UNIX Custom Application Migration Guide

Version 2.0

Volume 4: Migrate Using .NET

Published: May 2006

Contents

About This Volume

Introduction to Volume 4

Volume 1 of the *UNIX Custom Application Migration Guide* discussed how to apply the Envisioning and Planning Phases of the Microsoft® Solutions Framework (MSF) Process Model when conducting a UNIX to Microsoft Windows® migration project. This volume, Volume 4: *Migrate Using .NET*, applies the next phases in the Process Model—the Developing Phase and the Stabilizing Phase—and directs it specifically for using Microsoft .NET. This volume describes the architectural and potential coding differences between the UNIX and Windows environments using .NET and discusses various ways to implement these differences in the Windows environment using .NET. This volume addresses these potential coding differences by looking at the solution from various categories. These categories are:

- .NET interoperability.
- Process management.
- Thread management.
- Memory management.
- File management.
- Infrastructure services.
- User interface migration.
- Deployment considerations and testing activities.
- Stabilizing Phase activities.

For each of these categories, this volume:

- Describes the implementation on the .NET environment.
- Outlines options for converting the code using .NET.
- Illustrates the options with source code examples.

This information helps you in choosing the solution that is appropriate to your application. Sufficient code examples and references are provided in this volume to aid you in the migration process. You can also refer to the .NET Framework and .NET class library documentation to obtain more details on .NET.

This volume considers Microsoft Visual Studio® .NET 2003 as the integrated development environment (IDE) for developing .NET applications. Although newer technologies exist, the guide is based on best practices developed by partners and customers. As new practices establish, they will be incorporated into future releases of the guide. These latest technologies and their features are briefly described in the “Roadmap for Future Migrations” section of Chapter 2, “Operations” of Volume 5: *Deploy and Operate* of this guide.

For more information on activities in the Developing Phase as they relate to a migration project, refer to Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of this volume.

Intended Audience

The technical information in this volume is provided to support the activities undertaken during the Developing Phase of a migration project. It is intended for developers and testers who are involved in migrating UNIX code to Windows using .NET. The developers could be UNIX or Windows programmers involved in developing the solution on Windows using .NET. Using the guidance provided in this volume, a UNIX programmer will learn how to rewrite code so that it can be recompiled to run in a .NET environment, and a Windows programmer will learn how to use .NET for achieving the functionality.

Specific advantages that this volume provides the developers and testers are:

- **Developers**. Developers can learn about the various alternative methods for migrating from UNIX to Windows using .NET and how to choose the best strategy to fit their environment and the application types.
- **Testers**. Testers can gain more insight on the testing methodology that is best suited for their migration scenario. With the help of this guide they can test the application for such aspects as functionality, management, performance, and stability.

Knowledge Prerequisites

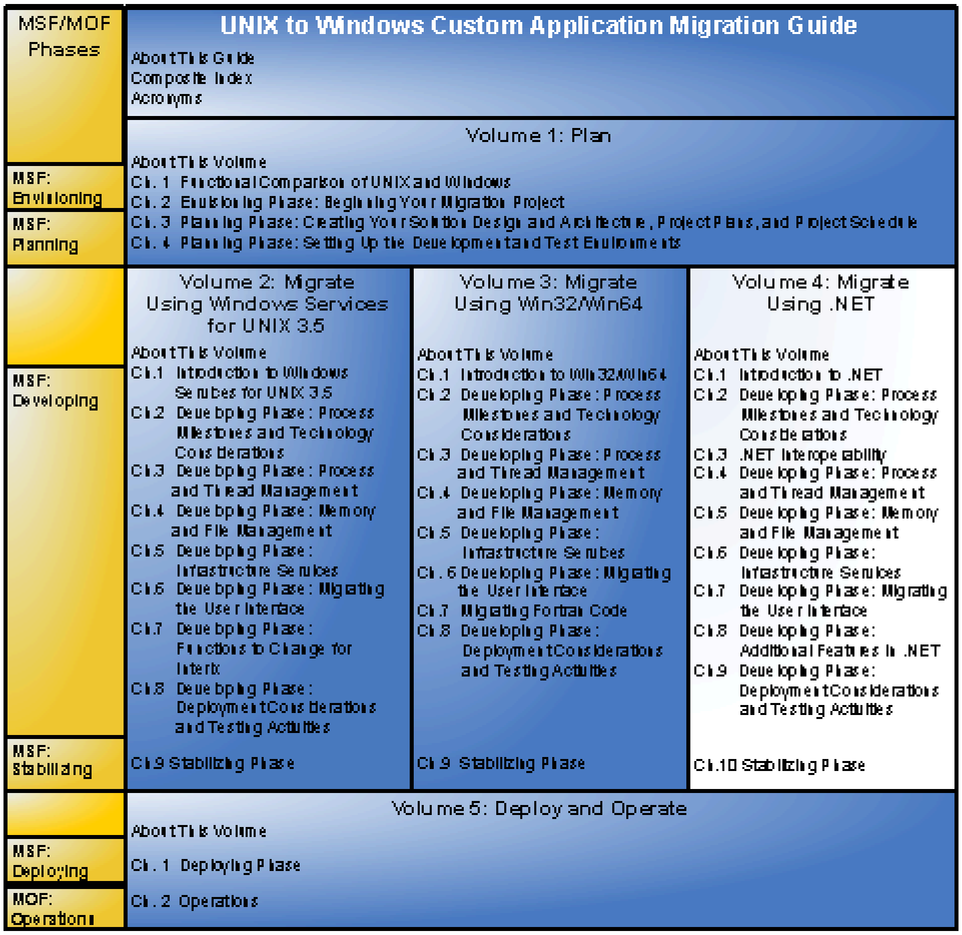
The readers of this volume should possess the following knowledge prerequisites:

- Basic knowledge of the UNIX and Windows environments.
- Basic understanding of process and thread management, file and memory management, and various infrastructure services features.
- Hands-on experience on Windows environments.
- Hands-on experience on any one of the .NET programming languages.
- Hands-on experience on using Microsoft Visual Studio .NET 2003.
- Familiarity with UNIX administration skills.

It is also recommended that you read the “About This Guide” section in Volume 1: *Plan* as well as the rest of the *Plan* volume before reading this volume.

Layout of the Guide: Volume 4

The following diagram depicts the layout of the guide and how the volumes of the guide correlate with the components of the MSF Process Model. The portion shaded in white depicts the position of the current volume in the layout of the entire guide.



MSF/MOF Phases

MSF: Envisioning

MSF: Planning

MSF: Developing

MSF: Stabilizing

MSF: Deploying

MOF: Operations

UNIX to Windows Custom Application Migration Guide
About This Guide
Composite Index
Acronyms

Volume 1: Plan
About This Volume
Ch. 1 Functional Comparison of UNIX and Windows
Ch. 2 Envisioning Phase: Beginning Your Migration Project
Ch. 3 Planning Phase: Creating Your Solution Design and Architecture, Project Plans, and Project Schedule
Ch. 4 Planning Phase: Setting Up the Development and Test Environments

Volume 2: Migrate Using Windows Services for UNIX 3.5
About This Volume
Ch.1 Introduction to Windows Services for UNIX 3.5
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch.6 Developing Phase: Migrating the User Interface
Ch.7 Developing Phase: Functions to Change for Interix
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase

Volume 3: Migrate Using Win32/Win64
About This Volume
Ch.1 Introduction to Win32/Win64
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch. 6 Developing Phase: Migrating the User Interface
Ch.7 Migrating Fortran Code
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase

Volume 4: Migrate Using .NET
About This Volume
Ch.1 Introduction to .NET
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 .NET Interoperability
Ch.4 Developing Phase: Process and Thread Management
Ch.5 Developing Phase: Memory and File Management
Ch.6 Developing Phase: Infrastructure Services
Ch.7 Developing Phase: Migrating the User Interface
Ch.8 Developing Phase: Additional Features in .NET
Ch.9 Developing Phase: Deployment Considerations and Testing Activities
Ch.10 Stabilizing Phase

Volume 5: Deploy and Operate
About This Volume
Ch. 1 Deploying Phase
Ch. 2 Operations

Figure 0.1. UCAMG organization

Organization of Content

- **About This Volume**. This chapter provides information on the organization of the guide and about its intended audience. It also lists the knowledge prerequisites required for this volume and provides resources, such as document conventions, used in this guide.
- **Chapter 1: Introduction to .NET**. This chapter introduces the Microsoft .NET architecture, application security, and .NET XML Framework. It explains the different paths, scenarios, and techniques for migrating a UNIX application to .NET using Visual Studio .NET 2003.
- **Chapter 2: Developing Phase: Process Milestones and Technology Considerations**. This chapter provides information on starting the Developing Phase and provides insight into the development environment for the migration exercise. It also discusses the major tasks and deliverables that should be identified and planned at the start of the Developing Phase.
- **Chapter 3: Developing Phase: .NET Interoperability**. Microsoft .NET provides several interoperability mechanisms to interoperate with the native code, thus facilitating the migration process and preserving the investments in the existing code. This chapter discusses how these interoperability mechanisms can be applied to reuse the existing code with minimal changes.
- **Chapter 4: Developing Phase: Process and Thread Management**. This chapter discusses the differences between the UNIX and the Microsoft .NET environments in the context of process and thread management programming. In addition, this chapter outlines the various options available for converting the code from the UNIX to the Windows environment using .NET and illustrates these options with appropriate source code examples.
- **Chapter 5: Developing Phase: Memory and File Management**. This chapter discusses the programming differences between the Microsoft .NET Framework and the UNIX environment in the following two categories: memory management and file management.
- **Chapter 6: Developing Phase: Migrating Using .NET**. This chapter discusses the programming differences between UNIX and Microsoft .NET Framework. These differences are addressed in the following categories: signals and events, exception handling in .NET, sockets and networking, interprocess communication, daemons versus services, and database connectivity.
- **Chapter 7: Developing Phase: Migrating the User Interface**. This chapter describes how to migrate from a UNIX-based user interface (UI) to a Microsoft Windows UI using Windows Forms provided by Microsoft .NET.
- **Chapter 8: Developing Phase: Additional Features in .NET**. This chapter describes some of the other features of the Microsoft .NET Framework that you can use in migrating applications from UNIX. These features include securing applications in .NET, isolated storage, serialization, .NET remoting, XML Web services in .NET, Enterprise Services in .NET, and enterprise templates.
- **Chapter 9: Developing Phase: Deployment Considerations and Testing Activities**. This chapter discusses the key development considerations for deploying applications migrated to the Microsoft .NET Framework. It also discusses the testing activities involved in the Developing Phase.
- **Chapter 10**: **Stabilizing Phase**. This chapter describes the suggested strategy for stabilizing an application that has been migrated from UNIX to the Microsoft Windows operating system. The Stabilizing Phase involves testing the application for the expected functionality and improving the quality of the application to meet the acceptance criteria set for the project.

Resources

This section describes the various resources that are included in the *UNIX Custom Application Migration Guide* and information that will assist in using the guide.

Acronyms

Please see the Acronyms list accompanying this guide for a list of the acronyms and their meanings used in this volume.

Document Conventions

The document conventions used in this volume are primarily designed to help you to quickly identify the operating system and the interface (command line or graphical) being discussed. The platforms discussed in this volume are Microsoft Windows and UNIX. In general, Windows operating system commands are executed by clicking user interface (UI) elements, and these elements are visually distinguished in this volume by the use of bold text. In contrast, the UNIX operating system typically uses a command-line interface, and these instructions are visually distinguished in this volume by the use of monospace font.

These interface and execution differences are not absolute; and in cases where visual cues do not clearly delineate between operating systems, the text will clearly make this distinction.

Table 0.1 lists the document conventions used in this volume.

Table 0.1. Document Conventions

Text Element	**Meaning**
Bold text	Used in the context of paragraphs for commands; literal arguments to commands (including paths when they form part of the command); switches; and programming elements, such as methods, functions, data types, and data structures. Also used to identify the UI elements.
Italic text	Used in the context of paragraphs for variables to be replaced by the user. Also used to emphasize important information.
Monospace font	Used for excerpts from configuration files, code examples, and terminal sessions.
Monospace bold font	Used to represent commands or other text that the user types.
Monospace italic font	Used to represent variables that the reader supplies in command- line examples and terminal sessions.
Shell prompts	The MS-DOS® prompt is used in Windows.
Note	Represents a note.
`Code`	Represents code.

Code Samples

The build volumes, Volume 2: *Migrate Using Windows Service for UNIX 3.5*, Volume 3: *Migrate Using Win32/64,* and Volume 4: *Migrate Using .NET,* of this guide contain several code samples to illustrate certain programming concepts. These code samples are available as source files in a Tools and Templates folder in the download version of this guide, available at http://go.microsoft.com/fwlink/?LinkId=30864.

.

Chapter 1: Introduction to .NET

This chapter introduces the Microsoft® .NET architecture, application security, and .NET XML Framework. It explains the different paths, scenarios, and techniques for migrating a UNIX application to .NET using Visual Studio® .NET 2003. The chapter also discusses the platform differences between UNIX and Microsoft Windows® operating systems and their implementation in .NET.

.NET Overview

This section provides an overview of the .NET Framework, its components, and their advantages.

The .NET platform is an integral component of the Microsoft Windows operating system for building and running next generation software applications and Web services. The .NET development framework provides a new and simplified model for programming and deploying applications on the Windows platform. It provides such advantages as multiplatform applications, automatic resource management, and simplification of application deployment. As security is an essential part of .NET, it provides security support, such as code authenticity check, resources access authorizations, declarative and imperative security, and cryptographic security methods for embedding into the user's application.

