Managing Software Engineering in LabVIEW™ Course Manual

Course Software Version 2009
October 2009 Edition
Part Number 372907A-01

Worldwide Technical Support and Product Information
ni.com

National Instruments Corporate Headquarters
11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

Contents

Student Guide

A. NI Certification v
B. Course Description vi
C. What You Need to Get Started vi
D. Installing the Course Software vii
E. Course Goals vii
F. Course Conventions viii

Lesson 1
Introduction

A. Software Engineering Process 1-2
B. Roles of a LabVIEW Architect 1-10
C. Configuration Management 1-13

Lesson 2
Requirements Gathering

A. Introduction 2-2
B. Project Management 2-3
C. Gathering and Managing Requirements 2-5
D. Time Estimation 2-15
E. Style and Coding Standards 2-25
F. Developing a Project Plan 2-30

Lesson 3
Design

A. Introduction 3-2
B. Creating a Software Model 3-3
C. Designing a Software Architecture 3-11
D. Creating a Prototype 3-17
E. Developing a Design Document 3-20

Lesson 4
Development

A. Introduction 4-2
B. Automating Development Tasks 4-2
C. Code Reviews 4-9

Lesson 5
Validation

A. Introduction....5-2
B. Static Code Analysis....5-6
C. Dynamic Code Analysis....5-10
D. Functional Validation of Code....5-15

Lesson 6
Deployment

A. Introduction....6-2
B. Methods of Deployment....6-3
C. Advanced Application Options....6-9
D. Advanced Installer Options....6-19
E. Shared Library Development....6-29
F. Source Code Distribution....6-35
G. Additional Build Specifications....6-38
H. Comparison of Deployment Processes....6-40

Appendix A
Summary of Software Engineering Process Deliverables

A. Requirements Gathering....A-2
B. Design....A-2
C. Development....A-3
D. Validation....A-4
E. Deployment....A-4

Appendix B
Additional Information and Resources

Course Evaluation

3

Design

This lesson describes tools and practices you can use to aid in designing a LabVIEW project. You learn how to develop a prototype for the user interface, how to develop a graphical model of the application, and how to develop a design document that will be used throughout project development.

Topics

A. Introduction

B. Creating a Software Model

C. Designing a Software Architecture

D. Creating a Prototype

E. Developing a Design Document

A. Introduction

Through design, the LabVIEW architect makes sure the software framework of a system meets the specified requirements. The design defines the system components and the interaction of those components. The LabVIEW architect is also responsible for determining the external interfaces and tools to use in the project. Development teams using LabVIEW to create applications should spend time considering how to optimally convert their system requirements into code.

Designing large applications in LabVIEW, which is no different than software design in any other language, entails dividing an application into logical pieces of manageable size. In fact, LabVIEW graphical programming makes top-down design of an application very straightforward and intuitive.

If you are working on a project with multiple developers, then the architecture of your application should also include an understanding of how work will be distributed and divided. Any project will have several clearly distinguished areas. One team might be tasked with the network connectivity functionality, while another group may be working on the user interface. Establishing APIs is key in order for teams to know what will be the best way in which to communicate with other components of the application.

Note This lesson treats architectural design as a black box, focusing on the objectives and the deliverables of the stage. The details of architectural design are covered in other courses.

Objectives

- Define the software components of your application
- Design the software architecture that developers will use to complete the project
- Define the user interface functionality

Deliverables

- Software model(s)
- Software architecture
- Prototype(s)
- Design document

B. Creating a Software Model

A software model is a visual representation of a software application. Software models are independent of the development environment and can be used to clarify communication between different code modules. This is especially important when the application is complex and that communication becomes difficult to visualize.

Creation of a software model aids in the definition of the project, its components, and how data is passed between different components. The descriptions that you develop for each component as a part of your model can later be used to document the code that is implemented.

Defining System Components

One method for defining the abstract components of a software application is to review the requirements document and list all of the non-redundant nouns. When you have identified all of the nouns, you can divide them into logical, abstract groups. These groups can be used to identify the abstract components of your system.

When you have identified the abstract components of your system, review the requirements document again, this time list all of the verbs and verb phrases that identify system behavior. Match the verbs and verb phrases with the abstract components that perform those actions. Where applicable, break verb phrases into smaller actions to ensure that each module will be cohesive.

Modeling Languages

A modeling language is an artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. Modeling languages can be textual or graphical. This course focuses on graphical modeling languages.

Graphical modeling languages use a diagram technique with named symbols that represent concepts and lines that connect the symbols and represent relationships and various other graphical notation to represent constraints.

There is a wide variety of modeling languages available for use. This course focuses on the use of flowcharts and Unified Modeling Language (UML). For a more comprehensive list of modeling languages, refer to `http://en.wikipedia.org/wiki/Modeling_language`.

