Chapter 2 Project Planning

Framework development is a complex activity requiring thousands of decisions. Taking the time to formulate your expectations and plan your development efforts is essential to successfully completing a framework. In this chapter, I'll highlight the key documents that will help you organize and plan your framework.

A good project manager uses several tools during the analysis and development stages of any software development project. A framework is probably the most complicated category of software to develop; developing a solid implementation plan is critical to successfully completing your framework.

A good implementation plan is actually a compilation of several documents. These documents include, but are not limited to, a steering document, requirements list, design plan, implementation plan, and a test plan.

Thoroughly analyzing your requirements and developing a detailed implementation plan will highlight areas where flexibility is needed or potential conflicts exist. You may find requirements that seem to be at odds, yet each seems equally important. Developing a requirements list takes time. However, identifying these "trouble spots" in the analysis or design phase is much more productive than after you have spent hours writing code.

In this chapter, you will learn what items go into a implementation plan and how to prepare one. You will see an explanation of the items contained in the implementation plan. In addition, you will see a sample of the requirements list, design plan, task list, and test plan used to develop the MyFrame framework. The sample illustrates the level of detail that you should strive for when preparing for your own development effort. As you read through the chapters, you will see that the presentation resembles the Requirement, Design, Plan, and Test paradigm presented in this chapter.

Steering document

The steering document is the starting point for designing your framework. It consists of a goal statement, as well as guidelines and objectives, and is the basis for preparing the remainder of your implementation plan. It should be "high level," defining the purpose for developing the framework and some overall attributes about the completed project as a whole.

In this section, I'll review the steering document used to develop MyFrame. The steering document contains a goal statement, guidelines for development, and objectives for developing the framework. The contents of the document are shown in **bold text**, while comments about the document are shown in regular text.

The goal statement

The goal statement should capture the reason you are developing your own framework. It should be singular in purpose and easily understood. Avoid putting too much detail or specific requirements into the goal statement. Many things will change as you progress through the development process, but the goal of the framework should not change.

Developing a framework is different than developing an application. When you build an application, you know that you are building an “accounting application” or a “content management system.” When you build a framework you don’t know what types of systems will be built with it.

The goal for the framework should reflect something about the types of applications that can be built with your framework. Optionally you could describe the level of support your framework provides for building those types of applications. An example of a goal statement might be: “To provide the basic templates and tools necessary for creating air traffic control software.”

Here is the goal statement for MyFrame:

The goal of MyFrame is to provide the necessary components for producing fully functional client/server or LAN-based applications.

This statement is clear and unlikely to change during the course of development. It conveys to the developer the types of applications that can be built with this framework and that the necessary tools to build an application are included as part of the framework.

Guidelines

Guidelines are evaluative statements about the framework as a whole. They reflect how the framework is constructed, or how it is to be used. Ultimately, the success of a framework depends on how it is perceived. When preparing your guidelines, think of the characteristics you would want in a framework if you were going to purchase one.

The following guidelines for MyFrame are presented in **bold text**. Additional explanations included as part of the book are in plain text.

Developer Friendly

Information should be conveyed to the developer about the how the MyFrame framework works and where to place application code.

A framework is effectively useless without documentation or guidelines for how to use it. Introduce the developer to your framework with an overview of the framework’s structure and some examples of how to use the framework. If you are building a framework for your own use, sketching out these documents or examples will help to further clarify many of your design decisions and development tasks.

Once a developer begins working with the framework, builders or wizards can be used to help with the learning process. Builders and wizards help in a number of ways. First, they inform the developer about the types of information required to make a class or group of classes functional. Additionally, they can validate the developer’s selections (or prevent them from making incorrect entries), which can reduce the overall “frustration factor” associated with learning a new framework.

ASSERTS can be used to inform the developer when the expected conditions for a program are not satisfied. In addition to suspending execution (when SET ASSERTS is ON), the message in an ASSERT can be used to explain why a particular piece of information is required.

One of the key roles fulfilled by a framework is that the complexity or redundancy of application development is hidden from the end developer. As a framework developer, you provide the application infrastructure. You should provide developers with ways to customize and utilize your infrastructure. Make sure the areas where the developers are to place code are clearly identifiable. Strive for a “fill-in-the-blanks” or “connect-the-dots” approach.

These are just some of the ways to make your framework developer friendly. Remember that this is a guideline. As you develop your framework you’ll be in a better position to assess how a piece of code or a feature can be made developer friendly.

