Object-Oriented Design and Programming in LabVIEW™ Exercises

Course Software Version 2010
November 2010 Edition
Part Number 325617A-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

To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code feedback.

Contents

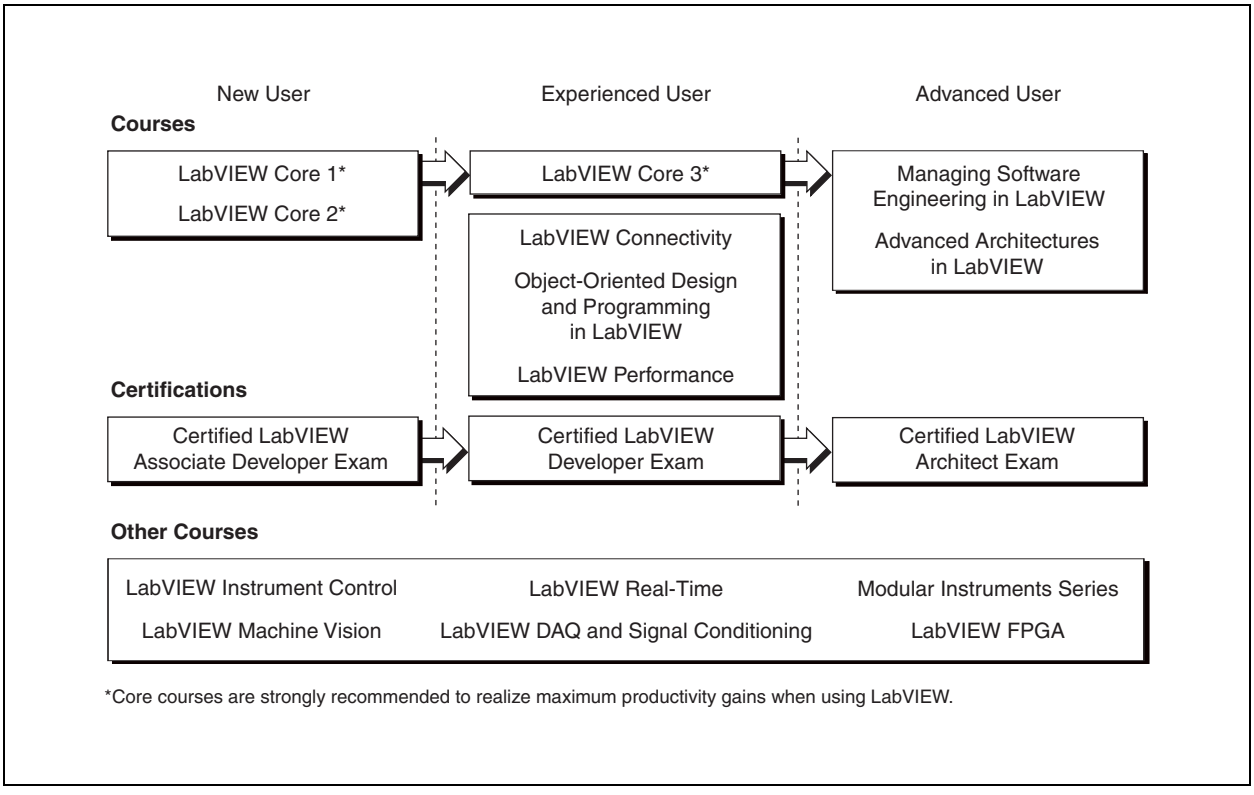
Student Guide

Thank you for purchasing the *Object-Oriented Design and Programming in LabVIEW* course kit. This course manual and the accompanying software are used in the two-day, hands-on *Object-Oriented Design and Programming with LabVIEW* course.

You can apply the full purchase price of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` to register for a course and to access course schedules, syllabi, and training center location information.

A. NI Certification

The Object-Oriented Design and Programming in LabVIEW course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for exams to become an NI Certified LabVIEW Developer and NI Certified LabVIEW Architect. The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.



B. Course Description

This course assumes that you have taken the *LabVIEW Core 3* course or have equivalent experience.

The course is divided into lessons, each covering a topic or a set of topics. Each lesson consists of the following parts:

- An introduction that describes what you will learn
- A discussion of the topics
- A set of exercises that reinforces the topics presented in the discussion

  Some lessons include optional exercises or challenge steps to complete if time permits
- A summary that outlines important concepts and skills taught in the lesson

Note For course manual updates and corrections, refer to `ni.com/info` and enter the Info Code `lvoop`.

C. What You Need to Get Started

Before you use this manual, make sure you have the following items:

- ❑ Windows XP or later
- ❑ LabVIEW Professional Development System 2010 or later
- ❑ *Object-Oriented Design and Programming in LabVIEW* CD, which contains the following folders and files:

Folder Name	Description
`Exercises`	Contains all the VIs and support files needed to complete the exercises in this course
`Solutions`	Contains completed versions of the VIs you build in the exercises for this course

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer. The **Object-Oriented Design and Programming in LabVIEW Course Setup** dialog box appears.
2. Click **Install the Course Material**.
3. Follow the onscreen instructions to complete installation and setup.

Exercise files are located in the `<Exercises>\Object-Oriented Design and Programming in LabVIEW` folder.

Tip Folder names in angle brackets, such as `<Exercises>`, refer to folders on the root directory of your computer.

Repairing or Removing Course Material

You can repair or remove the course material using the Add or Remove Programs feature on the Windows Control Panel. Repair the course manual to overwrite existing course material with the original, unedited versions of the files. Remove the course material if you no longer need the files on your computer.

E. Course Goals

This course prepares you to:

- Determine the appropriateness of using an object-oriented approach to solve the problem
- Design an application using object-oriented design principles
- Implement a basic class hierarchy using LabVIEW classes
- Modify an existing application written in G to replace common patterns with objects

This course does *not* present the following topic:

- Developing a complete application for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create

F. Course Conventions

The following conventions are used in this course manual:

»	The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **Options»Settings»General** directs you to pull down the **Options** menu, select the **Settings** item, and select **General** from the last dialog box.
	This icon denotes a tip, which alerts you to advisory information.
	This icon denotes a note, which alerts you to important information.
bold	Bold text denotes items that you must select or click 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. Italic text 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 enter from the keyboard, sections of code, programming examples, and syntax examples. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
`<Exercises>/.../`	This short-hand denotes the `Object-Oriented Design and Programming in LabVIEW` directory located in the `Exercises` directory.
`<Solutions>/.../`	This short-hand denotes the `Object-Oriented Design and Programming in LabVIEW` directory located in the `Solutions` directory.

2

Designing an Object-Oriented Application

Exercise 2-1 Identify Classes and Methods

Goal

Identify the classes required to complete the course project.

Scenario

You have been asked to develop a demo application for a company that makes devices for playing sound. You meet with a company representative who explains the job. The company makes two devices, a sound card that actually plays sound and a visualizer that renders the content of the sound wave visually. You notice the representative refers to both devices as “sound players.”

They want you to create a demo program that a sales employee could use to show off the company’s players. The application should allow the user to choose which of the two players to demonstrate, provide the ability for the user to configure the chosen player, and finally run a demonstration on that player.

While offering you the job, the company representative mentions that the company has other sound players in development, and if you do well on this project, you may be asked to develop demos for future products. Because you are confident you will get those future jobs, you decide that you need to make the first demo flexible so you can generate demos for those future products quickly and with as much code reuse as possible.

The representative lists several sounds that the demo should be able to perform on the players. Unfortunately, the exact order of these sounds is something that will be decided late in the development cycle due to disagreements within the company about what sounds best demonstrate the capabilities of the players. You think you might be asked to make a variety of late-in-development changes to the sounds themselves, so it seems wise to develop a framework for quickly plugging in new sounds, instead of just hardcoding a number of waveforms.

The company provides you with the programming API for both players. When you review them you can tell that although they are similar, these

APIs were written by different programming teams that did not coordinate with each other. Writing one application that can run either player will not be as simple as putting a case structure around subVI calls.

When you get back to your office, you take your notes from the customer meeting and rewrite them to be very explicit about what it is you think you are being asked to build, rewording the customer's sometimes vague requirements into more specific technical requirements.

Use this document to identify the classes that will be needed to complete the project and the methods that will be needed for each class.

Design

Identify the nouns in the requirements document. You will use these to determine the classes required to complete the project.

Identify the action verbs from the requirements document. You will use these to define the methods for each class you identified.

Implementation

Part A: Identify Potential Classes

Identify all of the potential classes for the Sound Player Demo project by reviewing the problem statement and identifying all of the nouns.

1. Review the Sound Player Demo Problem Statement.

Sound Player Demo Project Problem Statement

Develop an application that demonstrates a sound playing device. When the application runs, it should:

1. Allow the user to select the sound player to be demonstrated.
2. Allow the user to configure that player (different players require different configuration options).
3. Play the fixed set of chosen sounds on the player.
4. Shut down the system when the demo has completed.

Your demo application will show off one of two types of sound player: a sound card or a visualizer. At the start of the application, the demo will offer the user the choice of which player to demonstrate.

The application must direct the user to setup the application's sound player. Per the sound card API, the user must choose a device ID, the number of samples per channel, the sound format and the volume. The sound card will

configure itself to match the user's specifications. During shutdown, the demo application will close the device ID.

If the visualizer is selected as the sound player, the visualizer API will require that the user choose the bounding rectangle on the screen where the visualization should appear and the colors that should be used for rendering the waveforms. During shutdown, the demo application must close the visualization window.

The sounds to be played will be chosen ahead of time, but should be configurable from some sort of human-readable file to make it easy to change which sounds to play. The config file should allow the user to select from the following types of sound:

- **Silence**—Create a sound that contains a silence of user-defined duration.
- **Single Tone**—Create a sound that contains a tone of user-defined duration and frequency.
- **Recorded Sound**—Create a sound based on a WAV file whose path is provided by the user. The sound file path should be verified and return an error if the file does not exist.
- **Aggregate Sound**—Create a sound that consists of any combination of other types of sound that will be played concurrently.
- **Fade In/Out**—Create a sound that has an existing sound and fades it in or out over time.
- **Other Modifications**—the company might have other modifications, like Fade, that it decides to include in the demo later, such as echo or autotuning.

The above sounds must each be able to generate an array of waveforms since that is the format ultimately supported by the player APIs provided by the company.

The sound player will play the sounds: the sound card will output a sound from the system speakers, and the visualizer will open a window to display a graph of the waveform.

End of Sound Demo Player Problem Statement

2. Identify the potential classes required for the application

   ❑ Search the problem statement for nouns and underline them.

Part B: Refine the List of Potential Classes

Not all of the nouns from the problem statement will become classes. Review a list of the nouns found in that document and create a refined list of classes that could serve as a starting point for creating and developing your object-oriented application.

1. Open `<Solutions>\...\Exercise 2-1\Part A - Identify Potential Classes.docx`. This document contains a list of potential classes found by identifying the nouns in the problem statement.
2. Revise this list of potential classes using the following guidelines:

   - ❑ Remove any nouns that seem redundant when looking at the larger list. Keep the noun that sounds more specific or appropriate. For example, if you listed both `demo` and `demo application`, only keep `demo application`.
   - ❑ Generally, you should only list the singular form of each noun.
   - ❑ Remove any nouns that seem more like attributes of other nouns. For example, `frequency` is more of an attribute of `single tone` than a class of its own.
   - ❑ Remove any nouns that represent library functions or LabVIEW features that you will not have to develop. For example, `waveform` is a LabVIEW datatype that you do not have to develop.
   - ❑ Remove any nouns that represent system pieces that your application does not have to address and will not have to address in the future. For example, the `system speakers` will always be handled by the sound card API, which you do not have to develop.
   - ❑ Remove any nouns that seem like it would be unnecessary to develop a class to encapsulate it. For example, `volume` is a simple numeric value that probably does not require a class wrapper to define and manage it.