Flowchart

Flowcharts are a powerful way to organize ideas related to a piece of software. Flowcharts provide a good understanding of how the task or module will execute by showing how the application will flow from one abstract component to the next. The block diagram programming paradigm used in LabVIEW is easy to understand and similar to a flowchart. Once you have determined how the application will flow, the flowchart makes it easier to convert that design into executable code.

Flowcharts are defined as a programming language according to IEC-1131-3 which is the international standard for programmable controller programming languages.

The flowchart shown in Figure 3-1 demonstrates the functionality of the play module of the TLC application.

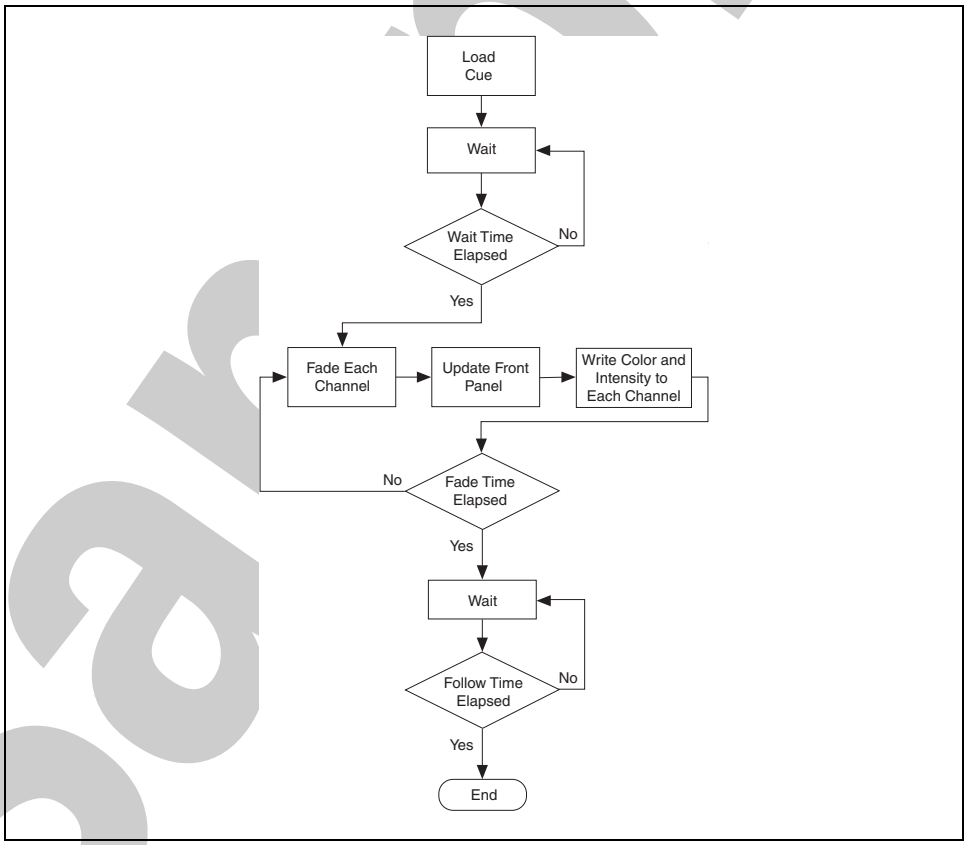

Figure 3-1. TLC Play Flowchart

Components of a Flowchart

- **Start and end symbols**—Represented as circles, ovals or rounded rectangles, usually containing the word Start or End, or another phrase signaling the start or end of a process.
- **Arrows**—An arrow that starts at one symbol and ends at another represents the passing of control from one symbol to the next.
- **Processing steps**—Represented as rectangles. In the TLC Play example, Load Cue, Wait, and Fade Each Channel are all examples of processing steps.
- **Inputs/Outputs**—Represented as parallelograms. The input/output represented by this symbol could be based on interaction with the user, with a file or with hardware components of the system.
- **Decisions**—Represented as diamonds. A Decision usually represents a Yes/No question or a True/False test, but it could ask a question that has more than two possible answers. For each possible answer to the question that is posed, there will be an arrow that originates from the symbol, showing how control is passed depending on the answer to the question. The arrows should always be labeled.

Note More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the pre-defined process symbol.

Unified Modeling Language

The Unified Modeling Language (UML) is an open method used to specify, visualize, modify, construct and document the artifacts of an object-oriented software intensive system under development. This modeling language defines a standard set of graphical representations that can be used to represent a system. As with other modeling languages, UML is not tied to any specific programming environment, and can be used to represent any application.

UML combines best practices from various other data modeling concepts such as entity relationship diagrams, business modeling, object modeling and component modeling.

UML Diagrams

A UML diagram is a partial graphical representation of the model that has been developed for an application. There are thirteen different UML diagrams that fall into two main categories, as shown in Figure 3-2.

