples and practical suggestions.

Microsoft Secrets. Michael A. Cusumano and Richard W. Selby, Free Press. In-depth examination of the programming practices Microsoft uses. Contains interesting discussions of what Microsoft has done right and what it has done wrong. Includes a good discussion of team organization, scheduling, and milestones.

Dynamics of Software Development. Jim McCarthy, Microsoft Press. Another look at what has worked and what has not for developers at Microsoft. This book is written by a developer from Microsoft and contains numerous real-world stories that help bring problems and solutions into focus.

Software Engineering—A Practitioner's Approach. Roger S. Pressman, McGraw-Hill Inc. A detailed survey of software engineering techniques with descriptions of estimation techniques, testing approaches, and quality control techniques.

Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products. Daniel P. Freedman and Gerald M. Weinberg, Dorset House Publishing Co., Inc. A discussion of how to conduct design and code reviews with many examples of things to look for and the best practices to follow during a review.

ISO 9000.3: A Tool for Software Product and Process Improvement. Raymond Kehoe and Alka Jarvis, Springer-Verlag New York, Inc.

Describes what is expected by ISO 9001 in conjunction with ISO 9000.3 and provides templates for documentation.

Software Engineering Economics. Barry W. Boehm, Prentice-Hall. Description of the wideband delphi and COCOMO estimation techniques.

Software Engineering. Edited by Merlin Dorfman and Richard Thayer, IEEE Computer Science Press. Collection of articles on a variety of software engineering topics, including a discussion of the spiral life cycle model by Barry W. Boehm.

B

Technical Support Resources

Web Support

National Instruments Web support is your first stop for help in solving installation, configuration, and application problems and questions. Online problem-solving and diagnostic resources include frequently asked questions, knowledge bases, product-specific troubleshooting wizards, manuals, drivers, software updates, and more. Web support is available through the Technical Support section of `www.ni.com`

NI Developer Zone

The NI Developer Zone at `zone.ni.com` is the essential resource for building measurement and automation systems. At the NI Developer Zone, you can easily access the latest example programs, system configurators, tutorials, technical news, as well as a community of developers ready to share their own techniques.

Customer Education

National Instruments provides a number of alternatives to satisfy your training needs, from self-paced tutorials, videos, and interactive CDs to instructor-led hands-on courses at locations around the world. Visit the Customer Education section of `www.ni.com` for online course schedules, syllabi, training centers, and class registration.

System Integration

If you have time constraints, limited in-house technical resources, or other dilemmas, you may prefer to employ consulting or system integration services. You can rely on the expertise available through our worldwide network of Alliance Program members. To find out more about our Alliance system integration solutions, visit the System Integration section of `www.ni.com`

Worldwide Support

National Instruments has offices located around the world to help address your support needs. You can access our branch office Web sites from the Worldwide Offices section of `www.ni.com`. Branch office web sites provide up-to-date contact information, support phone numbers, e-mail addresses, and current events.

If you have searched the technical support resources on our Web site and still cannot find the answers you need, contact your local office or National Instruments corporate. Phone numbers for our worldwide offices are listed at the front of this manual.

Glossary

B

black box testing	A form of testing where a module is tested without knowing how the module is implemented. The module is treated as if it were a black box that you cannot look inside. Instead, you generate tests to verify the module behaves the way it is supposed to according to the requirements specification.

C

Capability Maturity Model (CMM)	A model for judging the maturity of the processes of an organization and for identifying the key practices that are required to increase the maturity of these processes. The Software CMM (SW-CMM) has become a de facto standard for assessing and improving software processes. Through the SW-CMM, the Software Engineering Institute and software development community have put in place an effective means for modeling, defining, and measuring the maturity of the processes software professionals use.
COCOMO Estimation	COnstructive COst MOdel. A formula-based estimation method for converting software size estimates to estimated development time.
code and fix model	A lifecycle model that involves developing code with little or no planning, and fixing problems as they arise.
configuration management	A mechanism for controlling changes to source code, documents, and other materials that make up a product. During software development, Source Code Control is a form of configuration management: Changes occur only through the Source Code Control mechanism. It is also common to implement release configuration management to ensure a particular release of software can be rebuilt, if necessary. This implies archival development of tools, source code, and so on.

F

Function-Point Estimation	A formula-based estimation method applied to a category breakdown of project requirements.

I

integration testing — Integration testing assures that individual components work together correctly. Such testing may uncover, for example, a misunderstanding of the interface between modules.

L

lifecycle model — A model for software development, including steps to follow from the initial concept through the release, maintenance, and upgrading of the software.

S

Software Engineering Institute (SEI) — A federally funded research and development center chartered to study software engineering technology. The SEI is located at Carnegie Mellon University and is sponsored by the Defense Advanced Research Projects Agency.

source lines of code — The measure of the number of lines of code that make up a project. It is used in some organizations to measure the complexity and cost of a project. How the lines are counted depends on the organization. For example, some organizations do not count blank lines and comment lines. Some count C lines, and some count only the final assembly language lines.

spiral model — A lifecycle model that emphasizes risk management through a series of iterations in which risks are identified, evaluated, handled.

system testing — System testing begins after integration testing is complete. System testing assures that all the individual components function correctly together and constitute a product that meets the intended requirements. This stage often uncovers performance, resource usage, and other problems.

U

unit testing — Testing only a single component of a system, in isolation from the rest of the system. Unit testing occurs before the module is incorporated into the rest of the system.

W

waterfall model	A lifecycle model that consists of several non-overlapping stages, beginning with the software concept and continuing through testing and maintenance.
white box testing	Unlike black box testing, white box testing creates tests that take into account the particular implementation of the module. For example, white box testing is used to verify all the paths of execution of the module have been exercised.
wideband delphi estimation	Wideband delphi is a technique used by a group to estimate the amount of effort a particular project will take.

Index

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W