3. Review your revised list of nouns with the instructor and the other students.

   - ❑ Open `<Solutions>\...\Exercise 2-1\Part B - Refine the List of Potential Classes.docx`.
   - ❑ Review the identified **Reason for Elimination** for each potential class.

Note In practice, you may discover a need for additional classes after you begin code development. This technique is intended to serve as a starting point for developing your application.

Part C: Identify the Public Methods for Each Class

1. Identify the public methods that will be performed by each of the remaining classes.

Note These methods represent the public interface methods for each class and how the class interacts with other classes. You will need additional implementation-specific methods, but it is nearly impossible to predict what these will be until you begin development.

- ❑ Search the problem statement for action verbs and list them in the **Methods** column of Table 2-1 in the row of the class that should perform that action.

Tip For now, ignore verbs like “is,” “contains,” and “uses.” You will examine those verbs in Exercise 2-2 when you identify class relationships.

One noun, Application, has been abstracted for you, already.

Table 2-1. Classes and Methods

Classes	Methods
Application	Run, setup, execute, shutdown
Sound Player	
Sound	
Demo Application	

Table 2-1. Classes and Methods (Continued)

Classes	Methods
Sound Card	
Visualizer	
Silence	
Single Tone	
Recorded Sound	
Aggregate Sound	
Fade	
Modified Sound	

2. Revise your list of methods for clarity.

   - ❑ When appropriate, create a verb phrase to clarify the action described by the verb. Some verbs, like `run` or `set up`, are adequate by themselves. Others, like `choose` or `select`, do not make much sense without including their direct objects.
   - ❑ Some verbs may be combined if the same action is being taken on multiple direct objects. For example, `set path` and `verify path` could be combined into `set and verify path`.
   - ❑ Some verbs may be listed next to more than one class. This is not a problem as long as the action should be performed by both classes.
   - ❑ Rephrase implementation-specific verbs so they describe actions from the interface perspective. For example, `define` could be rephrased as `select`.
   - ❑ Remove any redundant verbs. For example, `set sound` and `select sound` do not both need to be listed for the same class.

Note Not every class will have specific methods associated with it. Child classes may inherit their methods from an ancestor, and some ancestors may expect to be overridden by their children.

3. Review your revised list of methods with the instructor and the other students.

   - ❑ Compare your class abstractions to `Part C - Identify the Public Methods for Each Class.docx`, located in `<Solutions>\...\Exercise 2-1`.

Note As with most solutions, the abstractions described in `Part C - Identify the Public Methods for Each Class.docx` are not the only solution that could be used to design the application. However, for the sake of simplicity, it is the solution you will use as a basis for the exercises in Lessons 3, 4, and 5.

End of Exercise 2-1

Exercise 2-2 Identify Class Relationships

Goal

Identify the relationships between the classes that you identified in Exercise 2-1.

Scenario

Now that you have identified the classes you need for the course project and the data and methods associated with each class, identify the relationships that exist between them.

Design

There are three possible relationships that can exist between two classes:

- **Inheritance**—One class is a more specific example of another class.
- **Composition**—One class contains an instance of another class.
- **Usage**—One class has a method (other than a simple accessor method) that takes another class as input or produces another class as output.

Not every class will exhibit every one of these relationships.

Implementation

Part A: Inheritance

1. Identify the inheritance relationships that exist for each class.

   ❑ For each class, identify the single class that is a more general instance of that class and write it in the **Parent** column of Table 2-2. If you cannot identify a parent, leave it blank.

Tip In the problem statement, look for verbs like “is,” nouns like “instance,” and phrases like “type of” when identifying inheritance relationships.

Table 2-2. Inheritance and Composition

Class	Parent	Components
Application		
Sound Player		
Sound		
Demo Application		

Table 2-2. Inheritance and Composition (Continued)

Class	Parent	Components
Sound Card		
Visualizer		
Silence		
Single Tone		
Recorded Sound		
Aggregate Sound		
Fade		
Modified Sound		

2. Discuss the parent classes that you identified with the instructor and the other students.

   ❑ Compare your list of parent classes to the list provided in `<Solutions>\...\Exercise 2-2\Part A - Inheritance.docx`.

Part B: Composition

1. Identify classes that are components of other classes.

Tip In the problem statement, look for key words and phrases such as “contains,” “consists of,” “is part of,” “stores,” and so on to locate this relationship between two classes.

   ❑ In the **Components** column of Table 2-2, identify any classes that are components of the class listed on that row. If there are no component classes, then leave the space blank.

Tip You do not need to restate any components that are already contained by the parent class.

2. Discuss the component classes that you identified with the instructor and the other students.

   ❑ Compare your list of component classes to the list provided in `<Solutions>\...\Exercise 2-2\Part B - Composition.docx`.

Part C: Usage

1. Identify class methods that use other classes as input or output.

   ❑ For each method listed in Table 2-3, determine if it needs any other class as an input or produces another class as an output. If so, list that class in the **Uses** column. If that method does not use any other classes, then leave the space blank.

Tip If you listed a class as a component of another class in Table 2-2, then we already know that it is used by that class and the used class does not need to be listed in this table.

Table 2-3. Usage

Class	Method	Uses
Application	Run	
	Setup	
	Execute	
	Shutdown	
Sound Player	Select Player	
	Configure Player	
	Play Sounds	
Sound	Configure Sound	
Demo Application	Shutdown	
	Choose Player	
	Setup Player	
	Read File	
Sound Card	Configure Sound Card	
	Close ID	
	Output from Speakers	
Visualizer	Configure Visualizer	
	Close Visualization Window	
	Open Visualization Window	
	Display Waveform Graph	

Table 2-3. Usage (Continued)

Class	Method	Uses
Silence	Create Silence	
	Select Duration	
	Generate Waveform	
Single Tone	Create Single Tone	
	Select Duration	
	Select Frequency	
	Generate Waveform	
Recorded Sound	Create Recorded Sound	
	Set and Verify Path	
	Generate Waveform	
Aggregate Sound	Create Aggregate Sound	
	Play Concurrently	
	Generate Waveform	
Fade	Create Fade	
	Fade In	
	Fade Out	
	Generate Waveform	
Modified Sound	Generate Waveform	

Tip Usage relationships determine which classes are used as I/O for methods of other classes. You could also use this table to identify the non-class I/O for each method.

2. Discuss the usage relationships that you identified with the instructor and the other students.

   ❑ Compare your list of used classes to the list provided in `<Solutions>\...\Exercise 2-2\Part C - Usage.docx`.

Part D: Class Hierarchy

In the space provided on the next page, draw a class hierarchy showing the relationships between the classes.

1. Arrange the identified classes in the space provided.
   - ❑ Place any classes for which you have not identified a parent at the top of the hierarchy.
   - ❑ Place the children that inherit from each class in a row under that class.
   - ❑ Place the children that inherit from the second row of your hierarchy under their parents, and so on.
2. Draw arrows and lines to identify the different relationships between classes.
   - ❑ Draw an arrow from each child to its parent.
   - ❑ Draw a line from each class to each of the classes that it is composed of. Place a diamond at the component end of the line.
   - ❑ Draw an arrow with a dotted line from each class to the classes that it uses.

Use this space to draw your hierarchy.

3. Discuss your hierarchy and the relationships that you identified with your classmates.

   - ❑ Did they identify the same relationships that you did? What are the differences?

   - ❑ Compare your class hierarchy to the one provided in `<Solutions>\...\Exercise 2-2\Part D - Class Hierarchy.docx`. The class hierarchy shown in this document will be used to define the classes and the relationships between them for the remaining exercises of this course.

Note As with most problem solutions, the hierarchy described in `Part D - Class Hierarchy.docx` is not the only solution that could be used to design the application. However, for the sake of simplicity, it is the solution that we will use as a basis for the exercises in Lessons 3, 4, and 5.

End of Exercise 2-2

Notes

Notes

3

Object-Oriented Programming in LabVIEW

Exercise 3-1 Creating a LabVIEW Class

Goal

Create a new class and customize its icon and wire appearance.

Scenario

In order to demonstrate a sound player, you must be able to create and configure different types of sounds. Create a new class that will represent a generic sound. Customize its icon and wire appearance so that it will be differentiated from other classes and block diagram items.

Design

- **Data**—Each specific type of sound will have different data used to define that sound. For this implementation, `Sound.lvclass` will not have any unique data.
- **Icon template**—Differentiates method VIs for this class from the methods for other classes on the block diagram.
- **Wire appearance**—Differentiates wires carrying objects for this class from other data on the block diagram.

Implementation

1. Open LabVIEW and create a new project. Save the project as `Sound Player Demo.lvproj` in `<Exercises>\...\Sound Player Demo Project`.
2. Create a virtual folder named `Sounds`.
3. Create a new class.
   - ❑ Right-click **Sounds** and select **New»Class**.
   - ❑ Name the class `Sound`.

4. Save the class and the project.

   - ❑ In the Project Explorer window, select **File»Save All**. When prompted to name the class, save it to `<Exercises>\...\Sound Player Demo Project\Sound` and save the class as `Sound.lvclass`.

5. Create an icon template for this class. This template is used for all method VIs you create later for this class.

Note You can modify the icon template however you like in order to distinguish the VIs you create in later exercises from other classes. However, the block diagrams that you create and modify later will differ from the diagrams in this manual.

- ❑ Right-click `Sound.lvclass` in the Project Explorer window and select **Properties**.
- ❑ On the General Settings page, click **Edit Icon**.
- ❑ In the Icon Editor dialog, use the Fill tool to color the top bar red.

Tip When selecting icon background colors, you want to be sure that there is sufficient contrast between the background color and the rest of the icon to make any images or text easily readable.

- ❑ Select the **Icon Text** tab of the Icon Editor and type `Sound` into Line 1 Text.
- ❑ Set the Alignment to `center`.
- ❑ Click **OK** to close the Icon Editor.

6. Create a custom wire appearance for this class. This will help to differentiate wires carrying objects of this class from other wires on the block diagram.

   - ❑ In the Class Properties dialog box, select **Wire Appearance** from the Category list.
   - ❑ Select **Use custom design**.
   - ❑ Modify the wire appearance so that the Edge foreground and Center foreground colors are both red.

Tip You have selected a color that matches the color scheme of the icon template that you created. When selecting a wire color, do not select a color that is too bright or light-colored. The wires might be difficult to see on the block diagram.

7. Document the purpose of the class.

   - ❑ In the Class Properties dialog box, select **Documentation** from the Category list.
   - ❑ In the **Description** text box, enter a description of the purpose of this class. For example: `Abstraction of a sound that can be played on any implementation of Sound Player`.
   - ❑ Click **OK** to close the Class Properties dialog box.

8. Save `Sound Player Demo.lvproj` and `Sound.lvclass`.

End of Exercise 3-1

Exercise 3-2 Encapsulating Methods within a Class

Goal

Create a method for the class you defined in Exercise 3-1.

Scenario

Now that you have created your class, it is time to create a VI representing the method of the class and determine the appropriate scope for this method.

Design

`Sound.lvclass` will have one method associated with it:

- `Generate Waveforms.vi`—This method creates a waveform for the sound to be played. The block diagram will be mostly blank because this method will be modified in later exercises.

Note This method is defined and implemented for a generic sound.

Implementation

1. If it is not already open, open `Sound Player Demo.lvproj`.
2. Create a method for `Sound.lvclass` that you will use to create a waveform.
   - ❑ Right-click **Sound.lvclass** and select **New»VI**.

Note You examine many of the other options for new VIs that can be created later.

   - ❑ Save the VI as `Generate Waveforms.vi` in `<Exercises>\...\Sound Player Demo Project\Sound`.
3. Create the front panel shown in Figure 3-1.