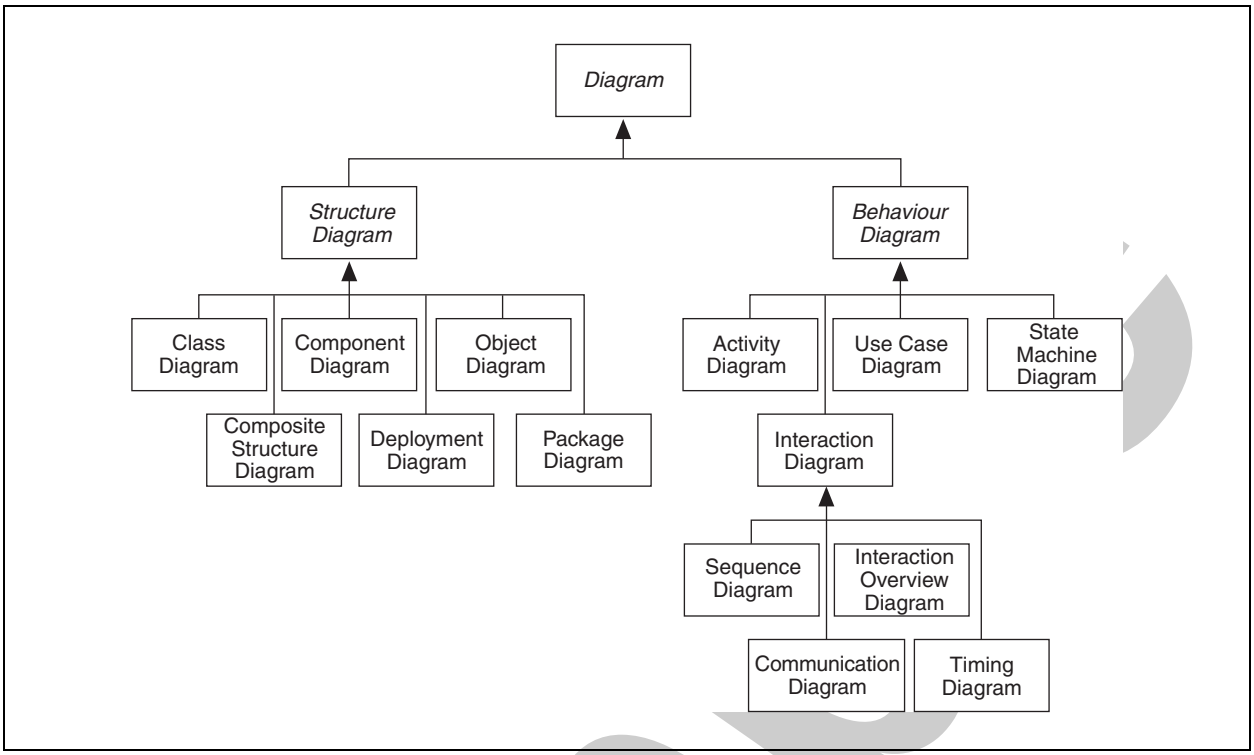

Figure 3-2. Types of UML Diagrams

- **Static (or structural) view**—Emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- **Dynamic (or behavioral) view**—Emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

Structure Diagrams

Structure diagrams emphasize what things must be in the system being modeled:

- **Class diagram**—Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes. A sample class diagram is shown in Figure 3-3.

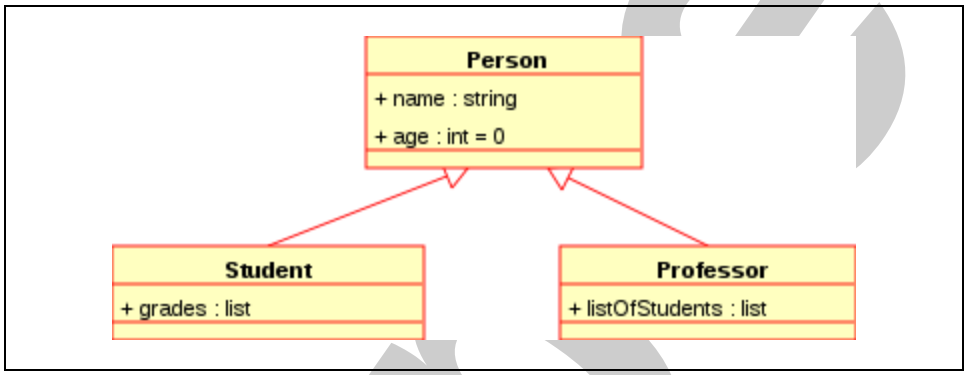

Figure 3-3. Class Diagram

This example shows a parent class (Person) and two child classes (Student and Professor). Each child inherits all of the properties and methods of the parent class and can have its own unique properties and methods.