Consistent

MyFrame should be constructed in an orderly and organized fashion.

Developers should expect to find similar problems handled in similar ways, even if that developer is you. Identify when you choose not to follow standards, and why. Developers should not have to remember the exceptions to your rules. For example, if you decide to provide a SetCaption() method for all your forms, adhere to that. Don’t set the caption directly with code in some places and use SetCaption() in others.

Maintaining consistency throughout the framework is essential to ensure that the developer does not get frustrated with it.

Reusable (Single Point Accountability)

The framework functionality in MyFrame should be implemented once. Once implemented, the implementation should be used wherever that functionality is required.

This guideline is similar to the consistency principle.

One of the biggest benefits attributed to object orientation is that it facilitates code reuse. The promotion generally goes something like, “Once you program that pesky function, you’ll never have to code it again.” True. The bigger benefit, in my opinion, occurs when requests are made for changes. Knowing where to make the change, and having the change automatically ripple through the entire application, is a powerful feature for users of your framework.

Flexible and Configurable

The developer must be able to extend the classes and functionality provided in MyFrame.

You cannot create a framework that will handle every need of every developer. Instead, you should consider where flexibility may be required and give the developer a way to extend the framework to meet his or her own needs.

There are several means to provide flexibility. The most obvious may be to provide a property that can be set to alter the behavior of a class. For example, you can instruct a form to act as a modal form by changing the WindowType property. However, as you’ll see later, many other alternatives exist.

With flexibility comes complexity, which can be a major deterrence to reuse. When simple tasks seem to take an inordinate amount of effort or require the configuration of many different components, the tendency is to avoid using those components. If the framework is considered overly complex, developers will feel frustrated.

Always consider flexibility from two perspectives: yours as the developer of the framework, and the developers who will use your framework. It is entirely appropriate, and likely, that you will write complex code in order to make the job of subsequent developers simpler.

In no case, however, does flexibility mean that you explain how a developer can modify your framework by altering framework code. Instead, you want to provide alternatives and hooks to extend the classes you provide.

Reduce Application Development Time

MyFrame should reduce the effort required to produce a quality application.

Be careful not to micromanage this guideline. Sure, if a form takes 100 percent longer to produce, you may want to rework its structure. Conversely, if using special tags for varying types of comments requires 5 percent more time, but returns 500 percent during the project documentation phase, developers will most likely see this as a positive use of their time.

Objectives

Objectives are high-level requirements for the framework. The term "high level" implies that you cannot map an objective to one element in the framework. Instead, objectives permeate the entire framework.

Be careful not to confuse objectives with requirements. Having the ability to work with arrays is not an objective of the framework. It may be a requirement of the framework, but it is certainly not an objective. You are not building an entire application framework so that you can work with arrays.

State the objectives in a way that does not define how the objective is achieved. For example, the first objective is that the framework must work with multiple sources of data. It does *not* state:

> "The framework should contain one class that is mapped to a particular type of data. This class will serve as the interface between each type of data. A new class will be created for each new source of data."

This explanation is actually a design decision (and the way that multiple data sources are handled in MyFrame). Working with multiple data types affects much more than one individual class; it affects how you navigate between records, how you write data validation code, and a host of other issues, as you will see throughout the remainder of the book.

The following objectives for MyFrame are presented in **bold text**. Additional explanations included as part of the book are in plain text.

Multiple Data Sources

The system must have the ability to work with multiple sources of data.

Data is not limited to DBF files only. Application data can exist in remote databases accessible via ODBC, Web Services, Microsoft Outlook, or a variety of other sources.

What does it mean to be able to work with multiple sources of data? Does it mean that a single form can write to data in a SQL Server database as well as a FoxPro table? Can contact information stored in a FoxPro table be connected to Outlook appointments? Can a form originally written using a FoxPro table be switched to SQL Server without having to redevelop the form?

To me, working with multiple data types means all these things. This is not to say that the framework will accomplish all these things "automatically." It means that the tools are provided to accomplish these things. If the application developer is willing to change the way he or she thinks about designing an application, these questions can be answered affirmatively.

Multiple projects

I am always working on several projects at the same time, and each project is at various stages of completion. Although my framework doesn't change much anymore, when I do make a change it is usually something that would be beneficial to all projects in development.

There is nothing more frustrating than working on several projects at once and having to add new features to each project individually. One key principle of inheritance is that that you should only have to make code changes in one place. Having multiple copies of the same files on disk just violates this basic concept.