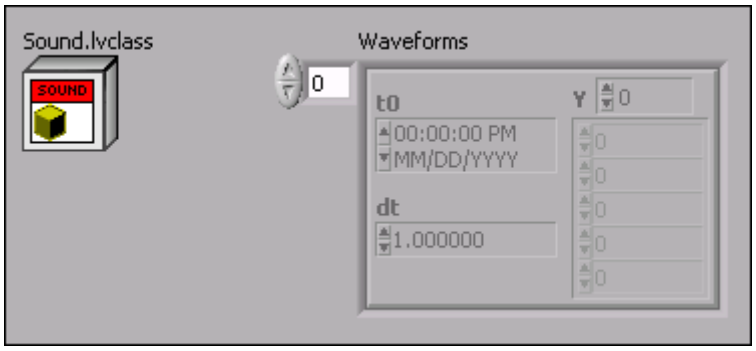


Figure 3-1. Sound.lvclass:Generate Waveforms.vi Front Panel

- ❑ Create the Sound terminal by selecting **Sound.ctl** in the Project Explorer window and dragging the control to the front panel.
- ❑ Create the Waveforms indicator terminal using the Array and Waveform controls, located in the Array and I/O palettes, respectively.

4. Modify the icon and connector pane for `Generate Waveforms.vi` as shown in Figure 3-2.

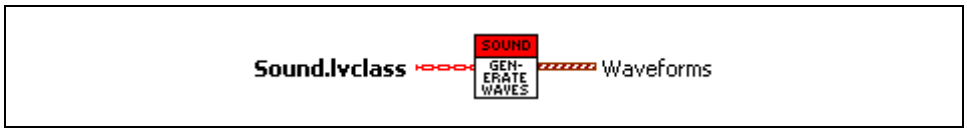


Figure 3-2. Sound.lvclass:Generate Waveforms.vi Icon and Connector Pane

- ❑ Use the 1:1 connector pane pattern with one input and one output.
- ❑ Make the **Sound.lvclass** input terminal **Required**.

5. The block diagram does not need to be edited because the children of `Sound.lvclass` will override this method.

6. Document the purpose of this method.

- ❑ Select **File»VI Properties**.
- ❑ In the VI Properties dialog box, select **Documentation** from the Category list.
- ❑ In the **VI description** text box, describe the purpose of this method. For example: `This method creates a waveform for the sound to be played. This method MUST be overridden by children of Sound.`
- ❑ Click on **OK** to close the VI Properties dialog box.

7. Save and close the VI.

   ❑ Your project should now resemble Figure 3-3.

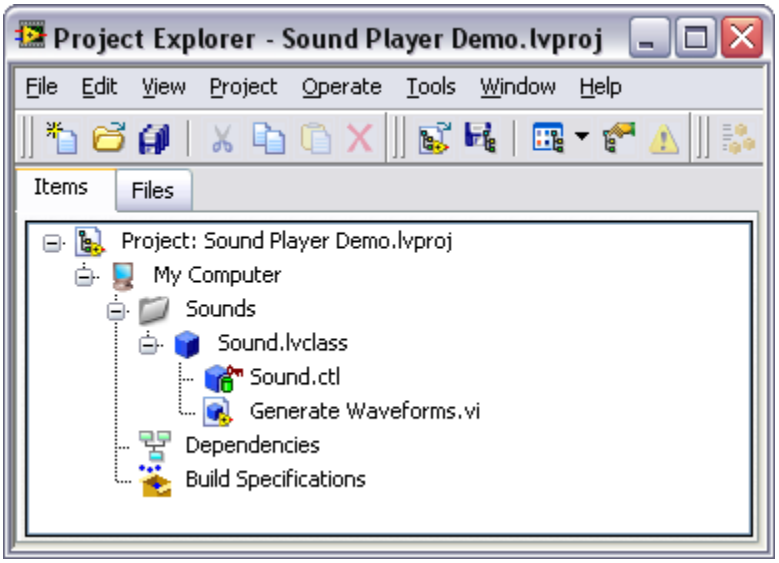


Figure 3-3. Test Sound Player.lvproj with Sound.lvclass

Note The red key next to `Sound.ctl` indicates that the data of the class is private and can only be accessed by VIs within this class. `Generate Waveform.vi` is currently configured as a public method that can be accessed by any VI.

End of Exercise 3-2

Exercise 3-3 Inheriting Methods from a Parent Class

Goal

Create a new class that will inherit from its parent.

Scenario

Create a new class for a single tone, which is a more specific type of sound. In addition to the data and methods defined by its parent, it will also have new data and methods that are particular to this specific type of sound.

Design

Data

- **Duration (numeric)**—The period of time (in seconds) that the tone will last.
- **Frequency (numeric)**—The frequency (in Hertz) of the tone determines its pitch.

Method

- **Generate Tone**—This method sets the parameters that will be used to create a waveform for Single Tone.

Implementation

1. If it is not already open, open `Sound Player Demo.lvproj`.
2. In the Sounds virtual folder, create a new class named `Single Tone.lvclass`.
3. Modify `Single Tone.lvclass` to inherit from `Sound.lvclass`.
   - ❑ Right-click **Single Tone.lvclass** and select **Properties**.
   - ❑ In the Class Properties dialog box, select **Inheritance** from the Category list.
   - ❑ Click **Change Inheritance**.
   - ❑ In the Change Inheritance dialog box, select `Sound.lvclass` from the **All Classes in Project** list. Click **Inherit from Selected**.
   - ❑ Do not close the Class Properties dialog box.

4. Create an icon template that uses the same color scheme as `Sound.lvclass` with different text in the title bar.

   - ❑ In the Class Properties dialog box, select **General Settings** from the Category list and click **Edit Icon**.
   - ❑ Modify the title bar of the icon so that its background color is red and enter `Tone` in the Line 1 text field and click **OK**.

Tip It is good practice for class icons to share some common theme between the parent class and its siblings. In this case we are using a common color scheme, but you could use a common glyph or other method for showing that relationship.

5. In the Class Properties dialog box, select **Wire Appearance** from the Category list and verify that the wire appearance is set to **Use parent's/default design**.
6. Document the purpose of the class.
   - ❑ In the Class Properties dialog box, select **Documentation** from the Category list.
   - ❑ In the Description text box, enter a description of the purpose of this class. For example: `This sound represents a single tone of a given frequency and duration.`
   - ❑ Click **OK** in the Class Properties dialog.
7. Modify `Single Tone.ctl` to contain numerics for the Frequency and Duration of the sound, as shown in Figure 3-4.

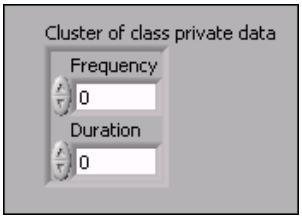


Figure 3-4. Single Tone Class Data

   - ❑ Double-click **Single Tone.ctl** from the Project Explorer window to open the class private data for the class.
   - ❑ Add two numerics to the cluster. Label them `Frequency` and `Duration`.
   - ❑ Save `Single Tone.ctl`. This prompts you to save `Single Tone.lvclass`. Save `Single Tone.lvclass` to

`<Exercises>\...\Sound Player Demo Project\Single Tone`.

8. Create an accessor method to write the Frequency and Duration data for the class.

   - ❑ Right click **Single Tone.lvclass** and select **New»VI for Data Member Access**.
   - ❑ In the Create Accessor dialog box, select `Frequency` from the **Data members of Single Tone.lvclass** list.
   - ❑ Select **Write** from the Access list.
   - ❑ Verify that **Create static accessor** is selected.
   - ❑ Expand the **Advanced Options** section of the Create Accessor dialog box and remove the checkmark from the **Include error handling terminals** checkbox.
   - ❑ Click **Create**.
   - ❑ Save the VI as `Generate Tone.vi` in `<Exercises>\...\Sound Player Demo Project\Single Tone`.

9. Modify the block diagram of `Generate Tone.vi` as shown in Figure 3-5.

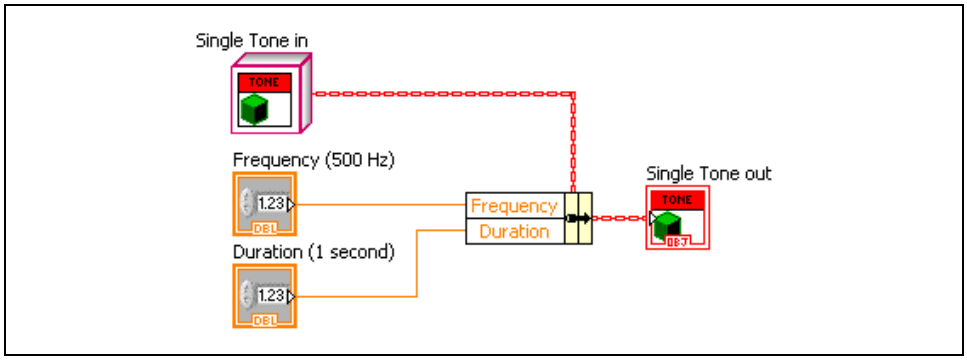


Figure 3-5. Generate Tone.vi Block Diagram

   - ❑ Right-click **Single Tone in** and select **Change to Constant**. Modify the constant to make the label visible.
   - ❑ Expand the Bundle by Name function so that it includes both Frequency and Duration.
   - ❑ Right-click the Duration input to Unbundle by Name and select **Create»Control**.

- ❑ Rename Frequency and Duration as shown to reflect the default values that they will use.

Note The block diagrams in this course are configured so that controls and indicators are viewed as icons. This is done to show the class icons to make the block diagrams easier to read. If you choose to deselect this option, your block diagrams will look slightly different.

10. Modify the front panel as shown in Figure 3-6.

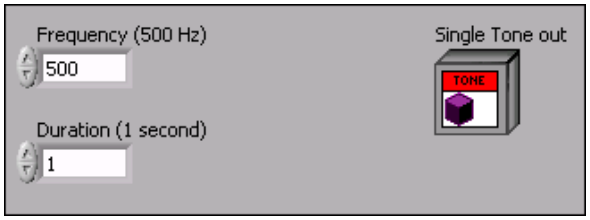


Figure 3-6. Generate Tone.vi Front Panel

- ❑ Change the values of Frequency (500 Hz) and Duration (1 second) as shown.
- ❑ Select **Edit»Make Current Values Default** so that those will be the default values used.

11. Modify the icon and connector pane as shown in Figure 3-7.

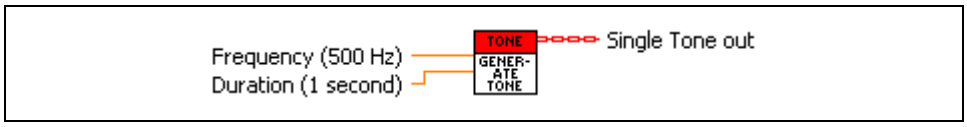


Figure 3-7. Generate tone.vi Icon and Connector Pane

Note Be sure to change the Frequency and Duration inputs from Required to Recommended.

12. Document the purpose of this method.

- ❑ Select **File»VI Properties**.
- ❑ In the VI Properties dialog box, select **Documentation** from the Category list.
- ❑ In the VI Description text box, describe the purpose of this method. For example: `This method sets the parameters that will be used to create a waveform for Single Tone.`
- ❑ Click **OK** to close the VI Properties dialog box.

13. Save and close `Generate Tone.vi`.

14. Select **Save All** from the Project Explorer window to save the changes to the project and `Single Tone.lvclass`.

Import Other Sounds

While you were developing `Single Tone.lvclass`, other developers on your team created `Silence.lvclass`, `Recorded Sound.lvclass`, `Aggregate Sound.lvclass`, `Sound Modification.lvclass`, and `Fade.lvclass`.

1. Add the classes created by the other developers on your team to the project.

   - ❑ In Windows Explorer navigate to `<Exercises>\...\Sound Player Demo Project\Silence`.
   - ❑ Select `Silence.lvclass` and drag it into the Sounds virtual folder of your project.
   - ❑ Repeat these steps for Recorded Sound, Aggregate Sound, Sound Modification, and Fade so that all five of the classes built for you have been added to the project.

❑ Rearrange the classes in the Sounds virtual folder so that your Project Explorer window resembles Figure 3-8.