- **Component diagram**—Depicts how a software system is split up into components and shows the dependencies among these components.
- **Composite structure diagram**—Describes the internal structure of a class and the collaborations that this structure makes possible.
- **Deployment diagram**—Serves to model the hardware used in system implementations, and the execution environments and artifacts deployed on the hardware.
- **Object diagram**—Shows a complete or partial view of the structure of a modeled system at a specific time.
- **Package diagram**—Depicts how a system is split up into logical groupings by showing the dependencies among these groupings.

Behavior diagrams

Behavior diagrams emphasize what must happen in the system being modeled:

- **Activity diagram**—Represents the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

- **State machine diagram**—Represents the behavior of a system using a set of states and the transitions between states. A sample state machine diagram is shown in Figure 3-4.

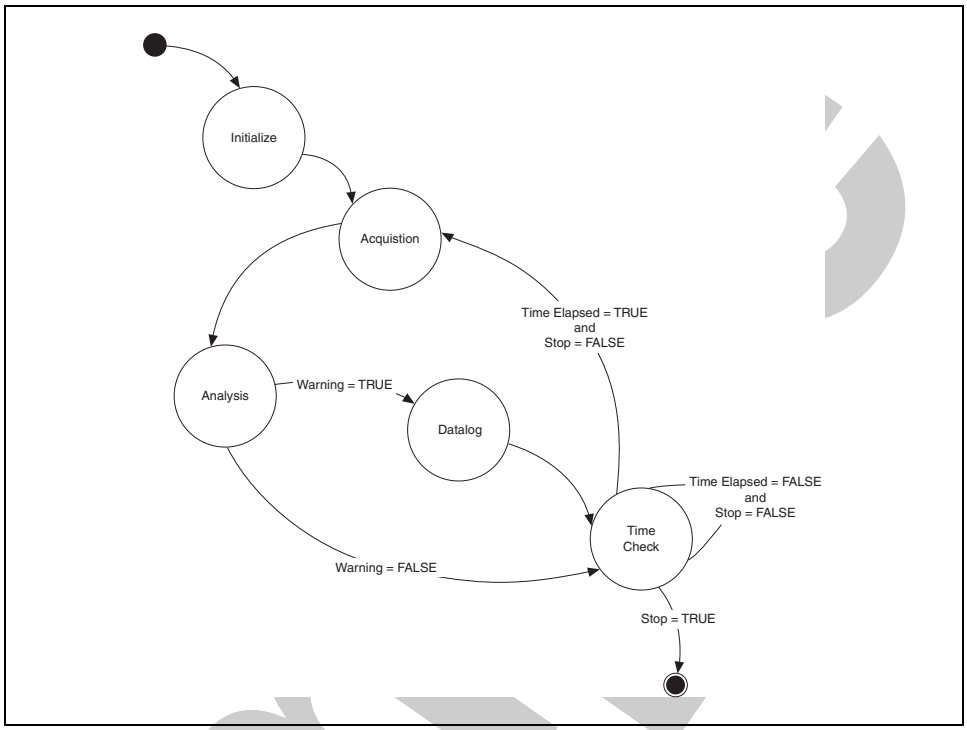

Figure 3-4. State Machine Diagram

This example describes a simple application that acquires, analyzes, and logs data. Circles represent different states, the arrows represent transitions between states, the solid black dot represents the starting point for the diagram, and the black dot with the circle around it represents the end point. If a state can proceed to more than one other state after executing, then the text over the transition indicates the condition that must be met for that transition to occur.

- **Use case diagram**—Shows the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.
- **Interaction diagrams**—Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:
  - **Communication diagram**—Shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.

- **Interaction overview diagram**—Are a type of activity diagram in which the nodes represent interaction diagrams.
- **Sequence diagram**—Shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
- **Timing diagrams**—Are a specific type of interaction diagram, where the focus is on timing constraints.

Modeling Tools

Modeling tools aid in the developing graphical representations to define an application. These tools are used to generate a diagram using a standardized format. In some cases, these tools also provide the ability to generate code. Because you have already learned about flowcharts and UML modeling, The following sections focus on tools you can use to create those models.

Vector-Based Drawing Programs

A flowchart can be created using any drawing program, but vector-based drawing programs are particularly suited for generating the geometric shapes used in a flowchart. Vector graphics are based on geometric primitives such as points, lines, curves, shapes, or polygons that are based on equations.

Any vector-based drawing program can be used to create flowchart diagrams, but not every program will have an underlying data model for sharing data with databases or other programs such as project management systems or spreadsheets. The following tools offer special support for flowchart drawing and data modeling:

- ConceptDraw
- SmartDraw
- Visio
- OmniGraffle

Online flowchart solutions, such as DrawAnywhere, have become available as free, easy to use, and flexible solutions. However, these online tools typically do not meet the power of off-line software like Visio or SmartDraw.