The term "project" here refers to each application being developed. Do not confuse the term "project" with the term "project manager." A project manager is (in FoxPro) a tool that organizes the files in your project and facilitates the compilation and distribution process.

Modular Construction

Whenever possible, MyFrame should be constructed in modular fashion.

Modular construction means that your framework and resulting systems are structured in logical pieces. Much of framework construction is the integration of components. However, once they're integrated, modifying one or more framework modules can change the behavior of the entire application.

FoxPro Designers

The MyFrame framework should work within the FoxPro Design Tools.

The objectives stated to this point are ones that you will most likely use as the basis for your own framework. Working within the FoxPro designers is a stated objective of the MyFrame framework for the following reasons:

- Developers are familiar with the FoxPro interface.
- The FoxPro interface has been user tested.
- The FoxPro interface is supported by the FoxPro documentation.

This is not to say that your framework must work exclusively with the FoxPro designers. In fact, many frameworks come with tools that either overcome the limitations of the FoxPro designers, or that support their specific design philosophy, or both.

Portable

The MyFrame framework and framework files should start in any directory.

I use the phrase "start in any directory" to indicate that whether the framework is loaded on your D:\ drive, or C:\drive, or any other directory, it should work without fail.

Implementation plan

There are no hard and fast rules as to what belongs in an implementation plan. In my experience, the most helpful project implementation plans include a detailed listing of framework requirements, design guidelines to direct the development process, a task list detailing specific development activities, and a test plan to assess the success of the project.

Each element of the implementation plan may be developed independently. In practice, however, the development of a comprehensive implementation plan is an iterative process. In this section, I will explain the type of things that go into each section and provide samples of the implementation plan for MyFrame.

Requirements list

The requirements section contains a detailed listing of features and capabilities the framework must have. Requirements are quantifiable. They can be measured. At the end of the day, either a requirement is satisfied, or it isn't.

State requirements clearly and in a quantifiable manner. Precisely defining your requirements reduces the chance for misinterpretations. Stating requirements in a manner that can be measured further reinforces the requirement and assists when developing a test plan.

Design (implementation)

The requirements list will probably resemble a "laundry list" of items to be included in the framework. Taken in isolation, each requirement could (possibly) be implemented in a variety of ways. For example, one requirement may be that users must be able to sort a particular result set. In isolation, that could mean that the result set could be sorted in memory using an array or persisted by storing the result set in a table.

In the design section of the implementation plan, you categorize one or more requirements and define a cohesive development strategy to satisfy those requirements.

Once you've completed your design, check the solution against the guidelines and objectives for the project. Ask yourself, "Does my design violate any of the objectives or requirements of the framework?"

Consider the following guideline and requirement:

- Guideline: The framework should be user friendly.
- Requirement: Each action taken in the system must be reversible or confirmed prior to taking the action.

A system that asks a user, "Are you sure you want to log on to the system?" after he enters his user name and password and clicks a Logon button, satisfies the requirement but probably fails the "user friendly" test. The user took the time to provide a user name and password, and click the Logon button, so it's safe to assume the user wanted to log on to the system. The additional confirmation is an extra step that the user may consider "unfriendly."

Task list

This is a list of the specific development actions to be taken to construct the framework. Your requirements list and design guidelines are the basis for the items appearing in your task list.

Test plan

The test plan is a list of tests that, when run successfully, ensure that your framework is satisfying its design specifications.

Sample (excerpt)

This section contains an excerpt taken from the MyFrame implementation plan. The level of detail in your implementation plan will vary. A rule of thumb might be that the more people working on the project, the greater level of detail is required—especially if you are working on a large project where several different classes and modules have to integrate with each other.

The familiarity of your developers with the project at hand is also a factor in how detailed each item might be. The more familiar, the less detail and more "bullet-like" your implementation plan will be.

Finally, development standards will also limit the level of detail required in your implementation plan. For example, if a development step is labeled "Develop customer maintenance screen" and your development standards dictate that all maintenance screens must have Save and Cancel buttons, it is unnecessary to repeat these items for every maintenance screen in your system.

The following excerpt is related to creating the startup routine. It assumes you are familiar with application development and is presented in a terse bullet format. If you are unfamiliar with any of the items presented in the example, relax … a full explanation of each item is presented in Chapter 5, "Beginning Development."