.NET provides a simple object-oriented model to access most of the Windows application programming interfaces (APIs). It also provides mechanisms by which you can use the existing native code. In addition, it significantly extends the development platform by providing tools and technologies to develop Internet-based distributed applications.

The .NET Platform

The Microsoft .NET platform, as illustrated in Figure 1.1 and described as follows, consists of four components:

- **.NET Framework**. The .NET Framework is a multilanguage, application execution environment that transparently manages core infrastructure services. It is a set of multiple languages/technologies used for developing and creating components to create Web Forms, Web services, and Windows applications. It supports the software life cycle for development, debugging, deployment, and maintenance of applications. The version of .NET framework that ships with Visual Studio .NET 2003 is version 1.1. The .NET Framework consists of the following parts (also depicted in Figure 1.2):
    - Common Language Runtime (CLR).
    - .NET Framework base class library.
    - Common Language Specification (CLS).
    - .NET-compliant languages.
    - Data and XML classes such as ADO.NET and XML.
    - A set of class libraries for building XML Web services.
    - ASP.NET Web Forms-based Web applications.
    - Windows Forms-based rich client applications.
    - Common Type System (CTS).
    - Microsoft Visual Studio .NET 2003 integrated development environment (IDE).

- **Development tools**. Microsoft provides the programming model, the development environment, and the tools necessary to build, deploy, and operate Web services with applications such as Visual Studio .NET 2003.
- **.NET enterprise servers**. The Microsoft .NET enterprise servers make up the Microsoft .NET server infrastructure for deploying, managing, and operating XML Web services and traditional applications. Examples of enterprise servers are Microsoft SQL Server™ 2000 and Microsoft Commerce Server 2000.
- **.NET foundation services**. A core set of building block services that execute standard tasks and act as a basis for developers to build upon. These foundation services are known as Microsoft .NET My Services and provide many features and functions. Most of the foundation services are hosted (outsourced) services. An example of a currently available Web service is Microsoft .NET Passport.

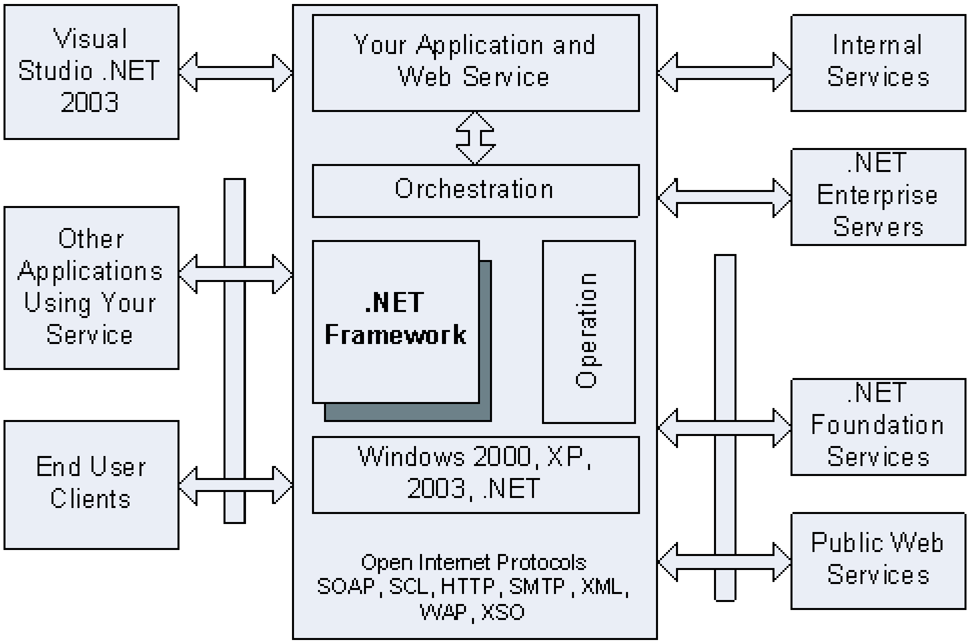


Figure 1.1. The .NET Framework overview

Advantages of .NET

The .NET Framework provides the following advantages:

- A consistent, object-oriented programming environment.
- A code-execution environment that:
  - Promotes safe execution of code.
  - Eliminates the performance problems of scripted or interpreted environments.
  - Minimizes software deployment and versioning conflicts.
- A consistent experience for both developers and users across various types of Windows-based and Web-based applications on multiple devices.
- Communication built on the industry standards to ensure that code based on the .NET Framework can integrate with any other code.

.NET is based on open Internet standards, which include Hypertext Transfer Protocol (HTTP), Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP).

Note More information on XML is available at http://msdn.microsoft.com/xml/.

More information on SOAP is available at

http://msdn.microsoft.com/library/en-us/dnsoap/html/understandsoap.asp.

Figure 1.2 depicts the overall architecture of the .NET Framework components.

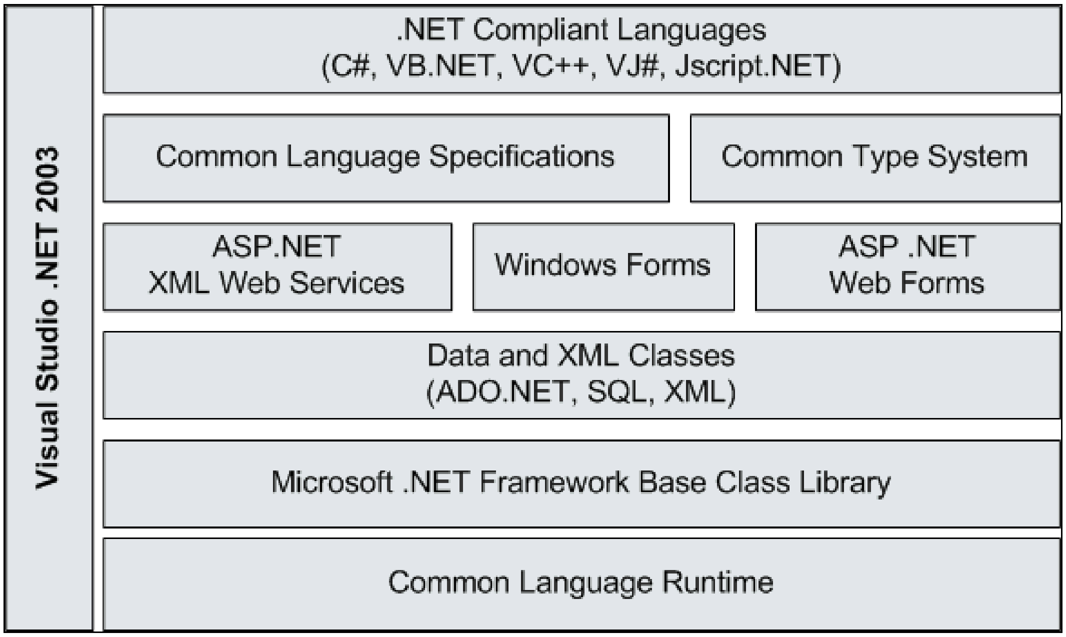


Figure 1.2. The .NET Framework architecture

Features of the .NET Framework

This section discusses some of the features of the Microsoft .NET Framework and how to use these features in migrating code from the UNIX environment.

Common Language Runtime (CLR)

The core of the .NET Framework is the CLR, the run-time environment provided by .NET. The runtime manages code at execution time and provides core services such as memory management, thread management, remoting, and strict type safety enforcement.

Figure 1.3 depicts the components of CLR.

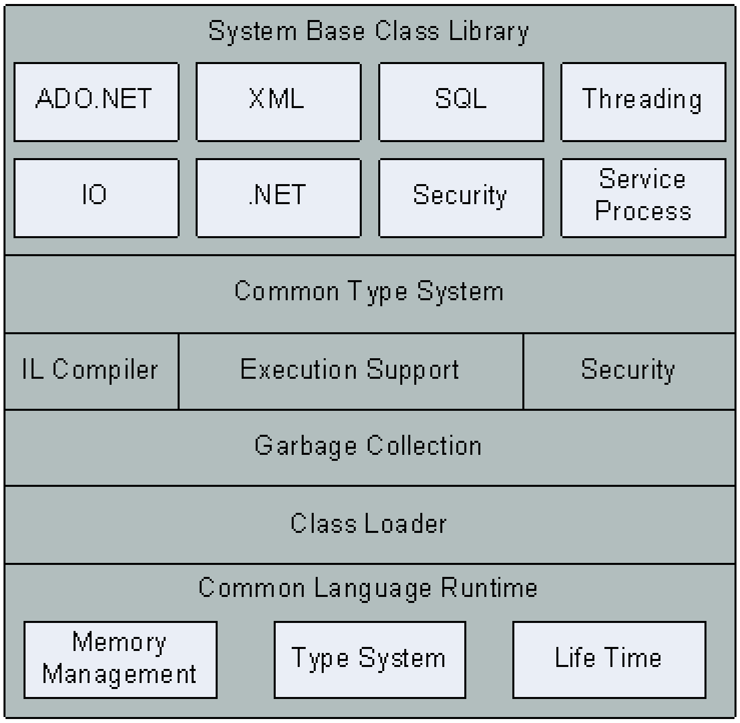


Figure 1.3. CLR components

CLR Features

UNIX applications, redeveloped on .NET, can make use of all the features provided by the CLR, including:

- Simplified development and deployment of applications.
- Application memory management.
- Improved performance, scalability, and reliability.
- Cross-language inheritance.
- Multiple language support.
- Automatic garbage collection.
- Security.
- Strong type checking.
- Access to type metadata.
- Unified exception handling.

- Interoperability with existing code in COM (Component Object Model) objects and Microsoft Win32® DLLs.
- Loading and executing code.
- Just-in-time (JIT) compilation of Microsoft intermediate language (MSIL) to native code.
- Side-by-side execution for multiple assembly versions.
- Other developer support services that include debugging and run-time profiling.
- Versioning and deployment support.

The .NET runtime also enforces other forms of controlled code access that promote security and robustness. Code management is a fundamental principle of the runtime. Code that targets the CLR is known as managed code, whereas code that does not target the CLR is known as unmanaged code. Unmanaged code can also be used in the .NET environment using interoperability techniques as explained in Chapter 3, ".NET Interoperability" of this volume.

All .NET applications compile to a common language called MSIL. A JIT compiler then compiles MSIL to optimized native code.

Benefits of CLR

This various benefits offered by the CLR are as follows.

Security

The runtime enforces code access security. The managed components have varied degrees of trust level depending on a number of factors, including their origin. Even if the same active application is using the managed component, depending on the trust level, the managed component might or might not be capable of performing file-access operations, registry-access operations, or other sensitive functions.

For example, users can trust that an executable embedded in a Web page can play an animation or a song, but cannot access their personal data, file system, or network.

Code Robustness

The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the Common Type System (CTS). The CTS ensures that all managed code is self-descriptive. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS.

Developer Productivity

The runtime also accelerates developer productivity by enabling the developers to write applications in the development language of their choice and still take advantage of all the features of the runtime. The language compilers that target the .NET Framework make the features of the .NET Framework available to the existing code written in that language, thus greatly facilitating the migration process for the existing applications.

Performance

The runtime is designed to enhance performance. JIT compiling enables all managed code to run in the native code of the system on which it is executed. At the same time, the memory manager removes the possibilities of fragmented memory and increases the memory locality-of-reference to improve the performance further.

Interoperability

The runtime, although designed for modern software, also offers backward compatibility by supporting older software. Interoperability between managed and unmanaged code provides seamless integration to developers to continue to use necessary COM objects and exported functions in unmanaged DLLs.

.NET Framework Base Class Library

The Microsoft .NET Framework 1.1 base class library is an object-oriented class library providing an integrated set of classes that expose the underlying functionality of the Win32 API as well as some other additional capabilities. These classes integrate tightly with the CLR. Third-party components can also integrate with the classes in the .NET Framework. In Microsoft .NET library, all class (types) are grouped in *namespaces*. A namespace is a grouping of similar kinds of classes.

The .NET Framework classes enable you to perform a range of common programming tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes classes that support a variety of specialized development functions. For example, you can use the .NET Framework to develop a variety of applications such as console applications, GUI (graphic user interface) applications, and Web applications.

All .NET languages can use these language-independent classes. This enables the programmers to choose the language and tools best suited for the job or the language with which they have the most experience and still share their code and create new subclasses from classes written in a different language. This code reuse can dramatically increase team productivity and decrease development costs. Figure 1.4 depicts some of the namespaces and their classes in the .NET Framework.