UML Tools

A UML tool is a software application that supports some or all of the notation and semantics associated with the Unified Modeling Language (UML), which is the industry standard general purpose modeling language for software engineering.

UML tool is used broadly here to include application programs which are not exclusively focused on UML, but which support some functions of the Unified Modeling Language, either as an add-on, as a component or as a part of their overall functionality.

- **Diagramming**—Creating and editing UML diagrams; that is diagrams that follow the graphical notation of the Unified Modeling Language.
- **Round-trip engineering**—The ability of a UML tool to perform code generation from models, and model generation from code (a.k.a., reverse engineering), while keeping both the model and the code semantically consistent with each other.
  - **Code generation**—The user creates UML diagrams, which have some connoted model data, and the UML tool derives from the diagrams parts or all of the source code for the software system.
  - **Reverse engineering**—The UML tool reads program source code as input and derives model data and corresponding graphical UML diagrams from it
- **Model transformation**—The capacity to transform a model into another model. For example, one might want to transform a platform-independent domain model into a Java platform-specific model for implementation. It is also possible to refactor UML models to produce more concise and well-formed UML models.

Examples of UML Tools

There is a wide variety of UML tools available that expose different UML functionality. The UML tools listed below have specific functionality that facilitates their use with LabVIEW:

- **Endevo UML Modeller**—A system modeling tool closely integrated in LabVIEW. Can be used to describe and discuss the design and architecture of your test and measurement systems. Can be used to generate LabVIEW code.
- **NI LabVIEW Statechart Module**—Provides a high level of abstraction for designing applications using states, transitions, and events. Statecharts are a type of behavioral diagram within the Unified Modeling Language (UML) specification.

  If you already have the NI LabVIEW Statechart Module installed, refer to the topic *Statechart Module Tutorial Part 1: Modifying a Statechart* in the *LabVIEW Help*. For an online tutorial, refer to the *LabVIEW Statechart Module Tutorial* topic on NI Developer Zone.

For a more comprehensive list of UML tools that do not generate LabVIEW code, refer to `http://en.wikipedia.org/wiki/List_of_UML_tools`.

C. Designing a Software Architecture

A software architecture is defined as "the structure or structures of the system which comprise software components, the externally visible properties of those components, and the relationships among them."[1] Design of the software architecture is a major portion of the software engineering process for the LabVIEW architect.

Basic Design Patterns

The first step in choosing a scalable architecture is to explore the architectures that exist within LabVIEW. Architectures are essential for creating a successful software design. The most common architectures are usually grouped into design patterns.

As a design pattern gains acceptance, it becomes easier to recognize when a it has been used. This recognition helps other developers and yourself to read and make changes to your VIs. It also rescues architects and developers from having to figure out how to solve classic problems for which the best possible answers already exist.

Another advantage of using design patterns is that they have already been tested. They have been refined and improved over time and there are a handful of patterns that have emerged as being exceptionally useful, elegant and powerful. Because these design patterns are such a popular topic of discussion there are a number of online forums with examples, content and thoughts on various patterns and some even ship within LabVIEW.

VI templates based on common design patterns are available from the **File» New** dialog box in LabVIEW.

- Simple VI
- General VI
- State Machine
- Parallel Loop VI
- User INterface Event Handler
- Master/Slave
- Producer/Consumer (Events)
- Producer/Consumer (Data)

1. Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice* (Addison-Wesley, 2003).

Software Development Kit

Quality applications are very rarely written from scratch. In fact, all applications make use of previously written code. For example, when writing an application in LabVIEW, developers use built-in functions to avoid writing common sections of code from scratch. In addition to built-in functions, many developers use components they have created from other projects or received from others. When developing code in a team environment, the team is most effective when they all share the same library of components.

A Software Development Kit (SDK) is a set of tools and resources that assists a developer in creating code for a particular application. The term SDK is traditionally used to describe kits that allow third-party developers to interact with, customize, or expand an application. The SDKs to develop various components for operating systems, such as the Windows Driver Development Kit, are an example of this type of use for an SDK. These types of SDK typically contain documentation and Application Programming Interfaces (APIs), as well as other tools, such as example code, prototyping hardware, software emulators, and/or testing tools.

An SDK is also an excellent format for sharing and distributing resources within a development team. At the beginning of a project, the LabVIEW architect identifies and designs or develops each of the components which will be used throughout the project. The architect designs an API for each component and an overall API for the SDK as a whole. The architect then packages the SDK and distributes it to each developer to use as they create their own portions of the project.

LabVIEW SDKs targeted for either audience include the same types of components, use the same types of API, and use similar distribution techniques. VIs, controls, and other resources that are used when developing modules or plug-ins for the core application are generally similar to the VIs that are necessary to expand or customize the application. Therefore, a well-documented and designed SDK created for internal developers can be easily repackaged for use by third-party developers.