REQUIREMENTS

- One file starts all applications.
- Starting the application also "starts" the framework.
- Use relative paths only.
- Start of application configures environment.
- Close of application restores environment.

DESIGN

One of the overall objectives for the project is to support multiple project development. One class serves as the "control" for starting and shutting down an application. This class is then subclassed as each new project is started.

A single file is required as the "main" file for each project. One program will be created to start each project. The stub program will contain the code for setting the default directory and search paths.

As a matter of preference, I like keeping the code for starting the application (Main.prg) close to the code responsible for handling all the activities as the program is started and closed. Therefore, the application class is created as a PRG class.

One program file is required to store the framework application object definition (Main_frame.prg) and another is required to store the application object and starting code for each application (Main.prg).

Task Plan

- Create a file for the framework class definition (.\MyFrame\FrameSource\Main_Frame.prg).

- Create the application class.
- Define the startup sequence for the application class.
- Define the shutdown sequence for the application.
- Create a file for the Application Class Definition.
- Add startup code responsible for setting default and search paths.
- Create the application-specific application class definition.

Test Plan

Test of startup and shutdown sequence
- Non-compiled
  - Check path
    - Load framework
      - (Path should now be set to the project's root directory)
      - (Environment settings: Set to application-specific settings)
    - Release framework
      - (Old path is restored)
      - (Original environment settings are restored)

Conventions

The first thing I do when I get a new piece of software is look through the menu and start poking. I expect the menu to be organized and indicative of the types of things the software can do. Assuming the software is reasonably organized, I know it won't take long to figure out how to use it, even before reading the documentation. I expect that you don't need to be persuaded about the importance of presenting an orderly and consistent user interface to the end user.

When building a framework, the application developer is your end user, and he or she expects the same degree of consistency in the interface: your code. Adhering to the following conventions is one way to ensure consistency.

Write clear, readable code. Application developers will read your code to gain understanding and insight into how they can better use your framework. Provide meaningful comments that explain what is happening and why. Avoid commenting on syntax; instead, focus on the objectives and why an action was taken, not just what the action was. Read through your code a few days after completing it. Are the variables named appropriately? Is the intent of the code clear from the comments provided or from the code itself?

This section focuses on the conventions you should consider for your framework, followed by a systematic approach to organizing the framework files.

Conventions are a set of rules describing how to format your code. For example:

- Indent case statements three spaces.
- Use CamelCase notation.

Follow the conventions recommended by Microsoft in the Visual FoxPro Help file. Most developers are familiar with or are already using these conventions. The conventions you choose to follow when developing the framework can be as loose, or as stringent, as you feel comfortable with. Establish a set of conventions for your code and follow it. You may add to or take away from the conventions as you see fit. Remember, one of your objectives is to make the development process smoother. Try not to make the conventions too restrictive or complicated. And finally, conventions are optional. Never impose conventions on the application developer.

Comments

Every piece of code should clearly state its intention. I have read somewhere that each piece of code should be documented in such a way that, if the code were removed, the reader would still understand its purpose. Taken in moderation, this is a good principle to follow. However, the best-case scenario is when the code is self-documenting. For an example of self-documenting code, see the LoadApp() method in Chapter 5, "Beginning Development," under the section titled "aApplication."

Place the comments in appropriate places. Consider a method named PrintSalaryHistory(). How PrintSalaryHistory() retrieves and calculates information is irrelevant to the calling program. Comments for PrintSalaryHistory() are better placed directly in the PrintSalaryHistory() method. On the other hand, if a precondition exists it would be better to state as much in both the calling method and the called method. An example of a precondition might be "Prior to calling PrintSalaryHistory(), the employee table must be open, selected, and on the record of the employee you wish to print."

Preconditions that must be explained in more than one place may be an indication that coupling between classes or methods exists.

Be careful not to over-comment. It is difficult enough to trace through code without having to read every thought a developer had while he or she assembled the code. The following is an example of over-commenting:

```
*-- The customers table is not opened when the form loads.
*-- It is only opened if the user tries to search for a customer.
*-- Since this form is one of the few forms where a private data session is
*-- not used and the customers table was not opened from the data environment,
*-- it is sometimes left open.
*-- Even though the table is opened in shared mode, and causes no problems if
*-- it remains open, it's cleaner to make sure it is closed.
*-- By the way, this next line of code I found in the FoxPro Advisor's tips
*-- section. Boy, it sure seems easier. I used to close tables like this:
*--  IF USED("TableName")
*--      USE IN "TableName"
*--  ENDIF
*--  Now I close them like this:
USE IN SELECT ("Customers")
```

Instead try:

```
*-- Cleanup
USE IN SELECT ("Customers")
```

The same is true of code that has been extensively modified over time. Obtaining a clear understanding of what code achieves is often muddied by comments about what the code used to achieve. Keep the comments in your code related to the code in effect. Place modification histories at the beginning or end of each code block; then, if the modification is of interest, the information is still readily available.