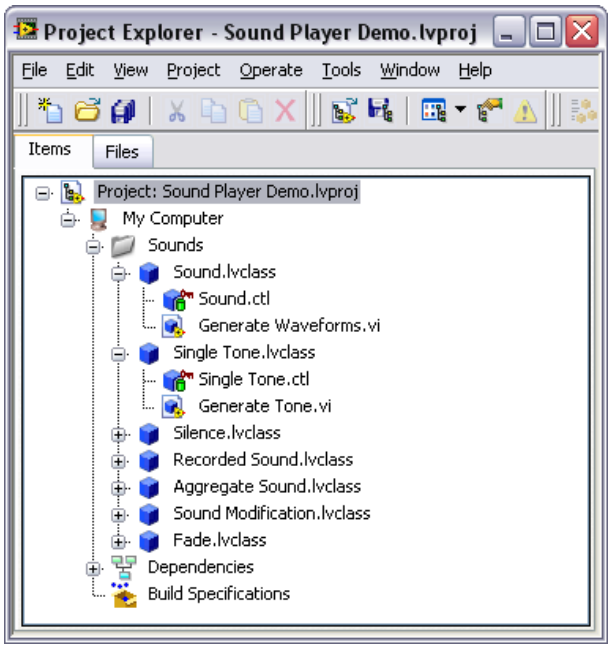


Figure 3-8. Sound Player Demo Project with All Sounds

2. In the Project Explorer window, select **File»Save All** to save the project and all of the classes that have been added to it.

End of Exercise 3-3

Exercise 3-4 Using Dynamic Dispatch Methods

Goal

Dynamically dispatch the Generate Waveforms method defined in `Sound.lvclass`. That method should be overridden by each of its children.

Scenario

In Exercise 3-1, you created `Generate Waveforms.vi` as a public method of `Sound.lvclass`. You left this method mostly empty because the implementation varies depending on the type of sound used.

You now use dynamic dispatch to override the `Sound.lvclass` implementation of this method in `Single Tone.lvclass`.

Design

- `Generate Waveforms.vi`—This method uses the Frequency and Duration data of a `Single Tone.lvclass` input to create a sinusoidal waveform for the single tone.

Implementation

1. If it is not already open, open `Sound Player Demo.lvproj`.
2. Open `Sound.lvclass:Generate Waveforms.vi`.
3. Modify the connector pane of `Generate Waveforms.vi` so that the **Sound.lvclass** input is set as **Dynamic Dispatch Input (Required)**.
4. Modify `Generate Waveforms.vi` to allow reentrant execution. You do this so classes that use recursion (Aggregate Sound, Sound Modification, Fade) can recursively call this method. You will share clones between executing instances of this method in order to reduce the overall memory usage.
   - ❑ Select **File»VI Properties**.
   - ❑ In the VI Properties dialog box, select **Execution** from the Category list.
   - ❑ Place a checkmark in the **Reentrant execution** checkbox.
   - ❑ Select **Share clones between instances (reduces memory usage)**.
   - ❑ Click **OK** in the VI Properties dialog box.

- ❑ A prompt appears, asking if you would like to make all the implementations match priority, execution system, and reentrancy of this VI. Click **Yes**. This causes this change to propagate through to the existing dynamic dispatch overrides created for the classes that you added to this project in Exercise 3-3.

5. Save and close `Generate Waveforms.vi`.

6. Configure `Sound.lvclass` so that its children must create their own implementations of `Generate Waveforms.vi`.

   - ❑ Right-click **Sound.lvclass** and select **Properties**.

   - ❑ In the Class Properties dialog box, select **Item Settings** from the Category list.

   - ❑ `Generate Waveforms.vi` is already selected for you under **Contents**. Place a checkmark in the **Require descendant classes to override this dynamic dispatch VI** checkbox.

   - ❑ Click **OK**.

     A prompt displays, asking if you want to make all the implementations match the access scope of this implementation. Click **Yes**.

7. Open `Single Tone.lvclass:Generate Tone.vi`. Notice that the Run arrow is broken. Click the Run arrow for an explanation of why the VI is broken.

   - ❑ In the Error list dialog box, select `Single Tone.lvclass from the` Items with errors list.

   - ❑ The explanation of the error indicates that `Single Tone.lvclass` "does not implement at least one dynamic dispatch member VI marked in an ancestor class as a required override." This error has occurred because you just marked `Generate Waveforms.vi` as a method that must be overridden in each child class of `Sound.lvclass` and we have not yet created a dispatch method for `Single Tone.lvclass`.

   - ❑ Close `Single Tone.lvclass:Generate Tone.vi`.

8. Create a dynamic dispatch method for `Single Tone.lvclass` that overrides `Sound.lvclass:Generate Waveforms.vi`.

   ❑ Right-click **Single Tone.lvclass** and select **New»VI for Override**. This creates a VI from a pre-determined template. You will modify this VI to suit your needs.

   ❑ In the New Override dialog box, select *Generate Waveforms.vi and click **OK**.

   ❑ Save the VI as `Generate Waveforms.vi` in `<Exercises>\...\Sound Player Demo Project\Single Tone`.

9. Modify the block diagram of `Single Tone.lvclass:Generate Waveforms.vi` as shown in Figure 3-9.

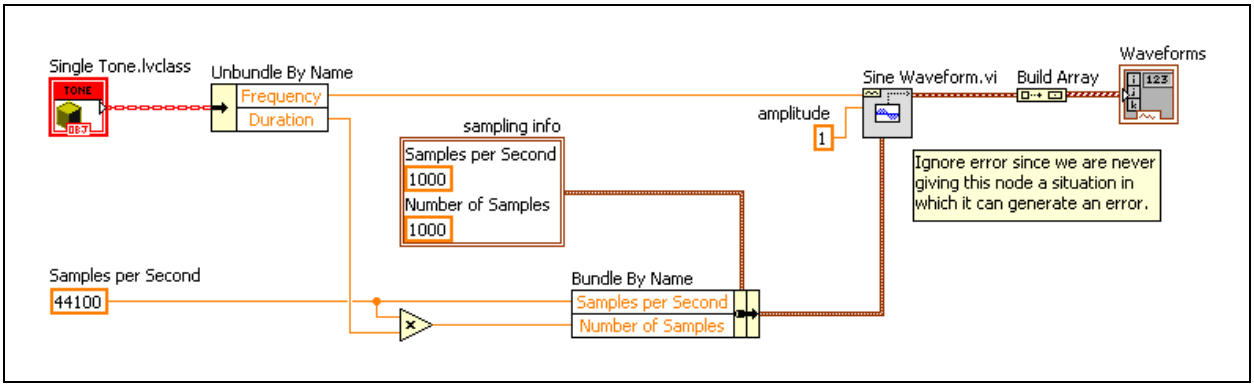


Figure 3-9. Single Tone.lvclass:Generate Waveforms.vi Block Diagram

   ❑ The **sampling info** cluster can be created from the input to **Sine Waveform.vi**. The contents of the cluster were renamed for clarity.

Note `Single Tone.lvclass:Generate Waveforms.vi` does not perform any error checking on `Sine Waveform.vi` because it is the responsibility of the method that acquires the sound data (in this case Frequency and Duration) to check that the values acquired are valid.

10. Modify the front panel of `Single Tone.lvclass:Generate Waveforms.vi` as shown in Figure 3-10.

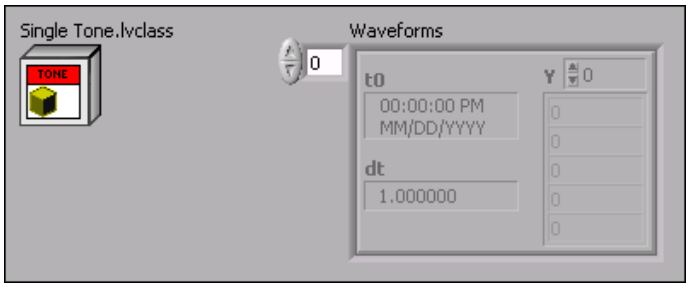


Figure 3-10. Single Tone.lvclass:Generate Waveforms.vi Front Panel

11. Modify the icon and connector pane as shown in Figure 3-11.

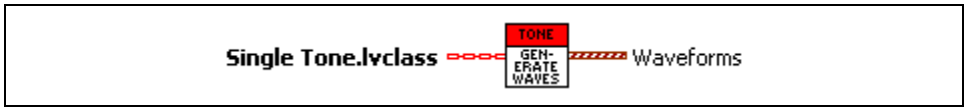


Figure 3-11. Single Tone.lvclass:Generate Waveforms.vi Icon and Connector Pane

Note The connector pane *must* match the connector pane that was created for `Sound.lvclass:Generate Waveforms.vi`. Otherwise the dynamic dispatch VI will be broken.

12. Document the purpose of this method.

    - ❑ Select **File»VI Properties**.
    - ❑ In the VI Properties dialog box, select **Documentation** from the Category list.
    - ❑ In the VI Description text box, describe the purpose of this method. For example: `This method creates a sine wave output based on the Frequency and Duration values that were acquired by Generate Tone.vi.`

13. Set `Single Tone:Generate Waveforms.vi` to allow reentrant execution.

    - ❑ In the VI Properties dialog box, select **Execution** from the Category list.
    - ❑ Place a checkmark in the **Reentrant execution** checkbox.
    - ❑ Select **Share clones between instances (reduces memory usage)**.

- ❑ Click **OK** in the VI Properties dialog.
- ❑ Notice that the Run arrow for `Single Tone.lvclass:Generate Waveforms.vi` is no longer broken. In this method, you have now fulfilled all of the promises made by the parent class implementation of the dynamic dispatch method.

14. Save and close `Single Tone:Generate Waveforms.vi`.

Another developer has created much of the remaining code for the sound player demo project. You need to add some finishing touches, but first you must add the code to your project.

15. Add the Application-level classes created by the other developers on your team to the project.
    - ❑ In Windows Explorer, navigate to `<Exercises>\...\Sound Player Demo Project\Application`.
    - ❑ Click `Application.lvclass` and drag it into **My Computer** in your project.
    - ❑ Repeat these steps for `Sound Demo.lvclass`.
16. Create a Sound Players virtual folder in your project and add the sound player classes to it.
    - ❑ Right-click **My Computer** and select **New»Virtual Folder**. Name the virtual folder `Sound Players`.
    - ❑ In Windows Explorer, navigate to `<Exercises>\...\Sound Player Demo Project\Sound Player`.
    - ❑ Click `Sound Player.lvclass` and drag it into the **Sound Players** virtual folder in your project.
    - ❑ Repeat the previous two steps for Sound Card and Sound Visualizer.

❑ Rearrange the classes in the Sound Players virtual folder so that your Project Explorer window resembles Figure 3-12.

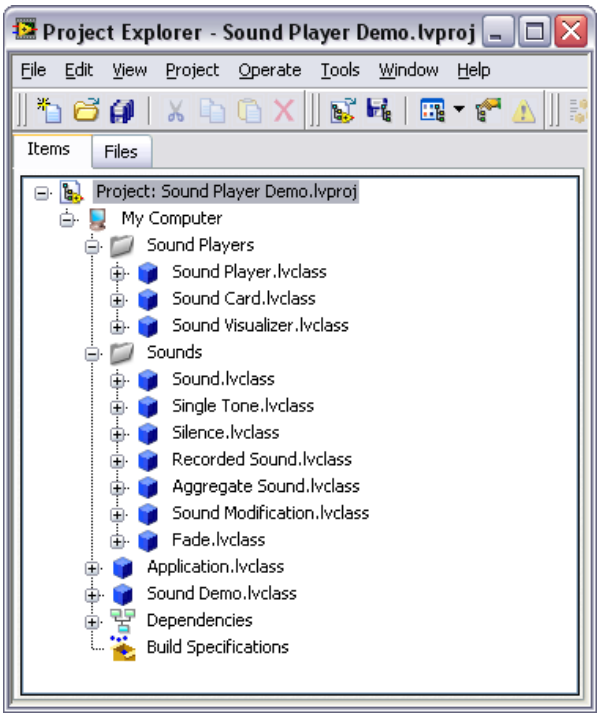


Figure 3-12. Sound Player Demo Project with All Classes

17. In the Project Explorer window, select **Save All** to save the project and all of the classes that have been added to it.