Using an SDK within a project team has the following advantages:

- **Improved code re-use**—Creating an SDK reduces the need for separate developers to develop similar or identical components.
- **Improved consistency**—When developers create their own resources, there may be subtle differences in the appearance and the data type of controls or VIs. For example, if you need to display similar data on a central control panel and on a configuration dialog box, developing a single strict type definition for both controls helps maintain consistency across the application.

- **Reduced chance of integration problems**—If a master set of controls is not used, developers may create controls with subtle differences in data types, such as different cluster or enum orders. These differences may cause significant problems when integrating code modules from different developers. For example, if two developers create a type defined cluster to represent an (x,y) point, one developer might order the cluster (x,y) and the other might order the cluster (y,x). This would probably not result in a broken Run button, but could cause a bug in the program.
- **Abstracted complexity**—By using advanced development techniques, such as XControls, a LabVIEW architect can abstract complex code away from less experienced developers. This can improve the readability and reduce the complexity of the main application, as well as spare developers the need to work around or interact directly with complex code.
- **Improved developer training**—When starting a new project, developers are often at a loss as to how to begin developing their assigned components. This situation is commonly referred to by writers and software developers as “blank page syndrome”. The LabVIEW community sometimes refers to it as “blank block diagram syndrome”. A detailed design and requirements document can go a long way towards getting developers started. However, a well documented SDK with intuitive APIs also provides considerable guidance to new developers. The SDK provides “puzzle pieces” that the developers can build upon to start creating a solution. In addition to helping train developers at the start of a project, an SDK also assists developers who join the project while it is underway. In this case, because the new developer will be using the same tools and resources as the developer before them, the new developer has a much easier time continuing along the same development path.

Potential Components of an SDK

Components in an SDK may be one of two types—reusable or project-specific. Reusable components are general purpose components that are intended for use in multiple projects. Reusable components include utilities, code snippets, or UI controls that solve frequently occurring challenges, such as error handling VIs. Project-specific components are intended for reuse, but only within a given project. Developers use these components within various modules and contexts throughout a specific project, but would find little or no use for them in other projects. Reference management or messaging components are an example.

A LabVIEW SDK can contain many different files and resources. The SDK should include documentation for each of its components. Components developed using good API design, documentation, and naming techniques

are easy for developers to understand and use. However, formal documentation for each API within the SDK should also be provided in the form of one or more help files. Documentation for third-party developers or large development teams may also include training materials designed to get developers started with the components, such as example code or written tutorials.

A LabVIEW SDK is not limited to LabVIEW code. The SDK may include outside code or resources such as DLLs, ActiveX controls, ActiveX servers, or .NET assemblies. Whenever possible, these components should have LabVIEW "wrapper" VI or container to integrate them into the overall API of the SDK.

SDK Controls

LabVIEW controls are independent files that represent a front panel object and/or block diagram data type. Including controls in your SDK will improve the consistency and efficiency of developers as they create VIs.

There are several types of controls available in LabVIEW. Choosing the right type of control can be as important as choosing the data type of a control. The following control types are described in detail in the *LabVIEW Help* and other LabVIEW courses.

- **Custom Control**—Allows multiple developers to start with a consistent look and data type for a control. However, a developer can modify a custom control after adding it to a front panel window, and there is no linking or updating between a control instance and the control file. Use these controls as templates to give developers a starting point. Custom controls, type definitions, and strict type definitions give you more control over the appearance of a control than the standard control properties.
- **Type Definition**—Enforces the data type, but not the appearance of the control. Any changes to the control file update all instances of the control. Use type definitions for non-visible controls, or for constants on the block diagram, such as clusters and enumerated values. Type definitions give developers freedom in the appearance of visible controls while still maintaining a consistent data type.
- **Strict Type Definition**—Enforces the data type and the appearance of a control. The data type and appearance of every control instance are linked to the control file and receive any updates made to the file. Use strict type definitions when you want a control to have a consistent appearance throughout an application.
- **XControl**—Allows you to include code with a control. Refer to the following sections for more information on XControls.

- **Other controls**—You can also develop or use ActiveX or .NET controls for your project. Functionally, these controls are similar to XControls.

SDK VIs

The SDK typically contains different types of VIs and controls for developers to use. Your SDK can include the following types of VIs.

- **SubVIs**—Provide a self-contained, fully functional code segment for your developers to use.
- **Templates**—Provide a starting point for developers to write their own VIs.
- **Snippet VIs**—Provide code segments your developers can add directly to their block diagrams.
- **Express VIs**—Provide a user friendly interface for configuring code.