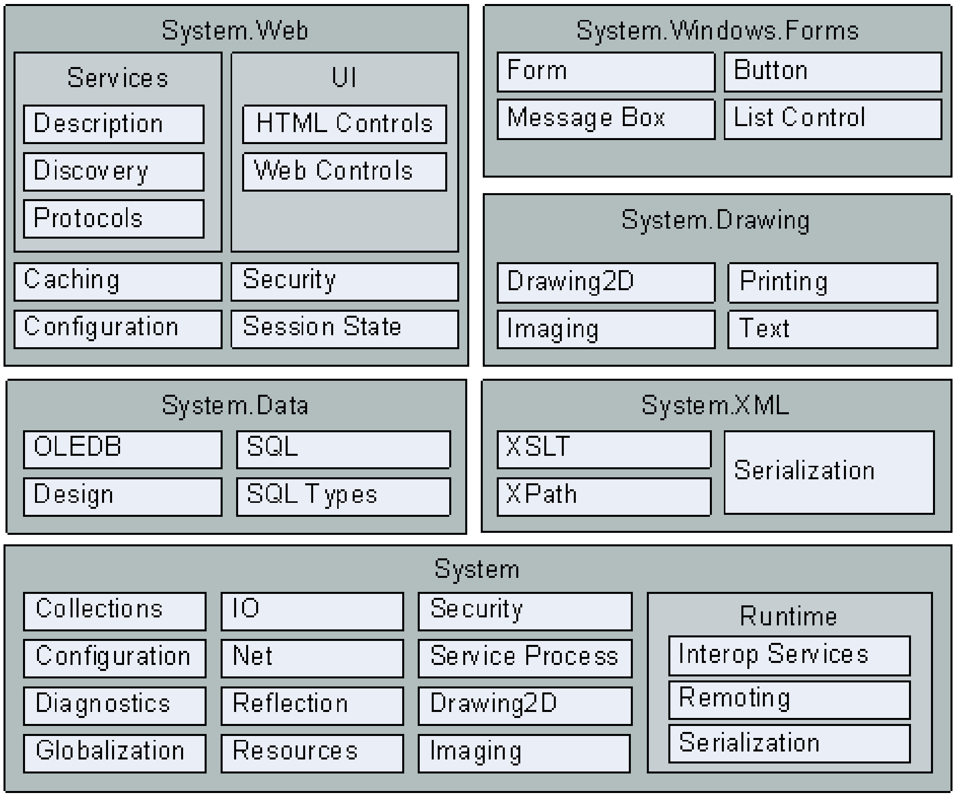


Figure 1.4. The .NET Framework base class library

.NET Tools and Technologies

From a migration perspective, .NET provides various tools and technologies that help you to migrate or redevelop a UNIX application on Windows. Some of these tools and technologies are discussed in the following sections.

Database - ADO.NET

.NET provides ADO.NET for migrating the database-related components of a UNIX application to .NET. ADO.NET is a collection of classes, structures, and interfaces that manage the data access for different databases. .NET provides this data access technology to enable you to connect to different databases including ODBC-aware (open database connectivity) databases.

Networking - System.NET

The **System.NET** namespace in .NET allows you to replicate the networking functionality of a UNIX application on Windows.

Transaction - COM+ Services

.NET provides COM+ services, also referred to as Enterprise Services in .NET, to migrate applications that involve large amount of transactions.

Rich Client - Windows Forms

Windows Forms enable you to replicate the X/Motif-based GUI of a UNIX application on Windows. Windows Forms facilitate building of Windows rich-client applications that take advantage of the CLR. The Visual Studio .NET 2003 IDE also aids in the rapid redevelopment of GUI on Windows with the same look and feel as in UNIX.

Web Applications - ASP.NET

.NET provides Web Forms and ASP.NET for migrating the existing Web application on UNIX to Windows. Web Forms and ASP.NET enable you to develop real-world Web applications on Windows.

Application Integration - XML, SOAP, and Web Services

.NET supports XML, SOAP, Web services, and .NET servers that enable a migrated application to integrate with the older applications and other applications in your enterprise.

.NET XML Framework

This section lists the XML APIs available in the .NET class library that you can use for various XML-related operations.

XML is truly a core technology substrate in .NET. All other parts of the .NET Framework (such as ASP .NET and Web services) use XML as their native data representation format. The .NET Framework XML classes are also tightly coupled with Managed Data Access (ADO .NET). Traditionally, there have always been different programming models for working with relational and hierarchical data. .NET breaks that tradition by offering a more deeply integrated programming model for all types of data.

New Suite of XML APIs

Microsoft .NET introduces a new suite of XML APIs built on such industry standards as Document Object Model (DOM), XPath, XML Structured Definitions (XSD), and Extensible Stylesheet Language Transformations (XSLT). The .NET Framework XML classes offer convenience and better performance. The .NET XML Framework also provides a more familiar programming model, tightly coupled with the various classes present in **System.Data** and **System.Xml** namespaces, which encapsulate a number of functionalities that previously had to be accomplished manually.

.NET XML Namespaces

The System.Xml assembly contains a broad range of general-purpose XML support features, such as:

- Basic I/O model.
- I/O of primitive types.
- In-memory traversal.
- Filtering based on XPath expressions.
- Transformations based on XSLT.

The .NET XML stack is partitioned over several namespaces, such as:

- **System.Xml.XPath**
- **System.Xml.Xsl**
- **System.Xml.Schema**
- **System.Xml.Serialization**

XML-based I/O

All XML-based I/O is performed using a streaming interface suite as follows:

- Streams are supported in both pull-mode (read) and push-mode (write).
- Built-in streaming adapters use the **System.IO.Stream** class library.
- Abstract interfaces allow you to provide your own XML providers/consumers.

.NET DOM Implementation

The .NET DOM implementation (**System.Xml.XmlDocument**) supports all W3C DOM Level 1 core and all DOM Level 2 core specifications, but with a few minor naming changes. The DOM loading is built on top of XmlReader, while the DOM serialization is built on XmlWriter. This makes it possible to extend how the DOM interacts with applications in numerous ways.

Note More information on W3C DOM Level 1 core and Level 2 core specifications is available at http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/ and http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/.

Transformations

The **XslTransform** class manages XSLT transformations in the .NET Framework. **XslTransform** resides in the **System.Xml.Xsl** namespace and uses **XmlNavigator** during the transformation process. As with all XSLT processors, **XslTransform** accepts an XML document, an XSLT document, and some optional parameters as input. It can produce any type of text-based output; it also supports reading the result of the transformation using a custom **XmlReader**.

.NET Application Security

This section provides an overview of various security models available in the .NET Framework. The .NET Framework provides a rich security system, capable of confining code to run in a tightly constrained, administrator-defined, security context.

Role-based Security

The .NET Framework provides a developer-defined security model called role-based security that attaches security to the users and their groups (or roles). The principal abstractions of role-based security are principal and identity.

Code Access Security

Additionally, the .NET Framework also provides security on code, referred to as *code access security* (also referred to as evidence-based security). With code access security, a user may be trusted to access a resource but if the code that the user executes is not trusted, then access to the resource is denied. Code access security also provides a highly protective way of securing the assemblies from malicious attacks. Security based on code, as opposed to specific user, is a fundamental facility that permits security to be expressed on mobile code. Any number of users may download and execute mobile code, which is unknown at the time of development. Code access security focuses on some core abstractions, namely, evidence, policies, and permissions.

The security abstractions for role-based security and code access security are represented as types in the .NET Framework class library and are user-extendable.

The .NET Framework security system functions atop the traditional operating system security. This adds a second, more expressive and extensible level to the operating system's security. Both layers complement each other. (It is conceivable that an operating system security system can delegate some responsibility to the CLR security system for managed code because the run-time security system is more configurable then the traditional operating system security.)

Note More information on .NET Framework security is available at http://msdn.microsoft.com/security/securecode/dotnet/default.aspx.

Implementation in .NET

Chapter 1, “Introduction to Win32/Win64” of Volume 3 has already discussed the platform differences between UNIX and Windows from various aspects. The following topics give an overview for implementing the following architectural elements in .NET:

- Processes and threads
- Memory management
- File management
- Signals, exceptions, and events
- Networking
- Interprocess communication
- User interface
- Daemons versus services
- Deployment

Processes and Threads

The .NET Framework further divides an operating system process into lightweight managed subprocesses, called application domains, which provide a versatile unit of processing in .NET applications. These application domains are used to provide isolation between applications and even within a single process. Several application domains can be run in a single process with the same level of isolation that would exist in separate processes. Historically, process boundaries have been used to provide isolation between applications, but application domains provide a level of isolation equivalent to that of a process boundary, however at a much lower cost of performance.

The **System.Threading** namespace in .NET provides all the classes and interfaces necessary to enable multithreaded programming. In addition to classes for synchronizing thread activities and providing access to data (for example, Mutex, Monitor, Interlocked, and AutoResetEvent), this namespace includes a **ThreadPool** class that allows use of a pool of system-supplied threads and a **Timer** class that executes callback methods on the thread pool threads. The next chapters discuss application domains and the threading namespaces in detail.

Memory Management

The garbage collector of the .NET Framework provides automatic memory management. It allocates and releases the memory for managed objects and, when necessary, executes the appropriate methods at the appropriate times in order to properly clean up the unmanaged resources. Automatic memory management simplifies development by eliminating the common bugs that arise from manual memory management schemes.

File Management

In .NET, the **System.IO** namespace provides an object-oriented tool to work with files and folders. It provides a collection of properties, methods, and events to process text and data, thus enabling you to perform file and directory operations with greater ease. For more information on the **System.IO** namespace, refer to Chapter 4, “Memory and File Management” of this volume.

Signals, Exceptions, and Events

.NET Framework provides an *event handler* mechanism to handle events. An event handler is a procedure in your code that determines the actions that must be performed when an event (such as the user clicking a button or a message queue receiving a message) occurs. When an event is raised, the event handler (or a handler) that receives the event is executed. Events can be assigned to multiple handlers and the method that handles a particular event can be changed dynamically.

The .NET Framework handles exceptions through the *exception handling* mechanism. In the .NET Framework, an exception is an object that it inherits from the **System.Exception** class. An exception originates from an area of code where a problem occurred. The exception is passed up the stack until the application handles it or the program terminates. All .NET languages handle exceptions in a similar manner. Each language uses a form of try/catch/finally structured exception handling.

Networking

The .NET Framework class library includes two namespaces that consists of classes that help you with networking; these are **System.Net** and **System.Net.Sockets**.

The **System.Net** classes provide a simple, yet complete solution for writing networked applications in managed code. The **System.Net.Sockets** classes deals with the TCP/UDP and sockets.

Interprocess Communication

In the .NET environment, application domains enable more than one application to run within a single process, thus eliminating the overhead of making cross-process calls but still maintaining the same level of application isolation that would exist in separate processes. .NET also supports the concept of thread local storage, by which data can be stored in a thread and accessed anywhere the thread exists.

Microsoft .NET Remoting provides a rich and extensible framework for objects residing in different application domains, in different processes, and in different computers to communicate with each other seamlessly. .NET Remoting offers a powerful, yet simple, programming model and run-time support for making these interactions transparent.

User Interface

In the .NET environment, user interfaces can be developed as Windows or Web Forms. Some of the advantages of using these forms include the following:

- Simplicity and power
- Rich graphics
- Flexible controls
- Lower total cost of ownership
- Architecture for controls
- High security
- XML Web services support
- Data awareness
- ActiveX control support
- Easy licensing
- Enhanced printing support
- Accessibility
- Design-time support

Daemons vs. Services

The .NET Framework class library includes the **System.ServiceProcess** namespace that provides classes to implement, install, and control Windows service applications.

Services are installed using an installation tool, such as InstallUtil.exe. The **System.ServiceProcess** namespace provides installation classes that write service information to the registry.

The **ServiceController** class enables you to connect to an existing service and manipulate it or get information about it. This class is typically used in an administrative capacity; it enables you to start, stop, pause, continue, or perform custom commands on a service.

Deployment

In .NET Framework applications, assemblies are the building blocks. They form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. Current Win32 applications have two versioning problems with their building blocks (dynamic-link libraries):

- Versioning rules cannot be expressed between pieces of an application and enforced by the operating system.
- Inability to maintain consistency between sets of components that are built together and the set that is present at runtime.

These two versioning problems combine to create DLL conflicts, or DLL Hell, where installing one application can inadvertently break an existing application because a certain software component or DLL that was installed was not fully backward compatible with a previous version. The CLR uses assemblies to provide a complete solution for DLL conflicts. Assemblies allow the runtime to:

- Enable developers to specify version rules between different software components.
- Enable the infrastructure to enforce versioning rules.
- Enable the infrastructure to allow multiple versions of a component to run simultaneously (called side-by-side execution).

Each computer, where the CLR is installed, has a machine-wide code cache called the Global Assembly Cache (GAC). The GAC stores assemblies specifically designated to be shared by several applications on the computer.

Assemblies deployed in the GAC must have a strong name. When an assembly is added to the GAC, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with.

Summary of Platform Differences

Table 1.1 lists the basic platform differences that exist between UNIX, Windows, and .NET.

Table 1.1. Summary of Platform Differences

Architectural Element	UNIX	Windows	.NET
Processes	Processes have a parent-child relationship. Process boundary provides isolation between applications.	Processes do not have a parent-child relationship. Process boundary provides isolation between applications.	Processes do not have a parent-child relationship. An operating system process is further divided into subprocesses called application domains. Application domains provide isolation between applications.
Threads	UNIX threads are built upon the POSIX standard, known as Pthreads.	The Windows operating system provides built-in support for threads and thread synchronization using Platform SDK.	.NET provides the **System.Threading** namespace to support multithreading in conjunction with Windows threads.
Memory	At application level, memory management is manual (in the hands of the programmer).	At application level, memory management is manual (in the hands of the programmer).	Memory management is automatic and controlled by the CLR, eliminating bugs such as memory leakages from manual memory management.
File	File operations are performed through low-level I/O functions and stream I/O functions.	File operations are performed through low-level I/O functions and stream I/O functions.	The **System.IO** namespace provides an object-oriented framework for handling files and folders.
Signals	UNIX supports a wide range of signals. Signals are software interrupts that catch or indicate different types of events.	Windows supports only a small set of signals that is restricted to exception events. Native signals, event objects, and messages are the recommended mechanisms to replace common UNIX signals.	.NET does not support signals. Events and exception handling mechanisms in .NET are the recommended mechanisms to replace common UNIX signals.
Networking	UNIX supports networking through sockets.	In Windows, sockets are implemented using WinSock libraries.	The **System.Net.Sockets** namespace provides networking support in .NET.