Challenge

You have spent quite a bit of time coding at this point and may have become concerned about whether or not the class hierarchy that you have developed for Sound and its children will dispatch the Generate Waveforms method properly.

Create a VI that builds an array of Sounds using Generate Tone, Generate Silence, and so on, and then dynamically dispatches `Generate Waveforms.vi` to create a single waveform graph that contains the sounds that you configured.

Use Highlight Execution to verify that the dispatch method works properly.

Save the VI as `Dynamic Dispatch Tester.vi` in `<Exercises>\...\Sound Player Demo Project\Sound Demo Support Files`. Create a virtual folder in your project named `Sound Demo Support Files` and move the VI into that folder.

End of Exercise 3-4

Exercise 3-5 LabVIEW Tools for OOP

Goal

Use LabVIEW tools to explore the class hierarchy and debug behavior of classes that were created by other developers.

Scenario

In Exercises 3-3 and 3-4, you added classes and methods to the project that were provided for you. Use some of the built-in LabVIEW tools to learn more about these classes and their methods.

Design

Use the following tools to explore the code that was provided for you:

- Find Children
- Find Callers
- VI Hierarchy Window
- Class Hierarchy Window
- Custom probe

Implementation

1. If it is not already open, open `Sound Player Demo.lvproj`.
2. Identify the descendants of a given class.
   - ❑ Right-click **Sound Modification.lvclass** in the Project Explorer window and select **Find»Children**.
   - ❑ Which classes inherit from Sound Modification?
3. Determine which VIs use a given VI on their block diagrams.
   - ❑ Right-click **Aggregate Sound.lvclass:Generate Waveforms.vi** and select **Find»Callers**.
   - ❑ Which VIs in the project call `Aggregate Sound.lvclass:Generate Waveforms.vi`?

4. Determine which VIs are called by a given VI.

   - ❑ Open `Sound Card.lvclass:Play.vi` and select **View» VI Hierarchy**.
   - ❑ Which VIs are called directly by `Sound Player.lvclass:Play.vi`?

Tip Look for the VIs that have lines drawn directly to `Sound Player.lvclass:Play.vi`. Do not include all of the subVIs of those VIs.

Note The VI Hierarchy view includes all instances of dynamically dispatched methods, because any one of them could be called depending on the **Sound** input.

5. View the Class Hierarchy of a given LabVIEW class.

   - ❑ Right-click **Application.lvclass** and select **Show Class Hierarchy**.
   - ❑ Right-click **LabVIEW Object** and select **Show all Child Classes**.
   - ❑ How does the Class Hierarchy window compare to the diagram that you created in Exercise 2-2?

6. Add `Dynamic Dispatch Tester.vi` to your project.

   - ❑ If you completed the Challenge portion of Exercise 3-4, `Dynamic Dispatch Tester.vi` is already in your project. Open the VI and proceed to step 7.
   - ❑ If you did not complete the Challenge portion of Exercise 3-4, complete the following steps to add the solution VI to your project.
     - In Windows Explorer, navigate to `<Solutions>\...\Exercise 3-4 Challenge\Sound Demo Support Files`.
     - Select `Dynamic Dispatch Tester.vi` and put a copy of this VI in the following location: `<Exercises>\...\Sound Player Demo Project\Sound Demo Support Files`.
     - Create a virtual folder in your project named `Sound Demo Support Files`.
     - Select `Dynamic Dispatch Tester.vi` in your project folder and drag it into the virtual folder you just created.
     - Open this VI.

7. Use a generic probe.

   ❑ On the block diagram of `Dynamic Dispatch Tester.vi`, right-click the Sound wire inside the For Loop used to dispatch the correct implementation of `Generate Waveforms.vi` and select **Probe**.

   ❑ Turn on Highlight Execution.

   ❑ Run `Dynamic Dispatch Tester.vi`.

   Does the probe return any useful information?

8. Create a custom probe to view data that will travel on a class wire that could be Sound or any of its children.

   ❑ Open the block diagram of `Dynamic Dispatch Tester.vi`.

   ❑ Right-click the Sound wire inside the For Loop used to dispatch the correct implementation of `Generate Waveforms.vi`. Select **Custom Probe»New**.

   ❑ In the Create New Probe dialog box, select **Create a new probe**.

   ❑ Configure the Save the New Probe page of the dialog box as shown in Figure 3-13. The path for **Directory to save the probe** should reflect the location of your `Exercises` directory.

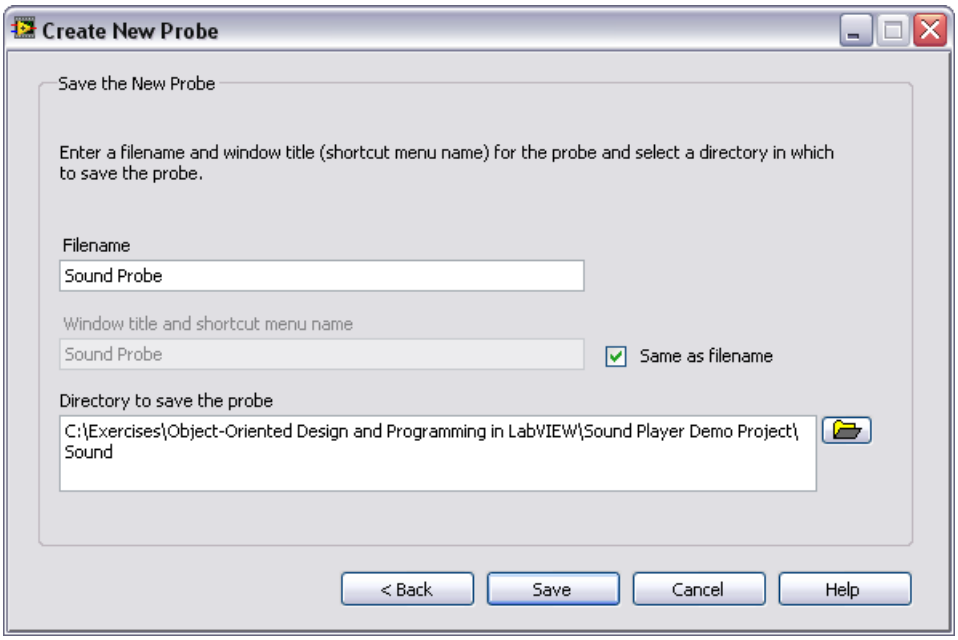


Figure 3-13. Create New Probe Dialog Box

❑ Click **Save**.

9. Modify the block diagram as shown in Figure 3-14.

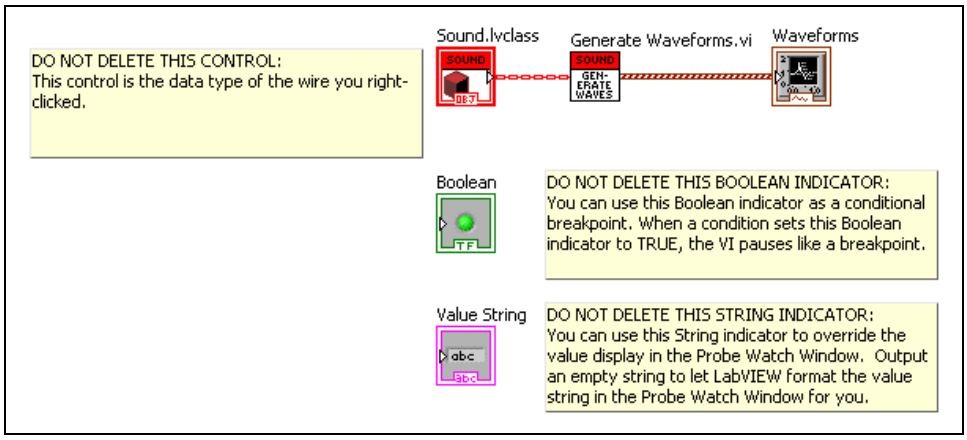


Figure 3-14. Sound Probe.vi Block Diagram

❑ You must create the waveform chart on the front panel and then wire it to the Waveforms output of `Sound.lvclass:Generate Waveforms.vi`.

10. Modify the front panel as shown in Figure 3-15.

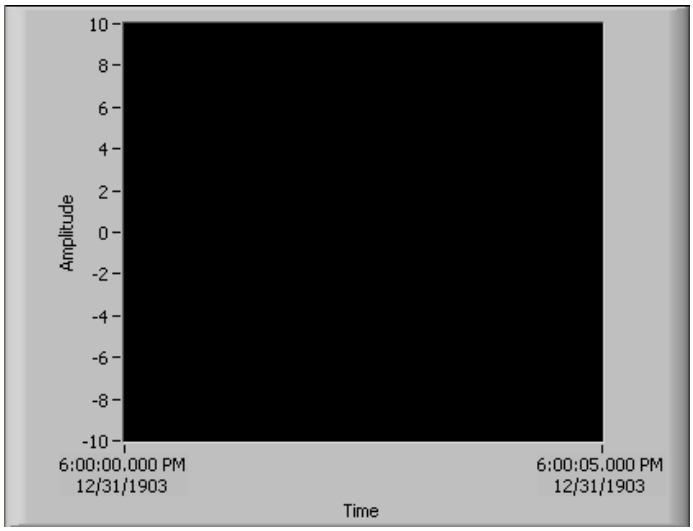


Figure 3-15. Sound Probe.vi Front Panel

Note Do not delete the `Sound.lvclass` input. You are only scaling the waveform chart to fill the front panel. `Sound.lvclass` is obscured by the chart, but it is still there.

❑ Right-click the chart and select **Visible Items»Label**.

❑ Right-click the chart and select **Visible Items»Plot Legend**.

- ❑ Right-click the chart and select **Fit Control to Pane**.
- ❑ Save and close `Sound Probe.vi`.

11. Add `Sound Probe.vi` as a private-scoped method for `Sound.lvclass`.
    - ❑ In the project, add `Sound Probe.vi` to `Sound.lvclass`. This is necessary to set this method as the default probe of the class.
    - ❑ Right-click **Sound Probe.vi** and select **Access Scope»Private**. This probe is only usable for `Sound.lvclass` and should not be called from other VIs.
12. Set `Sound Probe.vi` as the initial default probe for `Sound.lvclass` and its children.
    - ❑ Right-click `Sound.lvclass` and select **Properties**.
    - ❑ In the Class Properties dialog box, select **Probes** from the **Category** list.
    - ❑ Select `Sound.lvclass:Sound Probe.vi` from the **Default Probe** list.
    - ❑ Click **OK**.
13. Use your custom probe to observe the data carried on the Sound wire inside of the For Loop.
    - ❑ Open `Dynamic Dispatch Tester.vi`.
    - ❑ Right-click the Sound wire inside the For Loop and select **Probe**. Notice that the Probe Watch window shows the front panel of `Sound Probe.vi` as the **Probe Display**.

Note A custom probe can only be set as the default probe for the class that it is a member of. In this case, `Sound Probe.vi` was created for, and added to the generic `Sound.lvclass`. You would have to create separate custom probes for the children of `Sound.lvclass`.

- ❑ Turn on Highlight Execution.
- ❑ Run `Dynamic Dispatch Tester.vi`. Notice that each iteration of the For Loop results in a separate waveform being created and displayed in the Probe Watch window.

14. Close `Dynamic Dispatch Tester.vi`.

15. In the Project Explorer window, select **Save All** to save the project and all of the classes modifications that have been made.

End of Exercise 3-5

Notes

Notes

4

Object-Oriented Tools and Design Patterns

Exercise 4-1 Channeling Pattern

Goal

Use the channeling pattern to guarantee the execution of pre-processing and post-processing steps in your application algorithm.

Scenario

Create the top-level methods for `Application.lvclass` and `Sound Demo.lvclass`. The main VI for the project will exist in `Sound Demo.lvclass`. The overall application algorithm will be defined by `Application.lvclass`.