Application Programming Interface

When you create code, you often want others to interact with it. Most programmers are familiar with the concept of a user interface, which is a visual tool through which an end user interacts with your application. Experienced programmers recognize the importance of putting thought and effort into the design of a user interface. However, many programmers often neglect the fact that they regularly create another type of interface. Whenever you create code that is intended to be called by someone else, you are creating an application programming interface (API). An API is a tool or method used to programmatically call code that you or someone else has written. API design also requires thought and effort, and has similar goals to UI design.

The SDK may also include an API that provides a centralized interface to all of its components. This API usually takes the form of one or more custom palettes that are added to the installation of each LabVIEW development system used for the project. The API may also use one or more Project Libraries to provide namespaces and protect private VIs.

An example of a common API you might encounter is a Dynamically Linked Library. DLLs contain a list of function prototypes, which describe the code inside the DLL and give the developer a way to call the code. Other examples of APIs outside of LabVIEW are ActiveX objects and controls, which provide properties and methods for a developer to call.

LabVIEW APIs

In LabVIEW, an SDK or toolkit typically has a top-level API, which consists of a palette showing all the available components and a library, which defines public and private components. Each component also has an API that describes how a developer interfaces with the component. The

connector pane of each subVI serves as its API. Express VIs have connector panes, as well as their configuration dialogs. Xcontrols use properties, methods, and the XControl's data type as their API. Objects from LabVIEW Object Oriented Programming also use these methods.

Data Structures

Data structures determine how developers will represent and pass data throughout the application. It is up to the LabVIEW architect to define the data structures in order to ensure that all developers use the same representation of data throughout development. To maintain this consistency, you can include your definition of the data structures as part of your SDK. Data structures, as a whole, fall into one of three categories: scalars, arrays, and clusters.

Use scalars for data items that are associated with single, non-related items. Be sure that the data would never need to reside in a more advanced data structure.

- Numeric
- Boolean
- String
- Ring control
- Enumerated control—improves block diagram readability

Use arrays for items of the same type that should be grouped together. You can also use arrays to store data on the block diagram and graphs to display data on the UI.

Use clusters for items of different or similar types that can be grouped together to form a single, coherent data item. Use of clusters will help to reduce wire clutter on your block diagram and enable you to create your own custom data types.

If you use clusters, ring controls or enumerated controls on your front panel or block diagram, remember to build them into type definitions in order to ensure that they are used consistently throughout the application.

Advanced Software Architectures

Customization of design pattern templates and development of new software architectures, while a critical portion of the software engineering process, is outside the scope of this course. For more information on this topic, refer to the following LabVIEW courses:

- *Advanced Architectures in LabVIEW*
- *LabVIEW Object Oriented Programming System Design*

Advanced Architectures in LabVIEW

Learn how to architect an application and then design the components to support the architecture. In this course, you will learn about:

- Architecting an application
- Defining use cases and analyzing advanced design patterns
- Designing an API
- Developing advanced design patterns
- Architecting and designing components for an application

LabVIEW Object Oriented Programming System Design

Learn how to create LabVIEW applications using the object oriented programming architecture In this course, you will learn:

- Basics of object oriented programming
- Object oriented design
- Object oriented application development using by value and by reference methodologies

Architectural Design Review

After you create your architectural and modular design, it is important to review your design. This review should be conducted with another architect, or at least a developer with significant design experience.

In your design review, you are looking at the proposed architecture of either the entire application or an individual module. You are also evaluating the test plan for that code, and checking that any assumptions that you are making about your code are correct. You must decide which features you will absolutely have in your program and which ones would be nice to have, but are not essential.

D. Creating a Prototype

A prototype is a simple, quick implementation of a particular task to demonstrate that the design has the potential to work. The prototype usually has missing features and might have design flaws. In general, prototypes should be thrown away, and the feature should be reimplemented for the final version.

Prototypes are required for some process models (for example, the spiral model and agile model) but not for others.

Note Protoyping is not an excuse for a code-and-fix implementation.

The following sections cover three common types of prototypes:

- User interface prototype
- Proof of concept prototype
- Functional prototype

User Interface Prototype

User interface prototypes 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. Simulate the user interface with the controls and indicators that you need for the final application. Determine how the controls work and what actions would require additional front panels. For more extensive prototypes, the front panels can be tied together.

Systems with many user interface requirements are perfect for prototyping. Determining the method you use to display data or prompt the user for settings is difficult on paper. Instead, consider designing VI front panels with the controls and indicators you need. The focus should be on figuring out how the controls work and how various actions require other front panels.

The block diagram should be left empty or populated with non-functional modules to simulate functionality. This measure will help you to avoid falling into the code and fix trap.

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 other controls?

If you are bidding on a project for a client, using front panel prototypes is an extremely effective way to discuss with the client how you can satisfy his or her requirements. Because you can add and remove controls quickly, especially if the block diagrams are empty, you help customers clarify requirements. If new options are presented, follow those ideas by creating new front panels to illustrate the results. This kind of prototyping helps to define the requirements for a project and gives you a better idea of its scope.