Architectural Element	UNIX	Windows	.NET
Interprocess communication	UNIX provides shared memory, pipes, and message queues for interprocess communication.	Windows provides shared memory, pipes, events, Dynamic Data Exchange (DDE), Component Object Model (COM), and mailslots for interprocess communication.	.NET Framework provides .NET Remoting and Microsoft Message Queue for interprocess communication.
UI differences	UI is developed using X Windows and Motif, which are the standard windowing system and windowing system library respectively on UNIX.	UI is developed using MFC (Microsoft Foundation Classes), ATL (Active Template Library), or GDI+ (Graphics Device Interface).	.NET Framework provides Windows Forms and Web Forms for development of UI of rich client and Web applications respectively.
Deployment differences	UNIX provides shared objects for developers to group common functionality and deploy them.	Windows provides DLLs for developers to group common functionality and deploy them.	In .NET, assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions.
Daemons and services	UNIX supports daemons (a process that runs in the background and does not require a user interface).	Windows service is the equivalent of daemons.	.NET provides the **System.ServiceProcess** namespace to implement, install, and control Windows service applications.

.NET Migration Paths

This section discusses the various .NET migration paths. You can use this information in reengineering code in .NET and in understanding the interaction of the .NET code with existing applications. This section also helps you to choose the best migration path based on the nature of existing UNIX applications.

To move UNIX applications to the Microsoft .NET Framework, you need to migrate the existing UNIX C or C++ code to Windows. For example, if the UNIX applications use third-party libraries and if the equivalents of such third-party libraries are already available on Windows, then the UNIX to Windows move is much easier. The code migrated from UNIX can then integrate the .NET Framework features.

For UNIX applications involving code in Java, Visual Studio .NET 2003 provides a tool known as Java Language Conversion Assistant (JLCA) that can automatically convert existing Java language code into C#. You can use this tool in migration scenarios where the existing UNIX application contains Java code.

With a large code base of installed UNIX applications, you are unlikely to relish the thought of throwing out the entire environment and starting again with an unfamiliar platform. Fortunately, you do not necessarily have to do this. As illustrated in this guide, methods are available by which you can preserve the existing code while moving to .NET.

Analyzing Application Types

The application type and the ease with which you can move the application from UNIX to Windows should decide whether to use the .NET interoperability strategies or to redevelop the application completely on the .NET environment. Different application types and strategies for migrating these application types are discussed in the following sections.

Static Application

Static applications are applications that are in the later stages of their life cycle, with stable code, and with little or no changes planned. If these applications can be migrated to Win32 using little or no code changes, then they can also be adapted to make use of the capabilities of the .NET platform by using the interoperability services provided by .NET. This way you can not only speed the application migration but also preserve your investments in the existing code. If the static application is an enterprise application, such as a portal or a content management system, then you can take advantage of the capabilities of the .NET servers, in addition to reusing the existing code.

Evolving Application

Evolving applications are applications that are constantly being changed and enhanced. An evolving application typically contains both static and dynamic components. Static components are the parts of the application that do not change, whereas dynamic components are the parts of the application that are evolving.

First, identify the static and dynamic components of the application and then use the .NET interoperability strategies to migrate the static components.

The dynamic components can be redeveloped in .NET targeting the benefits offered by the CLR and making full use of the other benefits of the .NET development platform.

Table 1.2 lists the .NET migration strategies for the different types of applications.

Table 1.2. Migration Strategies

Application	Nature of Existing Application	Recommended Solution
Static application	Native UNIX application (UNIX APIs, X Windows, and Motif).	Migrate the application to Win32/Win64 with minor code changes and then interoperate with the .NET code to preserve and make use of the investments made earlier.
Static application	Enterprise applications such as portal, content management, and others.	Use .NET servers and interoperate with the migrated Win32/Win64 or unmanaged code.
Evolving application	Native UNIX application (Unix APIs, X Windows, and Motif).	Reengineer in .NET
Evolving application	Enterprise applications such as portal, content management, and others.	Use .NET servers; interoperate with the migrated Win32/Win64 or unmanaged code, and reengineer in .NET.

The subsequent sections discuss the different migration strategies listed in Table 1.2.

- Reengineering using the .NET Framework.
- Interoperating with the existing code.
- Utilizing .NET servers.

Reengineering Using the .NET Framework

This section describes reengineering the applications using the .NET Framework and its advantages. A reengineering strategy is appropriate when an application needs to evolve further and the costs of porting the application outweigh the benefits, or when specific application qualities (such as performance or scalability) require that the code be written specifically for the Windows platform.

Rewriting an application has a number of significant advantages, including:

- Code robustness.
- Accelerates developer productivity.
- Enhanced performance and scalability.
- Interoperability between managed code and unmanaged code.
- Flexible language options.
- Improved tool support.
- Rich class library.

You can redevelop an application using any of the .NET compatible languages, such as Microsoft Visual Basic® .NET, C#, and Managed C++. Managed code, which is compiled to Intermediate Language (IL) and not to machine code, is created using these languages. The redeveloped code on .NET can make use of all the features of the CLR. The code that is redeveloped in .NET to target the CLR goes through the managed execution process. The following steps are involved in the managed execution process.

1. **Choose a compiler**.

   To obtain the benefits provided by the CLR, you must use one or more language compilers that target the runtime, such as C#, Microsoft Visual C++®, Microsoft Visual Basic .NET, Microsoft Visual J#®, Microsoft JScript®, or one of the many third-party compilers such as Eiffel, Perl, or COBOL. Multiple .NET languages support a common set of data types. For example, a string in C# is the same data type as a string in Visual Basic .NET. If you are migrating a C++ application, consider using the Visual C++ .NET for redevelopment. If you are migrating a Java application, you can consider Visual C# for redevelopment.

2. **Compile code**.

   On compiling the code, it is converted to Microsoft Intermediate Language (MSIL) code using the selected .NET language compiler. When you execute the code for the first time, the Just-In-Time (JIT) compiler translates MSIL into the native code. During this compilation, code must pass a verification process that examines MSIL and the metadata to find out whether the code can be determined to be type safe. There are two kinds of JIT compilers: Standard JIT and Econo-JIT. Econo-JIT has a faster compilation speed and a lesser compiler overhead than Standard-JIT. However, the Standard-JIT generates more optimized code than the Econo-JIT and includes the verification of MSIL code.

3. **Execute code**.

   The CLR provides the infrastructure for executing the code as well as a variety of services that can be used during execution.

Figure 1.5 illustrates the managed execution process in a .NET application.

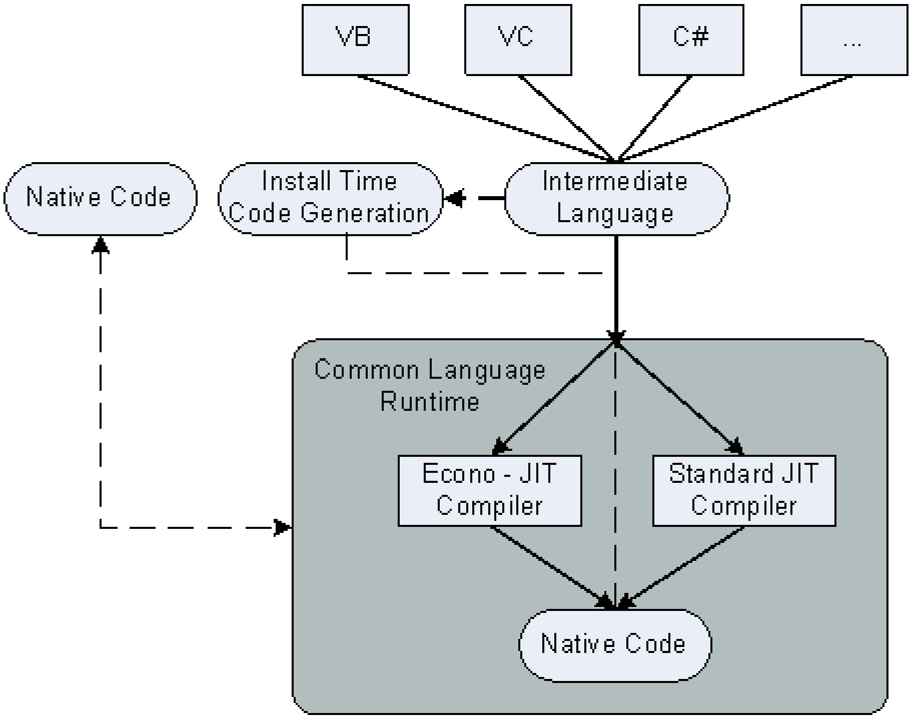


Figure 1.5. Managed execution process

Reengineering is a costly and time-consuming option. This strategy requires a thorough analysis of the UNIX application functionality before redesigning a .NET-based application architecture. This ensures that the rewrite of the UNIX code takes full advantage of the .NET platform and the Windows application platform. This strategy has a great risk, but it also can produce the best results.

Interoperating with the Existing Code

This section discusses interoperating with existing code and outlines the various interoperability techniques. With the help of the instructions provided in this section, you can choose the interoperability technique that is best suited for your UNIX application.

In the interoperate strategy, the application is migrated with the minimum necessary changes to the source code, using UNIX-compatible libraries and tools that are available on the Windows platform (and which can include Microsoft Interix). The migrated code may require significant changes to enable the code to run on the new platform if the original code is not fully standards-compliant or if the code elements (for example, device drivers) are specific to the original system.

Reasons for choosing to port an application directly to Windows are as follows:

- Rewriting the code from scratch is costly or time consuming.
- A large amount of source code already exists.
- The application requires cross-platform support.
- The application uses a large number of UNIX APIs.

As long as the application functionality is tightly controlled, the interoperate strategy reduces the risk of negatively altering the application by preserving the business logic, as well as reducing the need for new documentation.

Unmanaged code is the code that runs outside the CLR. Unmanaged code compiles directly to machine code, which runs on the computer where you compile the code and on other computers, as long as the other computers have the same chip or are nearly the same. Unmanaged code does not get services, such as security or memory management, from an invisible runtime; it gets these services from the operating system.

.NET provides interoperability with unmanaged code in the following ways:

- Managed Extensions for C++
- .NET Interoperability services
    - C++ Interoperability (It Just Works)
    - Platform Invocation services (P/Invoke)
    - COM Interoperability services

The following sections describe these interoperability techniques in detail.

Managed Extensions for C++

Managed Extensions for C++ provide a systematic methodology for wrapping of the existing unmanaged C++ classes. This enables use of the unmanaged code from the managed applications. In this method, a managed wrapper class for the existing C++ unmanaged class is created. This wrapper class interoperates with its unmanaged equivalent and, in essence, acts as a proxy for accessing the unmanaged class.

For example, if the UNIX application to be migrated is a high-performance application, then C++ wrappers can be used because they offer the following advantages:

- **Single level of wrapping**. In COM interoperability, there are two levels of wrapping: one at the COM level and another at the RCW (runtime callable wrapper) level.
- **Limited wrapping**. Member functions accessed from the managed code/application only need to be wrapped.
- **Data marshalling**. Fine-tuned data marshalling.

However, the disadvantage in using C++ wrappers is that it involves more coding because you need to wrap the unmanaged class, its constructors, destructors, and member functions.

For more information on wrapping unmanaged classes with Managed Extensions for C++, refer to Chapter 2, ".NET Interoperability" of this volume.

.NET Interoperability Services

.NET provides the following services for interoperability with unmanaged code:

- Platform Invocation services (P/Invoke)
- C++ Interoperability (It Just Works)
- COM Interoperability services

These topics are explained along with examples in Chapter 3, ".NET Interoperability" of this volume.

Platform Invocation Services (P/Invoke)

Platform Invocation services (P/Invoke) enable the managed code to call the C-style functions, which are exposed by the native dynamic-link libraries (DLLs). P/Invoke services provide a direct way of using the C functions from the existing native DLLs in a managed application. P/Invoke services are used in conjunction with the DllImport attribute, which is used to import functions from an unmanaged DLL into a managed application.

As an example, consider a situation where the UNIX code can be migrated to Windows with minimal changes and you need to use this code in a new C# or Visual Basic .NET application or project. After migrating to Windows, you can compile the code into a DLL and call it from any .NET project.

This method provides the following advantages:

- **Usage**. You can use this method with all .NET languages (C#, Visual Basic .NET, and Managed Extensions for C++). Therefore, new C# or Visual Basic .NET projects can use the DLLImport attribute to connect to the unmanaged code.
- **Quick reuse of code**. This method enables quick reuse of the existing native code.

However, you may not use this method in the following situation: If a function in the DLL returns an unmanaged string, such as

```
[DllImport("mylib")]
extern "C" String * MakeSpecial([MarshalAs(UnmanagedType::LPStr)]
String *);
```

then you will not be able to delete the memory of the unmanaged string that is returned. In such cases, the C++ Interoperability It Just Works (IJW) method described in the next section is the best choice.