Comment blocks of code rather than individual lines. I use the term "block of code" to indicate functions, programs, or, in the case of classes, the class itself. A class may contain several methods that, in effect, function to achieve a single goal. Commenting each method is tedious and is often not as helpful as a single explanation defining the purpose of the class and role of the more important methods. A technique first introduced by Drew Speedie (a framework developer) is to include a method in each class named "zReadMe()". zReadMe() will always sort toward the bottom of an alphabetically sorted list and is therefore easily found in the FoxPro designers.

Here's an example of a comment block in a header:

```
*==========================================
*| Purpose......A sentence or two describing what the
*|              code does.
*@ Author.......The name of the developer
*| Created......The date the program was created
*| About........This is a free-form area provided to expand
*)              on the purpose of the code, explain how
*)              the code is structured, or any other
*)              relevant information.
*| Dependencies: Another free-form area describing
*)               conditions expected to be true for the code
*)               to work properly. If possible, try stating
*)               dependencies as ASSERTions.
*//Mod.List.....A listing of the dates, developer(s) and
*)              reasons for the modifications
*==========================================
```

You may have noticed that various parts of the header are marked with different comment strings. FoxPro 8.0 introduces a new tool called Code References, which searches source code files for a particular string. Using the Code References tool, it would be possible to obtain a list of all developers that worked on a project by conducting a project-wide search for "*@". Other comment tags used in MyFrame are:

```
*--Comment tags
*// Modification
*ToDo:
*-? Question:
 *) Continuation of any comment
 *--
 NOTE:
 &&
```

Names (general)

When naming something, try to make the name meaningful in the context for which the name will be used. UpdateTableBuffer() is an appropriate name for a method on a data class. However, Save() might be a better name for a similar method on a form. Using descriptive names for variables and methods helps to keep the code readable. IntelliSense, first introduced in FoxPro 7.0, dramatically reduces the “pain in the butt” factor associated with using long names.

Names (formatting)

Most FoxPro developers expect to see names in CamelCase, where the first letter of each word in a name is capitalized. Using CamelCase makes it easier to read names comprised of more than one word. Several examples have been used already in this chapter, including PrintSalaryHistory() and UpdateTableBuffer().

In some circumstances, a prefix is added to the name to convey additional information about the item. For instance, variables are often prefixed with two letters, the first indicating the scope of the variable and the second indicating the data type of the variable. Therefore, lcDescription would indicate that the variable “Description” was local in scope and of type character.

Most FoxPro developers are familiar with or are already using the prefixes listed in the Visual FoxPro Help file. These prefixes are listed in the following tables.

Table 1. *Prefixes indicating scope.*

Scope	Description	Example
l	Local	lnCounter
p	Private (default)	pnStatus
g	Public (global)	gnOldRecno
t	Parameter	tnRecNo

Table 2. *Prefixes indicating variable types.*

Type	Description	Example
a	Array	aMonths
c	Character	cLastName
y	Currency	yCurrentValue
d	Date	dBirthDay
t	Datetime	tLastModified
b	Double	bValue
f	Float	fInterest
l	Logical	lFlag
n	Numeric	nCounter
o	Object	oEmployee
u	Unknown	uReturnValue