Design

The top-level public method of Application should be named `Run.vi`. It should guarantee execution of the following steps, in order:

1. **Setup**—This method does nothing for the abstract Application class. Child classes should override this method with any work needed to set up the main body of the execution. This includes requesting resources that will be needed for the entire duration of the execution and reading any configuration files. Child classes are not required to override Setup because there are plenty of applications that can jump straight into their main execution logic.

2. **Execute**—This method does nothing for the abstract Application class. This VI MUST be overridden by child classes. Children should place their main execution logic in this method. This method should only be called if Setup does not return any errors.

3. **Shutdown**—This method does nothing for the abstract Application class. Child classes should override this method with any work needed to quit the application. This includes releasing any resources acquired during Setup and writing any configuration files if options changed during the run of the application. This method will only be called if no error was returned from Setup and it will be called even if Execute returns an error.

4. **Handle Error**—This method is called downstream of the other methods so it can detect any error that originated in Setup, Execute, or Shutdown. It does not see any errors that were passed into Run. In other words, it only detects errors generated by this application. Child classes may override this VI to do error logging, throw error dialogs, or other appropriate actions when their application has an unclean exit. Handle Error does not have any way to restart the application. It is for error management, only.

The top-level method of `Sound Demo.lvclass`, `Main.vi`, should call into `Application.lvclass:Run.vi`, using a Sound Demo class constant to determine which implementation of Setup, Execute, and Shutdown should be called. This is the main VI of the project and is the starting point for demonstrating sound players.

Implementation

Implement Application.lvclass:Run.vi

1. Open `Sound Player Demo.lvproj`, if it is not already open.

2. Create the Run method to define the execution algorithm for Application and its children.

   - ❑ Right-click **Application.lvclass** and select **New»VI from Static Dispatch Template**.

   - ❑ Save the VI as `Run.vi` in `<Exercises>\...\Sound Player Demo Project\Application`.

3. Build the block diagram shown in Figure 4-1, using methods from `Application.lvclass`.

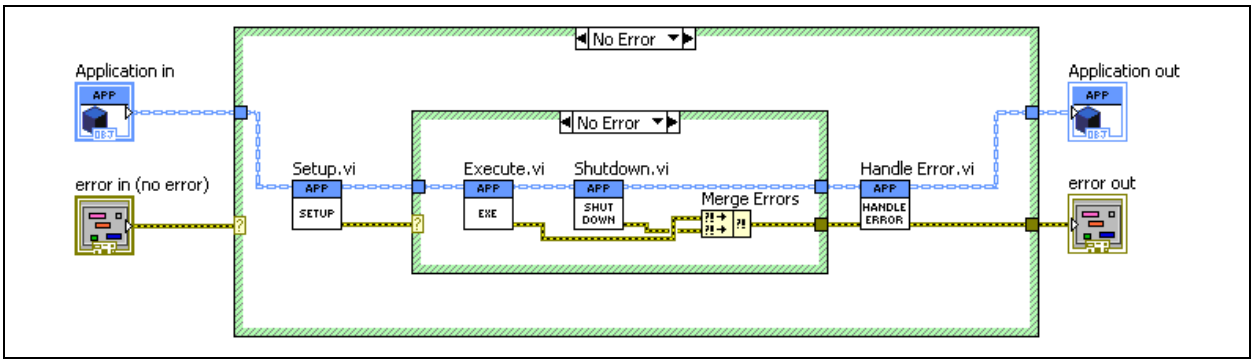


Figure 4-1. Application.lvclass:Run.vi Block Diagram

Notes Do not pass the error wire into Setup, Execute, or Shutdown because you have already guaranteed with the Error/No Error Case Structures that these method will never execute when an upstream error has occurred. By placing these structures in Run.vi, we

have removed the need for each dispatch version of Setup, Execute and Shutdown to replicate the implementation of Error/No Error Case structures.

The error wire from `Execute.vi` is wired to the top terminal of the Merge Errors function to give it priority over errors generated by `Shutdown.vi`.

The error wire is *not* wired from `Execute.vi` to `Shutdown.vi`. This is intentional, because you want to be sure that any resources allocated in `Setup.vi` are closed in `Shutdown.vi`, regardless of whether or not `Execute.vi` generated an error.

4. Resize the front panel to remove unnecessary empty space, as shown in Figure 4-2.

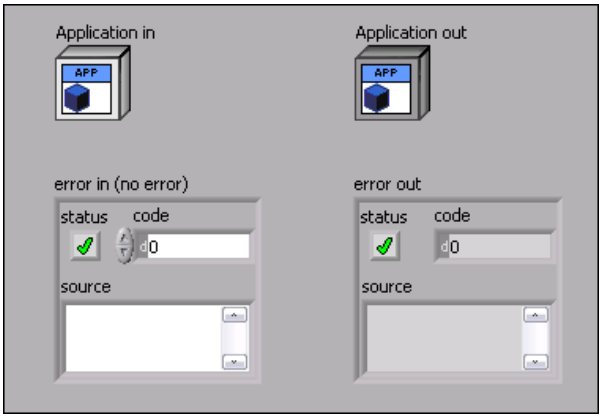


Figure 4-2. Applciation.lvclass:Run.vi Front Panel

5. Modify the icon and connector pane as shown in Figure 4-3.

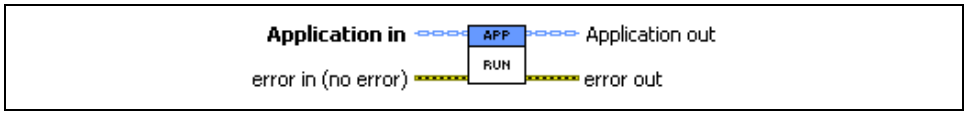


Figure 4-3. Application.lvclass:Run.vi Icon and Connector Pane

6. Document the purpose of this method.

   - ❑ Select **File»VI Properties**.
   - ❑ In the VI Properties dialog box, select **Documentation** from the Category list.
   - ❑ In the **VI Description** text box, describe the purpose of this method. For example: `This VI contains the main architecture for all Applications. It uses the channeling pattern to ensure that each implementation will follow the same basic steps: Setup, Execute, Shutdown, Handle Error.`

❑ Click **OK** to close the VI Properties dialog box.

❑ Save and close `Run.vi`.

Implement Sound Demo.lvclass:Main.vi

1. Create the main VI for the Sound Player Demo project. This VI uses the algorithm defined by `Application.lvclass:Run.vi`.

   ❑ Right-click **Sound Demo.lvclass** and select **New»VI**.

   ❑ Save the VI as `Main.vi` in `<Exercises>\...\Sound Player Demo Project\Sound Demo`.

2. Create the block diagram of `Main.vi` as shown in Figure 4-4.

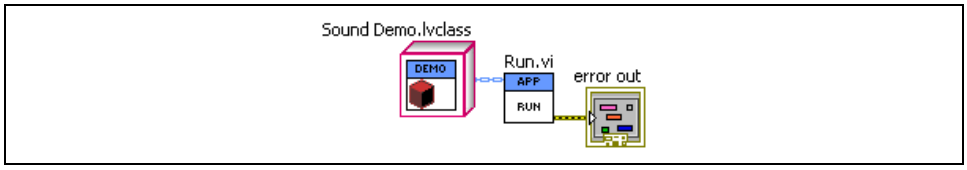


Figure 4-4. Sound Demo.lvclass:Main.vi Block Diagram

Note The `Sound Demo.lvclass` constant is passed into `Application.lvclass:Run.vi` to determine which implementations of Setup, Execute, and Shutdown to dynamically dispatch to.

3. Modify the front panel of `Main.vi` as shown in Figure 4-5.

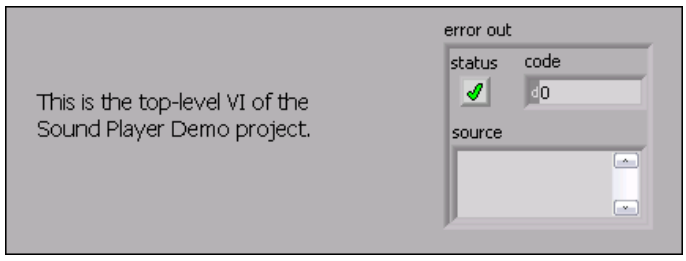


Figure 4-5. Sound Demo.lvclass:Main.vi Front Panel

4. Modify the icon and connector pane of `Main.vi` as shown in Figure 4-6.

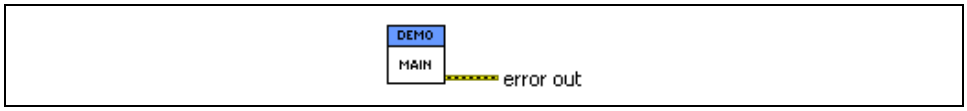


Figure 4-6. Sound Demo.lvclass:Main.vi Icon and Connector Pane

5. Document the purpose of this method.
   - ❑ Select **File»VI Properties**.
   - ❑ In the VI Properties dialog box, select **Documentation** from the Category list.
   - ❑ In the **VI Description** text box, describe the purpose of this method. For example: `This is the top-level VI that starts the Sound Player Demo application`.
   - ❑ Click **OK** to close the VI Properties dialog box.
6. Save and close the VI.

Test

At this point, you will test `Sound Demo.lvclass:Main.vi`. The application currently has most of the functionality. The only feature that remains is the ability to play sounds on either the Sound Visualizer or the Sound Card.

Currently, `Sound Demo.lvclass:Setup.vi` always plays sounds on the Sound Visualizer. You implement the ability to select either player in Exercise 4-2.

For now, verify that the application plays sounds on the Sound Visualizer.

1. In Windows Explorer, navigate to `<Exercises>\...\Sound Player Demo Project\Sound Demo Support Files`.
2. Select `sample.wav`, `Sound XML File Creator.vi`, and `sounds.xml`. Drag these three files into the **Sound Demo Support Files** virtual folder in `Sound Player Demo.lvproj`.
3. Open `Sound XML File Creator.vi`. Explore the block diagram to understand the functionality that has been implemented.
4. Run `Sound XML File Creator.vi`. This VI creates an XML file containing Sound data that will be read by `Sound Demo.lvclass:Execute.vi` and used to create the waveforms that are played.
5. After running `Sound XML File Creator.vi`, open and run `Sound Demo.lvclass:Main.vi`. After configuring the visualizer, you should see a series of waveforms play on the Visualizer.

6. Click **OK** in the Sound Demo dialog box.

7. Save and Close all VIs.

8. In the project window, select **File»Save All** to save the project and the class modifications that have been made.

End of Exercise 4-1

Exercise 4-2 Factory Pattern

Goal

Implement the factory pattern to place the appropriate child class data onto a parent class wire to execute the appropriate dynamic dispatch method.

Scenario

Modify `Sound Demo.lvclass:Setup.vi` so that it dynamically dispatches to the appropriate implementation of `Initialize With Dialog.vi` based on the type of Sound Player selected by the user.

The factory pattern is easily extended by adding additional child classes to the parent type.

Design

`Sound Demo.lvclass:Setup.vi` currently uses a `Sound Visualizer.lvclass` constant to always play sounds on the Sound Visualizer. We want to modify this code so that the user can specify either the Sound Card or the Sound Visualizer to play the sounds.

This implementation will occur in two stages:

1. Replace the `Sound Visualizer.lvclass` constant with an enumerated control wired to a case structure that will pass either a `Sound Card.lvclass` constant or a `Sound Visualizer.lvclass` constant, depending on the enumerated value selected. This will allow the demo to play sounds using either the Sound Card or the Sound Visualizer.

2. Replace the `Sound Visualizer.lvclass` and `Sound Card.lvclass` constants with code to dynamically load the appropriate child of `Sound Player.lvclass` based on the enumerated value selected. When this code is built into an executable, this will allow the Sound Player Demo to execute without having to load both classes and all of their methods into memory every time the code executes.