The disadvantages of the P/Invoke method are:

- **Used with attribute only**. Unmanaged APIs cannot be used directly without the use of the DLLImport attribute.
- **Memory leakage**. This method sometimes causes memory leakages for some C-style functions using unmanaged data types as illustrated by the earlier example.

C++ Interoperability (It Just Works)

This is a platform invocation service, available only in Managed C++, which is designed for using the unmanaged APIs directly without using the DLLImport attribute.

When the UNIX code can be migrated to Windows with minimal changes, then this code can also be compiled in a .NET-managed C++ project using the /CLR switch for the compiler.

This method offers the following advantages:

- **No memory leakage**. Memory leakages, such as the one shown in P/Invoke with the DLLImport attribute, do not occur.
- **Can be used directly with unmanaged APIs**. Use the unmanaged APIs directly.
- **Faster execution**. This method is slightly faster. For example, the IJW stubs do not need to check for the pin or copy data items as the developer does that explicitly.
- **Better performance**. For example, if you need to call several unmanaged APIs using the same data, marshaling all APIs once up front and passing the marshaled copy around is much more efficient than remarshaling APIs every time.
- **Reuse marshaled data**. This method provides the capability to marshal the data once and reuse the marshaled data at multiple call sites, thereby amortizing the cost of the marshaling.
- **Quick reuse of code**. This method enables quick reuse of the existing native code.

The disadvantages of this method are:

- **Used with C++ language**. This method can be used only with the C++ programming language (although the interop can be used with any language).
- **Pointer calls required**. Specify explicit marshalling in the code, which means extra pointer calls are necessary.

COM Interoperability Services

The .NET Framework has made provisions for interoperability by implementing various wrappers for COM objects to allow exposure of their properties and methods to .NET components. These wrappers make it easy to make the connection between COM and .NET.

For example, if there is a piece of common code that can be migrated to Windows with minimal changes and is to be reused across many applications in an enterprise, you can create a COM component. This COM component can be used in other applications as well as a .NET application. However, as this method involves creating a COM component over the code migrated from UNIX, it is least used in UNIX migration scenarios.

Some of the advantages offered by COM Interop services are:

- **Used with all COM-enabled languages**. Any COM-enabled language can use COM, which is a well-understood interface.
- **Marshaling**. COM supports marshaling, although COM objects reside in separate processes or address spaces or even different computers. The operating system takes care of marshaling the call and calling objects running in a different application (or address space) on a different computer.
- **Consistency**. Maintains the consistency of the programming model.
- **Bridge**. Provides the bridge between COM and runtime.

For example, a C++ code in UNIX that can run on Windows with minimal changes can interact with a Visual Basic application that already exists in the Windows platform and the .NET application on different computers using COM.

Other situations where you can select COM interoperability are:

- **Code in UNIX exists**. A large amount of UNIX code, which you can make a COM component with minimal changes.
- **Cross-platform libraries**. If the code in UNIX makes extensive use of third-party libraries that are available in Win32, but not in .NET, you can make this code as a COM component and access the COM component in .NET applications.
- **Application defined in layers**. If the application is defined in layers and one of the layers interacts with the system level APIs, which need minimal changes to migrate to Windows, the components of this layer can be made as COM components and access the COM components in .NET applications.

However, this method’s performance is low as it involves two levels of wrapping (the COM interface and RCW).

Performance Considerations

The interoperability strategies discussed earlier involve data marshaling. For example, the strings in the native code may use the ANSI format. These strings are not compliant with the CLS and so you cannot use them directly with other .NET Framework-compliant languages. The CLR supports Unicode string format. Converting between the two formats requires data marshalling, which creates an additional overhead. However, there is no marshaling cost when converting between *blittable types*. Blittable types have the same representation in managed and unmanaged code. For example, there is no cost for converting between int and Int32.

The P/Invoke services have an overhead of between 10–30 x86 instructions per call. In addition to this fixed cost, marshaling results in additional overhead. For higher performance, it may be necessary to have fewer P/Invoke calls that marshal as much data as possible instead of having more calls that marshal less data per call.

Utilizing .NET Servers

.NET incorporates the .NET enterprise servers, including the Microsoft Application Center, Microsoft Commerce Server, Microsoft Host Integration Server, Microsoft Internet Security and Acceleration Server, Microsoft Mobile Information Server, Microsoft SharePoint® Portal Server, Microsoft BizTalk® Server, Microsoft SQL Server™, and Microsoft Exchange Server. These servers support the .NET Framework and provide powerful back-office services on which you can build and run .NET applications.

.NET enterprise servers are integrated tightly with the Windows and COM, and they provide enhanced performance and capabilities for the existing Windows applications and architectures. The .NET enterprise servers are XML-enabled so that they also provide a comprehensive set of services for building applications and enterprise solutions on the new .NET Framework.

For example, an enterprise application in a UNIX environment can be migrated to the Windows environment by using the BizTalk Server. You can write pipeline components in .NET using any of the .NET languages, which you can refer in BizTalk Server applications for creating custom pipelines. The groups of .NET enterprise servers have been renamed as the Windows Server System.

Note More information on .NET servers is available at http://www.microsoft.com/windowsserversystem/products/default.mspx.

Migration Scenarios

This section describes the different categories of applications that are migrated, such as rich client applications, Web applications, and database applications. For each of these categories, it discusses the various techniques involved in migration.

Rich Client Applications

The majority of UNIX graphical interfaces are built on X Windows and Motif. The main user interface type in use on the UNIX platform today builds on the X Windows set of standards, protocols, and libraries.

.NET provides Windows Forms for migrating the GUI of rich client applications built on X Windows to the Microsoft Windows operating system. Windows Forms is a framework for building Windows client applications that uses the CLR. This framework provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows applications. You can write Windows Forms applications in any language that the CLR supports. Using Visual Studio.NET 2003 enables you to visually design Windows Forms, use the familiar drag-and-drop double-click techniques, and enjoy full-fledged code support including statement completion and color-coding. This enables developers to reengineer an application rapidly.

The application-programming model for Windows Forms consists of forms, controls, and their events.

- **Forms**. In Windows, the **Form** class is a representation of any window displayed in an application. Designing the user interface typically involves creating a class that derives from **Form** and then adding controls, setting properties, creating event handlers, and adding programming logic to the form.
- **Controls**. Each component added to a form, such as a button, a text box, or a radio button is called a control. Typically, you set the properties to alter the appearance and behavior of controls.
- **Events**. The Windows Forms programming model is event-based. When a control changes state, such as when the user clicks a button, it raises an event. To handle an event, an application registers an event-handling method for that event.

Web Applications

Web applications from a UNIX Web server are, usually, either a Common Gateway Interface (CGI) or Java Server Page (JSP). The CGI program is a standard for connecting external applications with information servers, such as Web servers.

A CGI program is executed in real time to generate dynamic information. For example, a CGI program on the Web server executes to transmit information to a database engine, retrieves result sets, and displays the result sets to the client browser.

JSP technology also enables development of dynamic Web pages. JSP technology uses tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page. The application logic resides in server-based resources such as a JavaBean component, which the page accesses by using the tags and scriptlets. Formatting tags such as HTML and XML tags pass directly back to the response page.

.NET provides ASP.NET, a unified Web development model that includes the services necessary for developers to build enterprise-class Web applications. ASP.NET provides Web Forms and Web services. Web Forms allow you to quickly build powerful forms-based Web pages. Web services enable the exchange of data using standards such as HTTP and XML messaging to move data across firewalls. XML Web services are not tied to a particular component technology or an object-calling convention. As a result, programs written in any language, using any component model, and running on any operating system can access XML Web services.

If an application uses CGI, then the migration strategy depends on the language in which the program was written. The languages include C/C++, Fortran, Perl, Tcl, UNIX shells, and Microsoft Visual Basic. Many Web developers write CGI programs in Visual Basic language because of its powerful text-handling capability. CGI2ASP is a framework for porting programs written in Visual Basic, using Windows CGI, to a component-based ASP application with virtually no change to the existing Visual Basic-based code.

If an application uses JSP technology, then you need to rewrite the JSP scriptlets (written in Java and residing in JavaBean components) in Microsoft Visual Basic Scripting Edition or Microsoft JScript and contained in ASP COM components. A tool named Java Language Conversion Assistant (JLCA) is also available for use in the Visual Studio .NET 2003 environment; this tool can automatically convert most JSP code to ASP.NET.

Database Applications

When moving the database applications to .NET, the most common migration strategy is to reengineer the application to use ADO.NET. Through ADO.NET, you can efficiently manage the database interactions in an application.

ADO.NET is an evolution of the Microsoft ActiveX® Data Object (ADO) data access model that directly addresses user requirements for developing scalable applications. ADO.NET was designed specifically for the Web, keeping scalability, statelessness, and XML in mind. The ADO.NET version that comes with Visual Studio .NET 2003 is version 1.1, which is the version used in this guide. A new version of ADO.NET 2.0 has been released with the latest release of Visual Studio .NET 2005. More details on various new features of ADO.NET 2.0 is available at http://msdn2.microsoft.com/en-US/library/ex6y04yf.aspx.

ADO.NET uses some of the ADO objects, such as the Connection and Command objects, and introduces some new objects, such as DataSet, DataReader, and DataAdapter.

The OLE DB and SQL Server .NET Data Providers (**System.Data.OleDb** and **System.Data.SqlClient**), which are part of the .NET Framework, provide five basic objects. These objects are the Connection, Command, DataReader, DataSet, and DataAdapter.

- **Connections**. Used to connect to the database and manage the transactions against the database.
- **Commands**. Used to execute the SQL statements against a database.
- **DataSets**. Allow you to store the database schema and to program against flat data, XML data, and relational data.
- **DataReaders**. Allow you to read a forward-only stream of data records from a data source.
- **DataAdapters**. Used as a bridge between data source and data sets.

The records are mapped to the given commands accordingly. Figure 1.6 depicts the database interactions in ADO.NET.

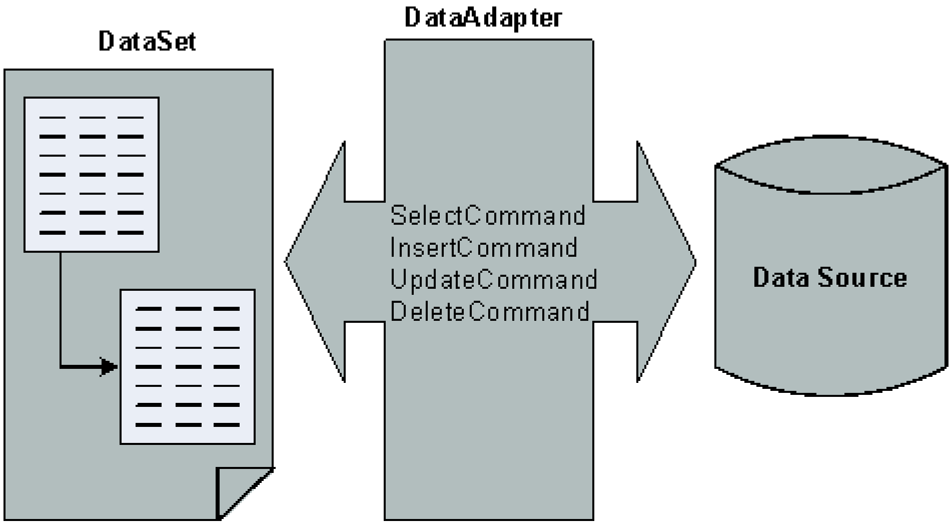


Figure 1.6. Database interactions

Three of these objects have been developed since Connections and Commands were introduced. The following sections describe these new objects: DataSet, DataReader, and DataAdapter.

DataSet

The following are major features of the DataSet object:

- DataSet is a stand-alone entity, which acts as a disconnected recordset that knows nothing about the source or destination of the data it contains. A DataSet contains entities such as tables, columns, relationships, constraints, and views.
- DataSet is a memory-resident representation of data that provides a consistent relational programming model independent of the data source.
- The DataSet has many XML characteristics, including the capability to produce and consume XML data and XML schemas. XML schemas can be used to describe schemas interchanged through Web services.
- A DataSet with a schema can actually be compiled for type safety and statement completion.
- The XML-based DataSet object provides a consistent programming model that works with all models of data storage including flat, relational, and hierarchical.
- The DataSet is independent of the source of its data; the managed provider has detailed and specific information. The managed provider connects, fills, and persists the DataSets across the datastores.

DataReader

The following are major features of the DataReader object:

- DataReader allows forward-only access over one or more of the resultsets obtained by executing a command and access to the column values within each row.
- Results are stored in the network buffer on the client after the execution of a query until you request them.
- DataReader increases application performance by retrieving data as soon as it is available instead of waiting for the entire results of the query to be returned.
- DataReader allows storage of only one row at a time in memory to reduce system overhead.

DataAdapter

The following are major features of the DataAdapter object:

- A DataAdapter is the object that connects to the database to fill the DataSet or DataReader and then connects back to the database to update the data, based on the operations performed while the DataSet held the data.
- DataAdapter represents a set of data commands and a database connection that are used to fill the DataSet and update the data source.
- DataAdapter serves as bridge between a DataSet and a data source for retrieving and saving data.
- DataAdapter has command object properties like InsertCommand, UpdateCommand, and DeleteCommand to update the data.

References

More information on XML is available at http://msdn.microsoft.com/xml/.

More information on SOAP is available at

http://msdn.microsoft.com/library/en-us/dnsoap/html/understandsoap.asp.

More information on CLR is available at

http://msdn.microsoft.com/library/en-us/cpguide/html/cpconthecommonlanguageruntime.asp.