Table 3. *Prefixes indicating class.*

Prefix	**Object**	**Example**
chk	CheckBox	chkReadOnly
col	Collection	colFormObjects
cbo	ComboBox	cboEnglish
cmd	CommandButton	cmdCancel
cmg	CommandGroup	cmgChoices
cnt	Container	cntMoverList
ctl	Control	ctlFileList
cad	CursorAdapter	cadInventory
<user-defined>	Custom	user-defined
dte	DataEnvironment	dteSalesForm
edt	EditBox	edtTextArea
frm	Form	frmFileOpen
frs	FormSet	frsDataEntry
grd	Grid	grdPrices
grc	Column	grcCurrentPrice
grh	Header	grhTotalInventory
hpl	HyperLink	hplHomeURL
img	Image	imgIcon
lbl	Label	lblHelpMessage
lin	Line	linVertical
lst	ListBox	lstPolicyCodes
olb	OLEBoundControl	olbObject1
ole	OLE	oleObject1
opt	OptionButton	optFrench
opg	OptionGroup	opgType
pag	Page	pagDataUpdate
pgf	PageFrame	pgfLeft
prj	ProjectHook	prjBuildAll
sep	Separator	sepToolSection1
shp	Shape	shpCircle
spn	Spinner	spnValues
txt	TextBox	txtGetText
tmr	Timer	tmrAlarm
tbr	ToolBar	tbrEditReport
xad	XMLAdapter	xadRemoteXMLData
xfd	XMLField	xfdPrices
xtb	XMLTable	xtbInventory

Variables
Variables begin with a two-letter prefix. The first letter indicates the scope of the variable. The second variable indicates the type. The name is in CamelCase.

Properties
Property names are prefixed with the “type” of information intended to be stored in the property. The name follows the CamelCase convention; for example: oApp.cAppName.

Functions
Function names are in CamelCase.

Tables
Avoid using long names for tables or including spaces in the name. Yes, FoxPro supports long table names and spaces, but some other databases do not. Some of the functionality you provide in the framework will rely on persisted data stored in tables. Don’t set yourself up for disaster. Using short names will prevent problems when porting to another database.

Another convention followed in MyFrame is to name the tables in the plural form and to provide one surrogate primary key for each table. An example of table names would be “Contacts” or “Invoices.” The primary key is stated in the singular form such as ContactID or InvoiceID.

Fields
Avoid using long field names for the same reasons outlined when naming tables. Additionally, many FoxPro programmers prefix the name of each field in their tables with the field’s type. Avoid that convention when naming fields. Many FoxPro data types are not comparable to data types in other database products. Using a prefix to indicate the field’s type offers little benefit because the type is easily determined and probably won’t convert well to another database product.

Methods
As a framework developer you’ll use methods for two reasons. The first reason is to accomplish work. The second is to provide a place for the application developers to do work. I’ll discuss this topic in much more depth throughout the book. Here’s the important point while discussing conventions: There is a difference in how methods are used, and they each have their own conventions.

- Use CamelCase when naming methods where you do the work.
- When providing methods for application developers, do not expect them to be clairvoyant. Consistently inform them, “This is where the code goes.” Prefix all methods that are specifically provided for future developers with “On” or “Is.” Prefix methods provided to return information with “Get.” Use CamelCase for the remainder of the method name.

  For example, “IsValid()” would indicate where a developer could place some validation code. “OnSave()” would indicate a place where a developer could take action during a save. “GetEmployeeListing()” would indicate a method where information was being returned. Developer hooks beginning with “Is” or “On” must return logical values (Boolean for purists). The “Get” methods return data.

I’ll discuss data formats in Chapter 8, “Data Access.”

Class definitions

Each class serves a purpose in the framework. For example, the “Cursor” class provides services that are useful for working with FoxPro cursors. The class name reflects what the class does.

In some cases, the class will be an abstract class serving only to define the interface for subclasses. Abstract classes are prefixed with an “a.” Alternately, fully defined classes that are ready for use are prefixed with “my.”

Each application created with the MyFrame framework does not use the MyFrame classes directly. Instead, each class is subclassed at the start of a new application. These classes are application-specific and are (optionally) prefixed with one to three letters specified by the application developer. To remain consistent throughout the book, all application classes will begin with “smp,” which is short for “sample application.”

I’ll cover much more about the roles classes play in Chapter 6, “Creating a Class Library.” A full discussion of how classes are created at the start of each application is discussed in Chapter 18, “Starting New Projects.”

Classes as form properties

When classes are used on forms, the class name is prefixed by three letters indicating the class type. For example, all text boxes are prefixed with “txt,” labels with “lbl,” and so on.

Summary

This chapter illustrated how to create a goal statement that defines the purpose for building your framework, as well as development guidelines and objectives to help guide you during the development cycle.

This chapter also illustrated several documents that will help you during the implementation of your framework. These documents include, but are not limited to, a requirements list, design plan, task list, and a test plan.

Finally, this chapter explained the need for adhering to conventions when developing your framework.

Updates and corrections to this chapter can be found on Hentzenwerke’s Web site, **www.hentzenwerke.com**. Click “Catalog” and navigate to the page for this book.