Implementation

Implement the Factory Pattern

1. Open `Sound Player Demo.lvproj` if it is not already open.

2. Open `Sound Demo.lvclass:Setup.vi`.

3. Modify the front panel of `Setup.vi` as shown in Figure 4-7.

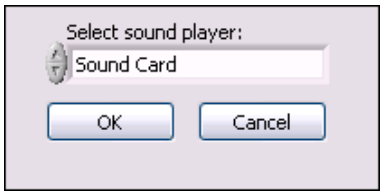


Figure 4-7. Sound Demo.lvclass:Setup.vi Front Panel

- ❑ Select sound player—This is a ring control with two values: Sound Card and Sound Visualizer.
- ❑ Configure this control as a **Strict Type Definition** and save it as `Select Sound Player.ctl` in `<Exercises>\...\Sound Player Demo Project\Sound Demo`.
- ❑ Add `Select Sound Player.ctl` to `Sound Demo.lvclass`.

4. Modify the Case Structure following the While Loop as shown in Figure 4-8 so that the structure will pass a different child of Sound Player.lvclass based on the sound player selected by the user.

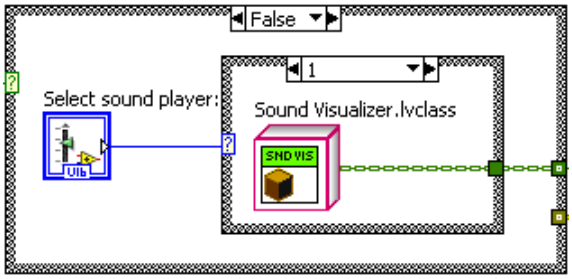


Figure 4-8. Sound Demo.lvclass:Setup.vi Block Diagram—Select Sound Player Case Structure

- ❑ Modify the `0, Default` case of the inner Case structure so that it passes `Sound Card.lvclass` in the same way that the `1` case shown in Figure 4-8 passes `Sound Visualizer.lvclass`.

5. Save and close `Setup.vi`.

6. Open and run `Sound Demo.lvclass:Main.vi` to verify that you are able to select either the visualizer or the system sound card to play the sounds. Verify that both players work.

Note If sound drivers are not installed on your system, attempting to use the Sound card results in an error.

Implement Dynamic Loading of Classes in the Factory Pattern

1. Create a new method named `Load Sound Player.vi` for `Sound Demo.lvclass`.

   ❑ Right-click **Sound Demo.lvclass** and select **New»VI**.

   ❑ Save the VI as `Load Sound Player.vi` in `<Exercises>\...\Sound Player Demo Project\Sound Demo`.

2. Modify the block diagram of as shown in Figure 4-9 so that it dynamically loads the appropriate class from disk based on which type of player the user selects.

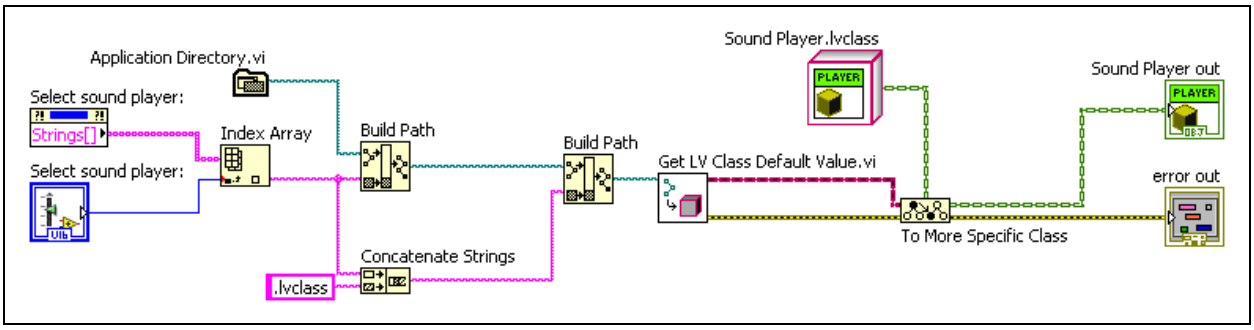


Figure 4-9. Sound Demo.lvclass:Load Sound Player.vi Block Diagram

Note This VI builds the path to the class files using the string values of the Select sound player: ring control. This removes the need for case structures and enables you to make a single change to the ring control to expand functionality to a new Sound Player.

3. Modify the front panel of `Load Sound Player.vi` as shown in Figure 4-10.

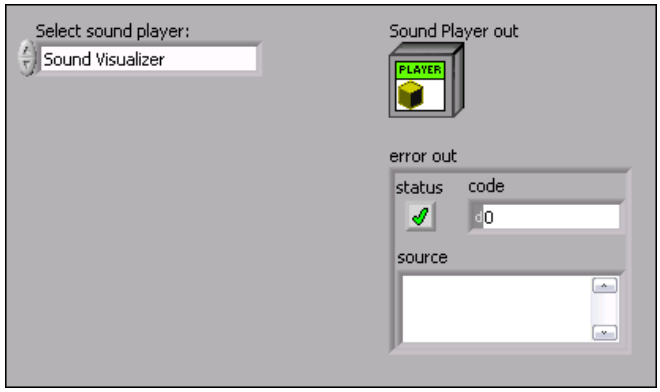


Figure 4-10. Sound Demo.lvclass:Load Sound Player.vi Front Panel

4. Modify the icon and connector pane of Load Sound Player.vi as shown in Figure 4-11.

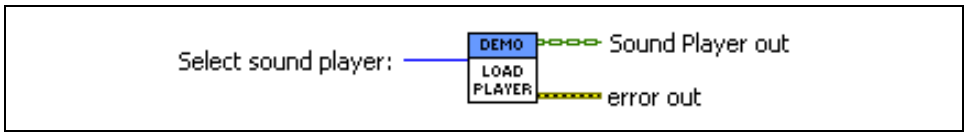


Figure 4-11. Sound Demo.lvclass:Load Sound Player.vi Icon and Connector Pane

5. Document the purpose of this method.

   - ❑ Select **File»VI Properties**.
   - ❑ In the VI Properties dialog, select **Documentation** from the Category list.
   - ❑ In the **VI Description** text box, describe the purpose of this method. For example: `Dynamically load either Sound Visualizer or Sound Card.lvclass based on user input.`
   - ❑ Click **OK** in the VI Properties dialog box.

6. Save and close `Load Sound Player.vi`.

7. Open `Sound Demo.lvclass:Setup.vi` and modify the Case structure that loads the Sound Player as shown in Figure 4-12.

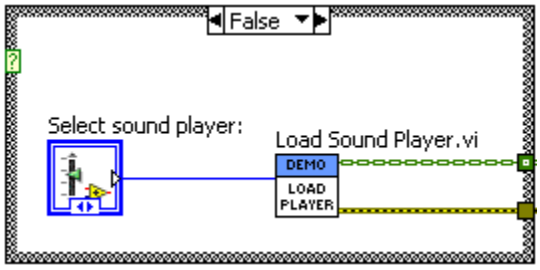


Figure 4-12. Sound Demo.lvclass:Setup.vi Block Diagram—Dynamically Loading Classes

8. Verify that classes are now dynamically loaded into memory.

   - ❑ Remove `Sound Visualizer.lvclass` and `Sound Card.lvclass` from the project. This is done so that we can build an executable that will only load the child of Sound Player into memory that is actually used.

9. In the project window, select **File»Save All** to save the project and all of the class modifications that have been made.

   - ❑ Expand **Dependencies** in the project window. Notice that `Sound Card.lvclass` and `Sound Visualizer.lvclass` are listed as Items in Memory.
   - ❑ Close and reopen `Sound Player Demo.lvproj`.
   - ❑ Expand **Dependencies** in the project window. Notice that `Sound Card.lvclass` and `Sound Visualizer.lvclass` are no longer listed.
   - ❑ Open and run `Sound Demo.lvclass:Main.vi` and select **Sound Card** as the sound player.
   - ❑ When the VI finishes, expand **Dependencies** and notice that only `Sound Card.lvclass` is listed under **Items in Memory**.

Note `Sound Card.lvclass` was loaded in memory when it was called.

   - ❑ Run `Sound Demo.lvclass:Main.vi` again, and select **Sound Visualizer** as the sound player.
   - ❑ When the VI finishes, expand **Dependencies** and notice that `Sound Visualizer.lvclass` is now also listed under **Items in Memory**.

Note You can dynamically load VIs, but you cannot dynamically unload them.

End of Exercise 4-2

Notes

5

Reviewing an Object-Oriented Application

Exercise 5-1 Building an Executable with Plug-In Classes

Goal

Build `Sound Player.lvclass:Main.vi` into an executable for deployment.

Scenario

You are ready to distribute your application to your end users. Build an executable that can be run on any station that has installed the LabVIEW Run-Time Engine.

Since your application makes use of dynamically loaded classes, you must include them in the build separately from the other classes and VIs.

Design

This application dynamically loads a number of classes in order to reduce the amount of memory required to run it when deployed as an executable. However, this technique can cause problems if you do not account for it when building your executable.

This process is broken into two stages:

1. Modify the project in preparation for being built into an executable.
2. Create a build specification and add the dynamically loaded classes and their methods for inclusion.

Implementation

Modify the Project

1. Open `Sound Player Demo.lvproj`, if it is not already open.
2. Add `Sound Card.lvclass` and `Sound Visualizer.lvclass` back into the **Sound Players** virtual folder of the project. These files need to be part of the project so you can access them from the build specification. Including them in the project does not affect the fact that they are being loaded dynamically.

3. Ensure that each child of `Sound.lvclass` will be built into the executable. These classes are not explicitly referenced anywhere in the Main VI hierarchy. Therefore, unless you add them explicitly before building the executable, they will not be included in the build and the executable will not function properly.

   ❑ Open `Sound Demo.lvclass:Execute.vi`.

   ❑ Add constants for the following classes to the block diagram.

   - `Single Tone.lvclass`
   - `Silence.lvclass`
   - `Recorded Sound.lvclass`
   - `Aggregate Sound.lvclass`
   - `Fade.lvclass`

   ❑ Save and close `Sound Demo.lvclass:Execute.vi`.

4. Modify the window appearance of `Sound Demo.lvclass:Main.vi` for use as the top-level VI of the application.

   ❑ Open `Sound Demo.lvclass:Main.vi`.

   ❑ Select **File»VI Properties**.

   ❑ In the VI Properties dialog box, select **Window Appearance** from the Category list.

   ❑ Select **Top-level application window**. This removes the scroll bars and the toolbar when the VI is running.

   ❑ Enter a meaningful name for the VI in the **Window title**.
   For example: `Sound Player Demo`.

   ❑ Click **OK** to close the VI Properties dialog box.

5. Modify the block diagram of `Sound Demo.lvclass:Main.vi` so that the window closes when the built application finishes execution, as shown in Figure 5-1.

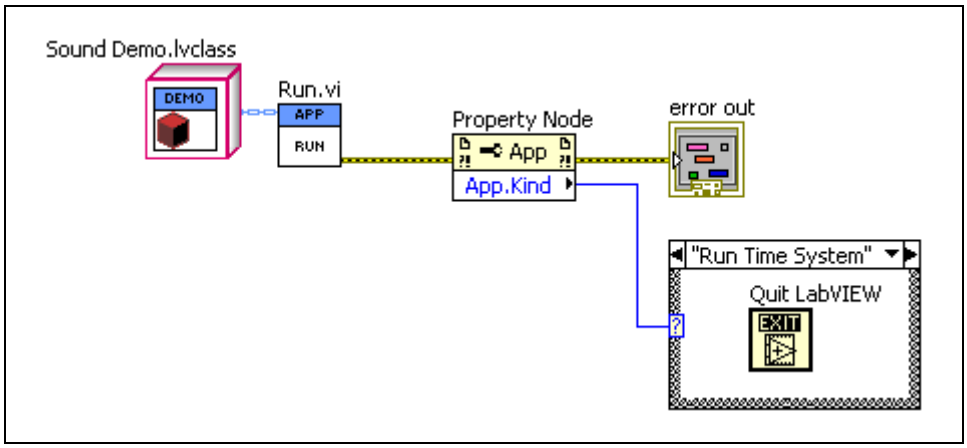


Figure 5-1. Sound Demo.lvclass:Main.vi Block Diagram—Modified for EXE

Note The App:Kind property is used to determine if the code is executing using the LabVIEW Run-Time Engine.