More information on Windows Forms is available at

http://msdn.microsoft.com/netframework/programming/winforms/.

More information on ADO.NET is available at

http://msdn.microsoft.com/library/en-us/cpguide/html/cpconoverviewofadonet.asp.

More information on wrapping unmanaged classes with Managed Extensions for C++ is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmxspec/html/vcmanexmigrationguidepart1_start.asp.

More information on interop marshaling is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmex/html/vcgrfmanagedextensionsforcdatamarshalingtutorial.asp.

More information on .NET servers is available at

http://www.microsoft.com/windowsserversystem/default.mspx.

More information on .NET Framework security is available at http://msdn.microsoft.com/security/securecode/dotnet/default.aspx.

Chapter 2: Developing Phase: Process Milestones and Technology Considerations

This chapter provides information on starting the Developing Phase and provides an insight into the development environment for the migration exercise. It also discusses the major tasks and deliverables that should be identified and planned at the start of the Developing Phase.

Goals for the Developing Phase

This section describes the primary goals to be achieved in the Developing Phase. It acquaints you with the major tasks to be performed and the deliverables to be expected from the Developing Phase so that you can plan activities for your team accordingly.

The primary goal during the Developing Phase is to build the solution components—code as well as documentation. Apart from code development, the infrastructure is also developed during this phase, and all roles are active in building and testing the deliverables. The team continues to identify all risks throughout the phase and address new risks as they emerge.

Major Tasks and Deliverables

Table 2.1 describes the major tasks that you need to complete during the Developing Phase and lists the roles (owners) responsible for accomplishing them.

Table 2.1. Major Developing Phase Tasks and Owners

Major Tasks	Owners
Starting the development cycle The team begins the development cycle by verifying that all tasks identified during the Envisioning and Planning Phases have been completed.	Development
Building a proof of concept The team does a final verification of the concepts from the designs within an environment that mirrors production as closely as possible.	Development
Developing the solution components The team develops the solution using the core components and extends them to the specific needs of the solution. The team also develops and conducts unit functional tests to ensure that individual features perform according to the specification.	Development, User Experience, Test
Developing the testing tools and tests The team prepares the test cases to ensure the entire solution performs according to the specifications.	Test

Major Tasks	Owners
Building the solution A series of frequent builds culminate with major internal builds and signify points where the development team delivers key features of the solution. These builds are tested against the test cases to track the overall progress of the solution	Development, Test
Closing the Developing Phase The team completes all features, delivers the code and documentation, and considers the solution complete, thus entering the approval process for the Scope Complete Milestone.	Project

Starting the Development Cycle

During the Developing Phase, every component of the solution is analyzed in terms of how to apply code changes to adapt to the Microsoft® Windows® environment. This section focuses on providing you with the knowledge to identify and monitor the risks involved in the migration exercise.

The Developing Phase can be the most challenging part of any UNIX migration project. Major issues that are likely to affect the project will become evident in this phase. How these issues are resolved will determine whether the project schedules will change, whether funding will be sufficient, and whether the project will be successful.

The envisioning and planning techniques and tools introduced in Chapters 2 and 3 of Volume 1: *Plan* of this guide are designed to prepare for and insulate the project from many of the setbacks and delays that might occur during the Developing Phase. The capability to identify and mitigate risks is particularly helpful. A system that identifies, prioritizes, tracks, and mitigates risks is more useful during the Developing Phase of a project than any other phase.

Any unfinished tasks on the task lists from the Envisioning and Planning Phases could be a risk during the Developing Phase. Any assumptions made during the Planning and Envisioning Phases could also be potential risks in the Developing Phase because they could also bring up unforeseen challenges during the Developing Phase.

The prescriptive checklist to mitigate the risks in this phase is as follows:

- Contact the vendor to expedite delivery of the equipments as per the hardware requirement list.
- For the GUI aspect of the code migration, procure the necessary software licenses for the necessary X Server used in the UNIX application from the vendor before starting the project.
- Prepare a requirements specification document, providing a detailed design scope of the migration project, design and architecture to be followed, and a sign-off from the customer.
- If there are many change requests from the customer, perform an impact analysis of the changes and get a sign-off from the customer on the project execution approach for the requested changes.
- Provide necessary training to the project team that will aid in the proper and timely execution of the project.

Implementing these suggestions is likely to be much easier if the risks have been identified and the mitigation plans formulated and evaluated ahead of time. Risk mitigation, as part of the risk management process, can be used to keep a project on track through adverse situations.

Respecting the tenets and adhering to the procedures in the Microsoft Solutions Framework (MSF) Risk Management Discipline is key to ensuring that the project risks do not bring the project to a halt.

Building a Proof of Concept

Typically, the proof of concept is a continuation of the initial development work (the preliminary proof of concept) completed in the Planning Phase. The proof of concept tests key elements of the solution in a nonproduction simulation environment that mirrors the proposed operational environment. The team guides operations staff and users through the solution to validate that their requirements are met by the new solution.

The proof of concept is not meant to be production-ready although there may be some solution code or documentation that carries through the different phases to the eventual solution development deliverables. The proof of concept is considered a throw-away development that gives the team a final chance to verify the functional specification content and to address issues and mitigate risks before making the transition into development.

Interim Milestone: Proof of Concept Complete

Reaching this interim milestone marks the point where the team is fully moving from conceptual validation to building the solution architecture and components.

Developing the Solution Components

After successfully completing the proof of concept, the actual development of the solution components is started as per the specified solution architecture and design. Important prerequisites for the development activity, such as the migration approach, the technology to be used, the language to be used for editing and compilation, are decided before starting the Developing Phase in order to improve the efficiency of the development activity and reduce the time required to complete it.

The several technologies used in the UNIX application are identified and analyzed and the corresponding mechanisms in Windows are chosen to implement the migration process. The following chapters of this volume address these potential coding differences:

- Process management
- Thread management
- Memory management
- File management
- Infrastructure services
  - Signals and events
  - Exception handling
  - Sockets and networking
  - Interprocess communication
  - Daemons versus services
- Migrating the Graphical User Interface

You can then choose the solution that is most appropriate to your application and use these instructions as the basis for constructing your .NET code. This guide gives you sufficient information to choose the best method of converting the code to .NET. After you have made your choice, you can refer to the MSDN or .NET Framework documentation to ensure that you understand the technical details and implement them correctly. Throughout this volume, references are given for obtaining further information on the recommended coding changes—including code samples and URLs.

Using the Development Environment

The development environment is the environment in which the user develops and builds the solution. The development environment provides the necessary compiler, linker, libraries, and reference objects. In some cases, the integrated development environment (IDE) is also provided.

This section discusses the important components of the development environment for .NET applications. The two main components of the .NET development environment are the .NET Framework SDK and Visual Studio® .NET 2003.

.NET Framework SDK

The .NET Framework Software Development Kit (SDK) contains samples, compilers, and command-line tools designed to help you build applications and services based on .NET Framework technology. The SDK also provides documentation that includes an extensive class library reference, conceptual overviews, step-by-step procedures, tools information, and tutorials that demonstrate how to create specific types of applications. The .NET Framework SDK should be the essential reference for developers migrating UNIX applications to .NET. It contains the following major sections that provide critical information related to .NET:

- **Getting Started**. For those new to the .NET Framework technologies, the Getting Started section of the .NET Framework SDK documentation are designed to point you in the right direction.
- **QuickStarts, Tutorials, and Samples**. The .NET Framework SDK QuickStarts, tutorials, and samples are designed to quickly acquaint you with the programming model, architecture, and components that make up the .NET Framework.
- **Documentation.** The .NET Framework SDK documentation provides a wide range of overviews, programming tasks, and class library reference information to help you in migrating the applications to .NET.
- **Tools and Debuggers**. The .NET Framework SDK tools and debuggers enable you to create, deploy, and manage applications and components that target the .NET Framework.

The .NET Framework SDK is freely available for downloading from the MSDN Web site at http://msdn.microsoft.com/netframework/downloads/updates/default.aspx.

Visual Studio .NET 2003

Visual Studio .NET 2003 is a complete suite of development tools that gives you an ideal platform for migrating or building various enterprise applications such as Web applications, desktop applications, or Web services. In addition, you can use its powerful component-based tools and other technologies to simplify the design, development, and deployment of enterprise solutions. Visual Studio .NET 2003 supports such languages as Visual C++® .NET, Visual Basic® .NET, Visual C#® .NET, and Visual J#® .NET, all of which use the same integrated development environment (IDE).This allows them to share tools and facilitates in the creation of mixed-language solutions. These languages also take advantage of the functionality of the .NET Framework, which provides access to key technologies to simplify the migration process.

Visual Studio .NET contains a number of tools and technologies to facilitate the porting and upgrading of existing applications. It provides various methods for interoperating with existing application code, which results in a more efficient migration path requiring minimal code changes.

Some of these technologies are:

- COM+ Services
- .NET Enterprise Services
- P/Invoke methodology

Details of these technologies are described in Chapter 3, ".NET Interoperability" of this volume.

Developing the Testing Tools and Tests

After developing the solution components, you need to test the code changes made as a part of the development process. The testing process helps in identifying and addressing potential issues prior to deployment. Testing spans the Developing and the Stabilizing Phases. It starts when you begin developing the solution and ends in the Stabilizing Phase when the test team certifies that the solution components address the schedule and quality goals established in the project plan. This also involves using the automated test tools and test scripts.

Figure 2.1 depicts project testing processes in the Developing and Stabilizing Phases as per the Microsoft Solutions Framework (MSF) Process Model.

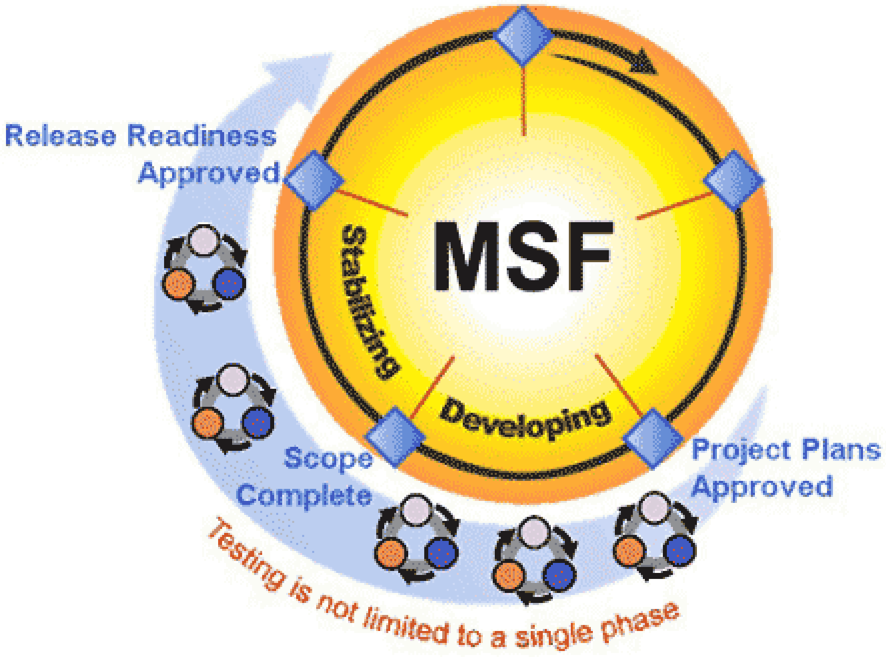

Release Readiness Approved
MSF
Stabilizing
Developing
Scope Complete
Project Plans Approved
Testing is not limited to a single phase

Figure 2.1. MSF Process Model - Testing processes across the Developing and Stabilizing Phases

Testing is performed, parallel to development, throughout the Developing Phase. This section discusses the unit testing activity that needs to be performed during the Developing Phase. The other necessary testing activities are discussed in Chapter 9, "Deployment Considerations and Testing Activities" and Chapter 10, "Stabilizing Phase" of this volume.

Testing in the Developing Phase is part of the build cycle, not a stand-alone activity. The development team designs, documents, and writes code and the test team performs unit testing and daily builds testing. The test team designs and documents test specifications and test cases, writes automated scripts, and runs acceptance tests on components submitted for a formal testing. The test team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components meet the project requirements. This process may be iterated several times based on the test results achieved from each testing activity.

In migration projects, testing is focused on finding the discrepancies between the behavior of the original application and that of the migrated application. In the migration of the infrastructure services, testing is focused on finding the discrepancies between the behavior of the original service, as seen by the clients, and the service that is exhibited by the newly migrated service.

All discrepancies must be investigated and fixed. It is recommended that you add new functionality to a migrated application or new capabilities to a migrated service in a separate project initiated after the migration project is complete.

Unit testing of the solution components is an integral part of the testing activity in the Developing Phase. Before starting with unit testing, a detailed code review is performed on the developed components and the review feedback is incorporated into the solution. Then the components are unit tested for their functionality.

Unit Testing

A unit may be a class, a program, or a specific functionality. Unit testing is part of the development process. Unit testing is the process of verifying that a specific unit of code functions according to its functional specifications and that it will be capable of interacting with other units as detailed in the specifications.

Unit testing in a migration project is the process of finding the discrepancies between the functionality and the output of individual units in the Windows application and the original UNIX application. However, this might not always be the case; in some cases, the application design in Windows may differ from the UNIX design. Therefore, it is important to identify units that are different in design from the UNIX units. Basic smoke testing, boundary conditions, and error tests are performed based on the functional specification of the unit.