However, do not get carried away with the prototyping process. Limit the amount of time you spend prototyping before you begin. Time limits help to avoid overdoing the prototyping phase. As you incorporate changes, update the requirements and the current design.

Front Panel Prototyping Caveats

Front panel prototyping cannot solve all development problems. You have to be careful how you present the prototype to customers. Prototypes can give an 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 you will rework much of it in the development phase.

Another danger in prototyping is overdoing 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, not with I/O constraints, data types, or algorithm issues in the design. Identifying front panel issues can 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, such as performance benchmarking or top-down design.

Proof of Concept Prototype

This type of prototype is used to test some aspect of the intended design without attempting to precisely simulate the user interface or other functionality. Such prototypes can be used to "prove" out a potential design approach. These types of models are often used to identify which design options will not work, or where further development and testing is necessary. Use this technique for specific new, complicated or unique approaches

Proof of concept prototypes can be developed toward the end of design for the architecture or scattered throughout development for the individual code modules. If you are using the spiral or agile process models, then each iteration will likely utilize some variation of a proof of concept prototype.

Functional Prototype

A functional prototype will, to the greatest extent practical, attempt to simulate the final design, aesthetics, and functionality of the intended design. The development of a fully working prototype is the engineers' final check for design flaws and allows last-minute improvements to be made.

For the waterfall model, functional prototypes are not created until nearly the end of the development phase, but they should be planned for during the design stage. However, for the agile process model, development of a functional prototype may begin very early in the process.

E. Developing a Design Document

A design document is a written description of a software product, that a software designer writes in order to give a software development team an overall guidance of the architecture of the software project.

For the waterfall model, the design document should be a stable, controlled, reference that does not change much after it is created. This document is the culmination of the design stage of the waterfall model and it will be referred back to frequently by the architect and the developers throughout development and testing of the application.

The application design may be encapsulated in one document or, more likely, over several documents for the different software components.

Refer to IEEE 1016 *Recommended Practice for Software Design Descriptions* for guidelines on how to organize your design document.

Data Design

This portion of the design document describes the data structures that will reside within the software. Selection and development of data structures should be based upon the necessary attributes for the data objects and the relationships between the objects. Data can be grouped according to the type of the data (array) or by relationships between the data objects (cluster).

The choice of how data structures are designed can have significant impact on the execution speed and implementation complexity of your code. Refer to the *Designing the Project* section and *Implementing the User Interface* section of *LabVIEW Intermediate I: Successful Development Practices* for more information on the selection and implementation of different data structures.

Architecture Design

This portion of the design document maps the flow of information through the program structure. One method of showing the flow of data into and out of the module is to use transformation mapping. Transformation mapping demonstrates the boundaries between incoming and outgoing data by:

- Listing inputs on the left side of the front panel
- Listing outputs on the right side of the front panel
- Documenting how the inputs are used to generate the output values in the area between the inputs and outputs

Software models may be included in the architecture design to illustrate the relationships between different software components within the module.

Interface Design

This portion of the design document describes the internal and external program interfaces that are used in the application. If will be using an SDK to aid in development, you should document the API for the SDK here. You should also include documentation regarding any user interface prototypes that have been developed.

Procedural Design

This portion of the design document describes the code algorithms that should be implemented for each software component using graphical, tabular, and textual notations. This documentation enables the architect to provide details on how each module should be implemented to the developers. This section of the design document forms the basis for all subsequent development work for the application.

Self-Review: Quiz

1. What is the main reason to develop a software model?
   a. To provide a method of programmatically calling code that has already been written
   b. The application will consist of modules that communicate with each other in a manner that is difficult to visualize
   c. The software engineering process requires that every application be modeled before development begins

Match each type of prototype to its intended purpose:

2. User interface prototype	a. Simulate the final design, aesthetics, and functionality of the final design
3. Proof of concept prototype	b. Test a single feature of the intended design
4. Functional prototype	c. Capture the intended design aesthetic

Self-Review: Quiz Answers

1. What is the main reason to develop a software model?
   a. To provide a method of programmatically calling code that has already been written—This is a reason to develop an application programming interface (API), not a software model.
   b. The application will consist of modules that communicate with each other in a manner that is difficult to visualize—Software models should chiefly be used for more complicated applications.
   c. The software engineering process requires that every application be modeled before development begins—The software engineering process makes recommendations for completion of a project. For simple applications that you can easily visualize, creation of a software model is probably overkill.

Match each type of prototype to its intended purpose:

2. User interface prototype	c. Capture the intended design aesthetic
3. Proof of concept prototype	b. Test a single feature of the intended design
4. Functional prototype	a. Simulate the final design, aesthetics, and functionality of the final design

Notes