- ❑ Run-Time System does not appear by default when you wire the output of App:Kind to the case structure. Add a case for every value of the Case selector and modify the Run-Time System case.

6. Save and close the VI.

Create a Build Specification

1. Create a new Application build specification.

   ❑ Right-click Build Specifications and select **New»Application (EXE)**.

2. Configure the Information page of the Sound Player Demo Executable Properties dialog as shown in Figure 5-2.

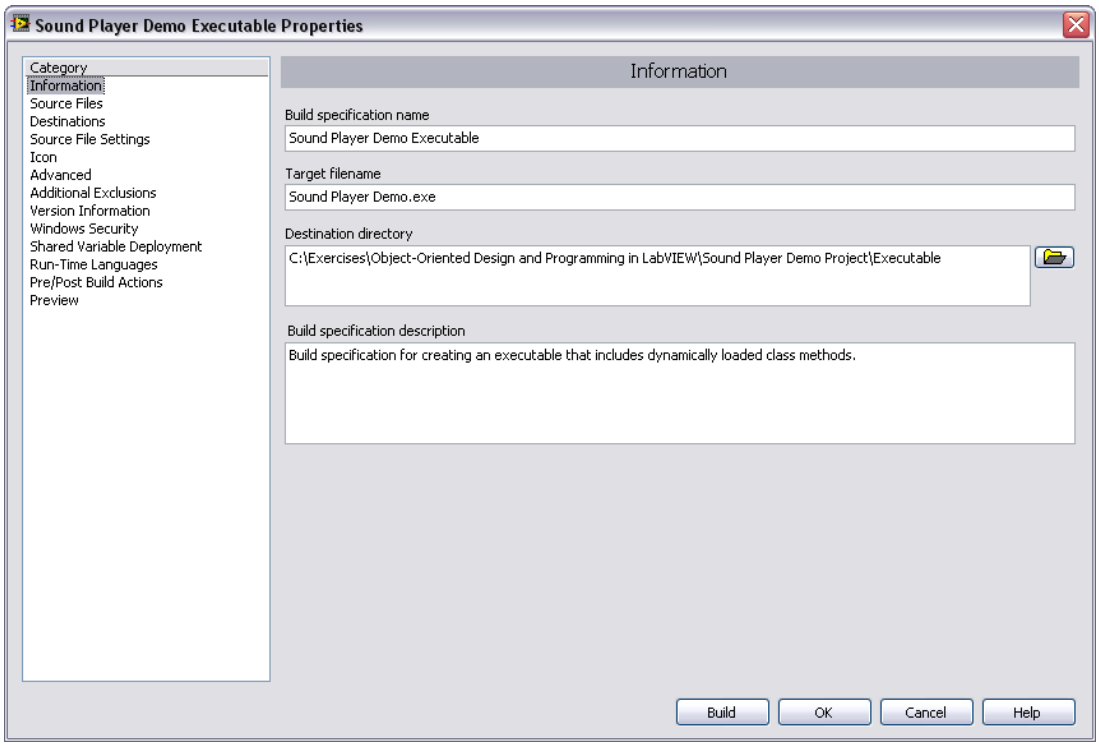


Figure 5-2. Sound Player Demo Executable Properties Dialog Box—Information Page

3. Configure the Main VI as the startup VI and any dynamically loaded files so that they will be built into the executable.

   ❑ In the Executable Properties dialog box, select Source Files from the Category list.

   ❑ Add `Sound Demo.lvclass:Main.vi` to the Startup VIs list. This step ensures that this VI will be executed as the top-level code when the executable is run.

   ❑ Add `sample.wav`, `sounds.xml`, `Sound Card.lvclass` and `Sound Visualizer.lvclass` to the Always Included list as shown in Figure 5-3. This step ensures that these files will be

included in the executable even though they are being called dynamically.

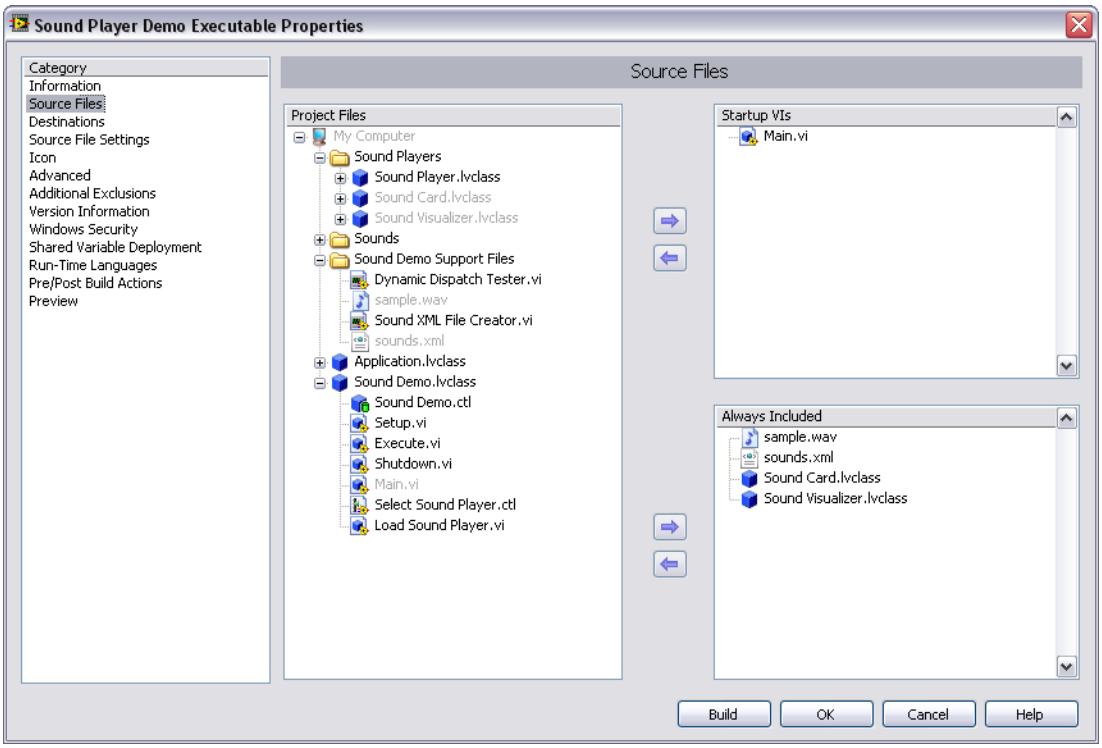


Figure 5-3. Sound Player Demo Executable Properties Dialog Box—Source Files Page

4. Create new destination paths for the files that the application dynamically loads. This must be done so that the dynamically loaded files will exist in the same relative path for the executable that they reside in for the development environment.

   ❑ In the Executable Properties dialog box, select **Destinations** from the **Category** list.

   ❑ Add the following new destination:

   – **Destination label**— `Sound Card`

   – **Destination path**—`<Exercises>\...\Sound Player Demo Project\Executable\Sound Card`

   ❑ Add the following new destination:

   – **Destination label**—`Sound Visualizer`

   – **Destination path**—`<Exercises>\...\Sound Player Demo Project\Executable\Sound Visualizer`

- ❑ Add the following new destination:
  - **Destination label**—`Sound Demo Support Files`
  - **Destination path**—`<Exercises>\...\Sound Player Demo Project\Executable\Sound Demo Support Files`
- ❑ The Destinations should now resemble Figure 5-4.

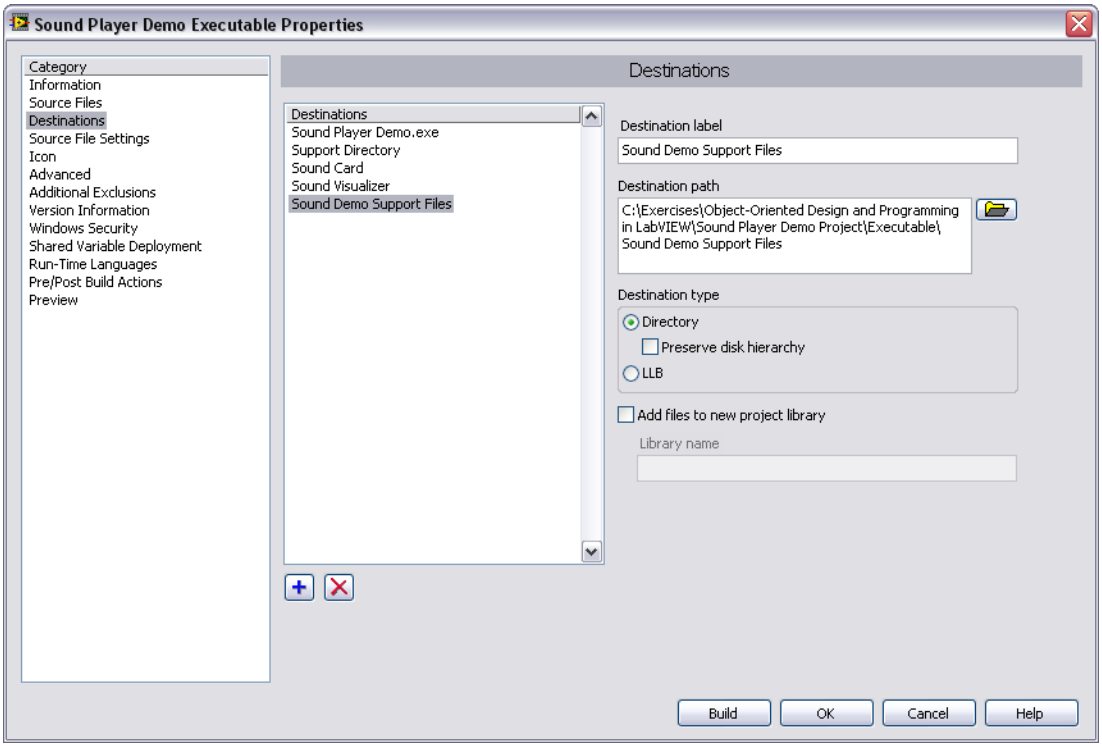


Figure 5-4. Sound Player Demo Executable Properties Dialog—Destinations Page

5. Associate each dynamically loaded file with the appropriate destination. The previous step created the correct relative paths for these files. This step copies the appropriate files to those paths.
   - ❑ In the Executable Properties dialog box, select **Source File Settings** from the **Category** list.
   - ❑ Browse through the Project Files tree. For each of the following dynamically loaded files, modify its Destination settings as shown in Table 5-1.

Table 5-1. Sound Player Demo Executable Properties—Source File Settings

Project File	Destination
`Sound Card.lvclass`	Sound Card
`Sound Visualizer.lvclass`	Sound Visualizer
`sounds.xml`	Sound Demo Support Files
`sample.wav`	Sound Demo Support Files

6. Ensure that the contents list for each project library your application uses is complete.

   - ❑ In the Executable Properties dialog box, select **Additional Exclusions** from the **Category** list.
   - ❑ Remove the checkmark from the **Modify project library file after removing unused members** checkbox.

Note The options on this page are used to minimize the size or build speed of the application. Removing the unused members of project libraries ensures that any unused code is not included in the application. Because you are dynamically loading classes that may call into the same libraries, you do not want to modify the library file after you remove unused members.

7. Click **OK** to save your changes to the build specification.
8. In the project window, right-click **Sound Player Demo Executable** and select **Build**.
9. Save and close `Sound Player Demo.lvproj`.
10. Close LabVIEW.
11. Navigate to `<Exercises>\...\Sound Player Demo Project\Executable`. Run `Sound Player Demo.exe`.
12. Verify the full functionality of the application.

End of Exercise 5-1

Notes