The test cases for unit testing include constraints on unit inputs and outputs (pre-conditions and post-conditions), the state of the object (in case of a class), and the interactions between methods, attributes of the object, and other units.

Building the Solution

By this stage, the individual components of the solution are developed and tested in the Windows environment using .NET to satisfy the project requirements. This stage helps you build the solution with the developed and tested components and then make the migrated application ready for internal release.

As a good practice, MSF recommends that teams working on development projects perform daily builds of their solution. In migration projects, on the other hand, you typically have to examine large bodies of existing code to understand what they are intended for and to make changes to the code. However, code changes can happen only after addressing porting issues, hence daily builds may not be required. The process of creating interim builds allows a team to find issues early in the development process, which shortens the development cycle and lowers the cost of the project. Note that these interim builds are not deployed in the live production environment. Only when the builds are thoroughly tested and stable are they ready for a limited pilot release to a subset of the production environment. Rigorous configuration management is essential to keeping builds in synch.

Interim Milestone: Internal Release

The project needs interim milestones to help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward the completion of the solution feature sets and achievement of the associated quality level. Depending on the complexity of the solution, any number of internal releases may be required. Each internal release represents a fully functional addition to the solution’s core feature set that is potentially ready to move on to the Stabilizing Phase. As each new release of the application is built, fewer bugs must be reported and triaged. Each release marks a significant progress in the approach of the team toward deployment. With each new candidate, the team must focus on maintaining tight control on quality.

Chapter 3: Developing Phase: .NET Interoperability

As introduced in the “Interoperating with the Existing Code” section in Chapter 1, “Introduction to .NET” of this volume, Microsoft® .NET provides several interoperability mechanisms to interoperate with the unmanaged code, thus facilitating the migration process and preserving the investments in the existing code. This chapter discusses how these interoperability mechanisms can be applied to reuse the existing code with minimal changes. With this knowledge, you can evaluate the available options for migrating your UNIX application to Microsoft Windows® using .NET and choose the best migration approach to carry out the migration exercise. This chapter also provides appropriate examples that you can use as a basis for constructing your new .NET application.

.NET Interoperability Mechanisms

The various interoperability mechanisms provided by .NET are:

- Wrapping unmanaged C++ classes with Managed Extensions for C++.
- Platform Invocation services (P/Invoke).
- C++ Interoperability, popularly known as IJW (It Just Works).
- COM Interop services.

Note COM Interop services are not very useful for migrating from UNIX to .NET, and hence will not be covered at length in this guide.

Wrapping Unmanaged C++ Classes with Managed Extensions for C++

Managed Extensions for C++ supports the interoperation of code written in any .NET Framework-compliant language with the code already existing in C++. ("Unmanaged C++" refers to C++ that is compiled to the assembly language of a processor.) Interoperability is achieved by writing a proxy, or "wrapper," class in Managed Extensions for C++ for an unmanaged C++ class. A wrapper class interoperates with its unmanaged counterpart and serves as a managed proxy for it. It provides an API with a functionality that is similar to the unmanaged class. The API can be called by code written in any managed language.

If the C++ program on UNIX can be compiled on Windows with minimal code changes, then use this method to reuse the existing code in .NET. For example, low-level file access programs on UNIX can be compiled on Windows with a few header file changes. In such a case, consider using this method of interoperation.

The following C++ example code in UNIX writes to the standard output file descriptor. If any errors occur, it writes an error message to the standard error file descriptor. This program can be compiled on Windows by replacing the UNIX header file <unistd.h> with the Windows header file <io.h>; in this case, the technique of wrapping unmanaged classes is an ideal choice for migrating this code to .NET.

UNIX example: Code forwriting to the standard output

```
#include <unistd.h>
class StandardOutput
{
public:
StandardOutput(){}
void WriteToStandardOutput();
~StandardOutput() {}
};
void StandardOutput :: WriteToStandardOutput()
{
if ((write(1, "Here is some data\n", 18)) != 18)
      write(2, "A write error has occurred on file descriptor 1\n",46);
}
int main()
{
      StandardOutput *objSO = new StandardOutput();
      objSO->WriteToStandardOutput();
}
```

(Source File: U_WrapUnmanagedC++-UAMV4C3.01.cpp)

The following example shows the migrated Microsoft Win32® code for the UNIX example. As you can see, the header <unistd.h> has been replaced with <io.h>.

Windows example: Code for writing to the standard output

```
#include <io.h>
class StandardOutput
{
public:
StandardOutput(){}
void WriteToStandardOutput();
~StandardOutput() {}
};
void StandardOutput :: WriteToStandardOutput()
{
if ((write(1, "Here is some data\n", 18)) != 18)
            write(2, "A write error has occurred on file descriptor
1\n",46);
}
int main()
{
      StandardOutput *objSO = new StandardOutput();
      objSO->WriteToStandardOutput();
}
```

(Source File: W_WrapUnmanagedC++-UAMV4C3.01.cpp)

The following steps explain how the unmanaged class, **StandardOutput**, can be wrapped using a managed class, **MStandardOutput**.

To wrap the unmanaged class, StandardOutput, using a managed class, MStandardOutput

4. Write a managed wrapper class **MStandardOutput** for the unmanaged class **StandardOutput** and declare a single member of **MStandardOutput** for which the type is **`StandardOutput*`**. This is achieved by adding the __gc keyword before the **MStandardOutput** class, which indicates that it is garbage collected and that its lifetime is managed by the CLR.
5. Define an **MStandardOutput** constructor for each constructor of **StandardOuput**. This creates an instance of **StandardOutput** through the unmanaged new operator by calling the original constructor of **StandardOutput** class.
6. If the managed class **MStandardOutput** holds the only reference to the unmanaged class **StandardOutput**, define a destructor for **MStandardOutput**, which calls the delete operator on the member pointer to **StandardOutput**.
7. For each remaining method in **StandardOutput**, declare an identical method that delegates the call to the unmanaged version of the method in **StandardOutput**, performing any parameter marshaling if required.

The following Managed C++ code sample illustrates how the unmanaged class **StandardOutput** is wrapped.

Windows example: Code illustrating wrapping of unmanaged classes for writing to the standard output

```
#using <mscorlib.dll>
#include <io.h>

class StandardOutput
{
public:
void WriteToStandardOutput();
};
void StandardOutput :: WriteToStandardOutput()
{
      if ((write(1, "Here is some data\n", 18)) != 18)
            write(2, "A write error has occurred on file descriptor 
1\n",46);
}



__gc class MStandardOutput
{
private:
      StandardOutput *ob;
public:
MStandardOutput()
      {
            ob = new StandardOutput();
      }
~MStandardOutput()
      {
            delete(ob);
      }
void MWriteToStandardOutput()
      {
            ob->WriteToStandardOutput();
      }
};
int main()
{
      MStandardOutput *objSO = new MStandardOutput();
      objSO->MWriteToStandardOutput();
}
```

(Source File: N_WrapUnmanagedC++-UAMV4C3.01.cpp)

Wrapping Technique Considerations

The specific technique that is used for wrapping an unmanaged class depends on the semantics of the class. It may not be necessary to wrap all the member functions or data members of the unmanaged class. Wrap only the selected members of the unmanaged class, such as the members that must be accessed by managed objects.

Before wrapping an unmanaged C++ class, consider its structure and decide which members must be wrapped. Following are some simple guidelines for identifying members that need to be wrapped:

- If a member function or data member is private, then by design it is not meant to be accessed by other unmanaged classes. That member must not be accessible to managed objects either.
- Typically, helper functions are used internally by a class and are not designed to be accessed by other classes. These functions too must not be wrapped.

Note More information on wrapping unmanaged classes with Managed Extensions for C++ is available in Part I of the *Managed Extensions for C++ Migration Guide* at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmxspec/html/vcmanexmigrationguidepart1_start.asp.

Data Marshaling

When calling unmanaged functions that take arguments, the data must be marshaled to convert managed types to unmanaged types. A typical example is the conversion of the .NET Unicode string to the Win32 ANSI string. For each argument, you must copy the contents of the string from the common language runtime (CLR) heap into the C++ run-time heap and return a pointer to the string. The classes provided as part of the .NET Framework class library enable this. The **System::Runtime::InteropServices::Marshal** class contains a collection of methods to handle tasks, such as managed to unmanaged type conversions, unmanaged memory allocations, and copying of unmanaged memory blocks. The static methods defined in the **Marshal** class provide a method to convert between managed and unmanaged data. In general, the methods in the **Marshal** class return an **IntPtr**. This is a CLR pointer.

One approach is to use the **ToPointer()** member function of the **IntPtr** class. This returns a pointer of type **System::Void** ** that can be cast to **char** * as illustrated in the following code example.

```
String __gc* str = S"managed string";
char __nogc* pStr =
static_cast<char*>(Marshal::StringToHGlobalAnsi(str).ToPointer());
```

Platform Invocation Services

Platform Invocation services (also known as P/Invoke), provided by the .NET Framework CLR, enables managed code to call C-style functions in the existing unmanaged DLLs. P/Invoke can simplify customized data marshaling because the marshaling information is provided declaratively in attributes, instead of writing procedural marshaling code.

P/Invoke services first load the DLL containing the function into memory; they then locate the address of the function in the memory, stack up all the marshaled arguments, and transfer control to the unmanaged function. If any exceptions are raised in the unmanaged function, the functions are returned back to the managed caller for resolution. The P/Invoke functionality is also bidirectional and the Win32 API unmanaged functions can call back into the managed code.

Figure 3.1 shows the P/Invoke functionality.

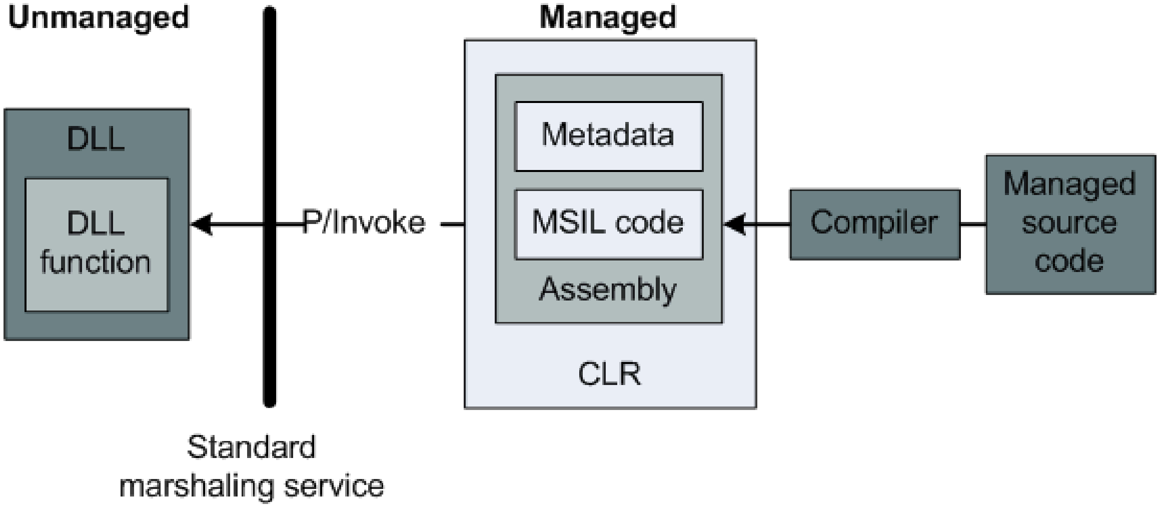


Figure 3.1. P/Invoke services

P/Invoke with UNIX Code

The following UNIX code sample in C reads characters from the standard input file descriptor and writes that information to the standard output file descriptor. If any input/output (I/O) errors occur, an error message is sent to the standard error file descriptor.

An example code in UNIX for using the standard input and output is shown as follows.

UNIX example: Using standard input and output

```
#include <unistd.h>
int main()
{
      char buffer[129];
      int num_read;

      num_read = read(0, buffer, 128);
      if (num_read == -1)
            write(2, "A read error has occurred\n", 26);
      if ((write(1,buffer,num_read)) != num_read)
            write(2, "A write error has occurred\n",27);
      exit(0);
}
```

(Source File: U_PInvoke-UAMV4C3.01.c)

Windows example: Using standard input and output through P/Invoke

```
#include <io.h>
void ReadandWrite()
{
      char buffer[129];
      int num_read;

      num_read = read(0, buffer, 128);
      if (num_read == -1)
            write(2, "A read error has occurred\n", 26);

      if ((write(1, buffer, num_read)) != num_read)
            write(2, "A write error has occurred\n", 27);
}
```

(Source File: W_PInvoke-UAMV4C3.01.cpp)

The following steps enable reuse of the existing code in your .NET application:

- Migrate the UNIX code in the preceding UNIX example to Win32 by changing the header file. The changed code is described in the preceding Windows example.
- Compile the changed code into a DLL (ReadandWrite.dll).
- After compiling into a DLL, the code can be accessed from any .NET project by using P/Invoke. The following code example shows how the changed code is accessed from a C# project using the DllImport attribute.

```
# using <mscorlib.dll>
using namespace System;
using namespace System.Runtime.InteropServices;

[DllImport("dllFileAccess",CharSet=CharSet::Ansi)]
extern void ReadandWrite();

int main()
{
   ReadandWrite();
}
```

(Source File: N_PInvoke-UAMV4C3.01.cpp)

P/Invoke with Win32 API

P/Invoke is also used for calling Win32 functions. A UNIX code that uses the **fork()** call to create a process can be easily migrated to Win32, using either the **CreateProcess()** or **_spawnlp()** calls.

The following code example illustrates the use of P/Invoke to call the Win32 API function **_spawnlp()** to create a Notepad process. The native function **_spawnlp()** is defined in msvcrt.dll. The DllImport attribute is used for the declaration of **_spawnlp().**

.NET example: Calling the Win32 API function _spawnlp through P/Invoke

```
# using <mscorlib.dll>
using namespace System;
using namespace System::Runtime::InteropServices;
[DllImport("msvcrt",CharSet=CharSet::Ansi)]
extern "C" int _spawnlp(int,[MarshalAs(UnmanagedType::LPStr)]
String*,[MarshalAs(UnmanagedType::LPStr)] String*,int);

int main()
{
      _spawnlp(1,S"notepad",S"notepad",0);
}
```

(Source File: N_PInvokeWin32API-UAMV4C3.01.cpp)

In the previous example, the **CharSet** parameter of the DllImport attribute specifies how the managed strings must be marshaled. In this case, they are marshaled to an ANSI string for the native side.

The **MarshalAs** attribute, located in the **System::Runtime::InteropServices** namespace, is used to specify the marshaling information for individual arguments on the native side. There are several choices for marshaling a managed **String** * argument, such as BStr, ANSIBStr, TBStr, LPStr, LPWStr, and LPTStr. The default is LPStr and it can be used to marshal other data types such as arrays.

C++ Interoperability – It Just Works

This technique is basically used to invoke unmanaged code from Managed C++ without using the DLLImport attribute. While migrating the UNIX code to Windows with minimal changes, the migrated code can also be compiled in a .NET environment using the /CLR switch for the compiler. The It Just Works (IJW) interoperability feature allows you to use the unmanaged APIs directly in managed code without having to use the DllImport attribute. This is done by including the header file and linking the import library. However, this feature is available only if the .NET programming language is Managed Extensions for C++.

The following example illustrates how the **_spawnlp()** function, used in the P/Invoke example, is implemented with IJW.

.NET example: Calling the Win32 API function _spawnlp

```
#using <mscorlib.dll>
using namespace System;
using namespace System::Runtime::InteropServices;

#include <process.h>
```

```
int main()
{
      String *pStr = S"notepad";
char *pChars = (char *)Marshal:StringToHGlobalAnsi(pStr).ToPointer();
      _spawnlp(1,pChars,pChars,0);
      Marshal::FreeHGlobal(pChars);
}
```

(Source File: N_IJW-UAMV4C3.01.cpp)

Explicit marshaling APIs return **IntPtr** types for 32-bit to 64-bit portability. Hence, you must use additional **ToPointer()** calls as shown in the earlier example.

The IJW mechanism is slightly faster because the IJW stubs do not need to check for the need to pin or copy data items as that is done by the developer. More importantly, if many unmanaged APIs need to be called using the same data, marshaling the APIs once up front and passing the marshaled copy around is more efficient than remarshaling APIs every time. However, you must specify the marshaling explicitly in your code. As the marshaling code is inline, it invades the flow of application logic.

If the migrated application mainly uses unmanaged data types and calls more unmanaged APIs than the .NET Framework APIs, then consider using the IJW feature. If the migrated application mainly uses managed data types and makes only occasional calls to the unmanaged APIs, then consider using P/Invoke with DllImport.

Marshaling Arguments

With P/Invoke, no marshaling is required between blittable types. Blittable types have the same representation in both the managed and the unmanaged world. For example, no marshaling is required between Int32 and int and Double and double.

Marshaling is required for types that do not have the same form, such as char, string, and struct types.

Table 3.1 lists the mappings used by the marshaler for various types.

Table 3.1. Data Type Mapping for Marshaler

Win32 Data Types in wtypes.h	C++	Managed Extensions	CLR
HANDLE	void *	void *	IntPtr, UIntPtr
BYTE	unsigned char	unsigned char	Byte
SHORT	Short	short	Int16
WORD	unsigned short	unsigned short	UInt16
INT	Int	Int	Int32
UINT	unsigned int	unsigned int	UInt32
LONG	Long	Long	Int32
BOOL	Long	Bool	Boolean
DWORD	unsigned long	unsigned long	UInt32
ULONG	unsigned long	unsigned long	UInt32
CHAR	Char	Char	Char

Win32 Data Types in wtypes.h	**C++**	**Managed Extensions**	**CLR**
LPSTR	char *	String * [in], StringBuilder * [in, out]	String [in], StringBuilder [in, out]
LPCSTR	const char *	String *	String
LPWSTR	Wchar_t *	String * [in], StringBuilder * [in, out]	String [in], StringBuilder [in, out]
LPCWSTR	const wchar_t *	String *	String
FLOAT	Float	Float	Single
DOUBLE	Double	double	Double

Chapter 4: Developing Phase: Process and Thread Management

This chapter discusses the differences between the UNIX and the Microsoft® .NET environments in the context of process and thread management programming. In addition, this chapter outlines the various options available for converting the code from the UNIX to the Windows® environment using .NET and illustrates these options with appropriate source code examples. You can use the information provided in this chapter to assist you in selecting the appropriate approach for migrating an existing application. You can also use the examples provided in this chapter as a basis for constructing your .NET application.

Process Management

The UNIX and Microsoft Windows operating systems provide process and thread management. Each process may have its own code, data, system resources, and state. Threads are part of processes and each process may have one or more threads running in them. Like processes, threads also have resource and state. When you know how UNIX and .NET differ in their management of processes, you can easily replace UNIX process routines with the corresponding .NET-compatible routines.

The UNIX and Windows process management models are very different from each other. The major difference lies in the creation of processes. When converting UNIX code to make it run on the .NET platform, you need to consider the following areas:

- Creating a process.
- UNIX processes versus .NET application domains.
- Processes versus threads.
- Managing process resources limits.

The following subsections describe these topics in detail.

Creating a Process

UNIX uses **fork** to create a new copy of a running process and **exec** to replace the executable file of a process with a new one.

.NET provides a **Process** class in the **System.Diagnostics** namespace to perform process-related operations. Using this component, you can create a process, obtain a list of processes running in the system, and access the processes running in the system (including the system processes). It is also useful for starting, stopping, controlling, and monitoring applications. A process' handle identifies it. This process handle is private to an application and cannot be shared. A process also has a process ID, which is unique and valid throughout the system. You can access the handle through the Handle property of the **Process** class, even after the process has exited. This enables you to access the administrative information about the process, such as the **ExitCode** (usually either zero for success or a nonzero error code) and the **ExitTime**.

.NET also provides application domains, which map to the processes in UNIX. In UNIX, an application runs within a process; whereas in .NET, an application runs within an application domain and there may be several such application domains running within the same process. Application domains are discussed in the "UNIX Processes vs. .NET Application Domains" section later in this chapter.

UNIX example: Creating a process using fork and exec

The following example code is a UNIX application that forks to create a child process and then runs the UNIX **ps** command by using **execlp**.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main()
{
pid_t pid;
printf("Running ps with fork and execlp\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
if (execlp("ps", NULL) < 0) {
perror("execlp failed");
exit(1);
}
break;
default:
break;
}
printf("Done.\n");
exit(0);
}
```

(Source File: U_CreatingProcess-UAMV4C4.01.c)

.NET example: Creating a process using System.Diagnostics namespace

The following Managed C++ example creates a Notepad process on Windows using the **System::Diagnostics::Process** class.

```
#using <mscorlib.dll>
#using <System.dll>
using namespace System;

int main()
{
      try
      {
//Creates an object of Process class
System::Diagnostics::Process *objPid = new
System::Diagnostics::Process();
//Starts a new notepad process
      objPid->Start(S"notepad.exe");
      Console::WriteLine(S"Done");
      } catch (Exception *e) {
            Console::WriteLine(S"Creating Process failed");
            Console::WriteLine(e->Message);
   }
}
```

(Source File: N_CreateProcess-UAMV4C4.01.cpp)

After creating and initializing an object of **Process** class, you can use it to obtain information about the running process. Such information includes the set of threads, the loaded modules (.dll and .exe files), and the performance information, such as the amount of memory the process is using.

The Microsoft Win32® API provides a **CreateProcess** function to create a process and then execute it. In theory, a P/Invoke call to the Win32 API **CreateProcess** can also be used to create a process in .NET. This is an unmanaged call, however, and **CreateProcess** can only be used to create unmanaged processes. The common language runtime (CLR) will not be capable of exercising any control over these processes.

UNIX Processes vs. .NET Application Domains

This section elaborates the differences between the UNIX processes and .NET application domains. It also discusses the advantages of using .NET application domains and provides instructions to create .NET application domains.

Operating systems and run-time environments generally provide some form of isolation between applications to ensure that the code running in one application does not affect other, unrelated applications. In UNIX, process boundaries are used to isolate applications running on the same computer. Each application is loaded into a separate process, which isolates the application from other applications running on the same computer. The applications are isolated because memory addresses are process-relative.

The .NET Framework further subdivides an operating system process into lightweight managed subprocesses, called application domains. These application domains provide isolation between applications. Application domains, which are typically created by run-time hosts, provide a more secure and versatile unit of processing. In .NET, the managed code must pass through a verification process before it can be run (unless the administrator has granted permission to skip the verification process).

The verification process determines that the code does not access invalid memory addresses or perform some other action that could cause the process in which it is running to fail. Code that passes this verification test is said to be type-safe. This capability to verify whether a piece of code is type-safe enables the CLR to provide a level of isolation that is equivalent to that of a process boundary, however, at a much lower performance cost.

Several application domains can run in a single process with the same level of isolation that exists in separate processes, but without the additional overhead of making cross-process calls or switching between processes. For example, a single browser process can run controls from several Web applications. However, these controls cannot access each other’s data and resources.

The following are the benefits offered by application domains:

- **Process isolation**. Faults in one application cannot affect other applications. The type-safe code cannot cause memory faults. Application domains ensure that code running in one domain cannot affect other applications in the process.
- **Operational flexibility**. You can terminate individual applications without terminating the entire process. Code running in a single application can be unloaded using application domains.
- **Restricted access to resources**. Code running in one application cannot directly access code or resources from another application. The CLR enforces this isolation by preventing direct calls between objects in different application domains.

In an application domain, you need to load assemblies before running the application. Running a typical application causes several assemblies to be loaded into its application domain. The code segment and data segment of the assembly are isolated from the application that uses it.

Application Domains and Threads

Application domains (AppDomains) provide an execution boundary for the managed code. Application domains play the role of an operating system process within the CLR, but with less overhead than operating system processes. Each AppDomain is hosted in a process, with more than one AppDomain allowed per process. Application domains are switched much faster than processes. A thread is an independent path of execution and is used by the CLR to execute code. During runtime, the run-time host loads the managed code into an application domain and a particular operating system thread runs that managed code. The operating system threads can switch application domains much faster than they switch processes.

One or more managed threads can run in one or any number of application domains within the same managed process. Although each application domain is started with a single thread, code in that application domain can create additional application domains and additional threads. Therefore, a managed thread can move freely between application domains within the same managed process. No one-to-one correlation exists between application domains and threads. Several threads can be executing in a single application domain at any given time, and a particular thread is not confined to a single application domain. That is, threads are free to cross application domain boundaries and a new thread is not created for each application domain.

At any given time, every thread is executing in one application domain. The runtime keeps track of different threads that are running in different application domains. You can locate the domain in which a thread is executing at any time by calling the **Thread.GetDomain** method.

Programming with Application Domains

Application domains are usually created and manipulated programmatically by run-time hosts. Hence, for most applications, you do not need to create your own application domain. However, you can create and configure application domains if you create your own run-time host application or if your application needs to create or work with additional application domains that are not automatically generated by the runtime. The **AppDomain** class is the programmatic interface to application domains. This class includes methods to create domains, to create instances of types in domains, and to unload domains. The following C# example code shows how to create an application domain.

.NET example: Creating an application domain

The following example creates an application domain using the **CreateDomain** method in the **System.AppDomain** class and executes the assembly at the specified location in that particular application domain.

```
using System;
using System.Reflection;
class AppDomain1
{
public static void Main()
{
 Console.WriteLine("Creating new AppDomain.");
 //Creating a new application domain named MyDomain
 AppDomain domain = AppDomain.CreateDomain("MyDomain");
 //Executing an assembly named MyProgram.exe in the application
domain
 domain.ExecuteAssembly("c:\\MyAssemblies\\MyProgram.exe");
 }
}
```

(Source File: N_ProcessVsAppDomain-UAMV4C4.01.cs)

Note More information and examples on application domains, such as configuring application domains, retrieving setup information from application domains, loading assemblies into application domains, obtaining information from assembly, and unloading application domains is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemappdomainclasstopic.asp.

Processes vs. Threads

This section will help you understand the programming differences between UNIX processes and .NET threads. In the following example, the UNIX code is forking a process, but not executing a separate run-time image. This creates a separate execution path within the application. In Windows, this is achieved by using threads instead of processes.

UNIX example: Code for forking executable

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main()
{
pid_t pid;
int n;
printf("fork program started\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
puts("I'm the child");
break;
default:
puts("I'm the parent");
break;
}
exit(0);
}
```

(Source File: U_Fork-UAMV4C4.01.c))

In .NET, the CLR uses application domains instead of processes. By using the appropriate methods in the .NET Framework class library, you can instantiate a new application domain and call a managed program. However, this is not a very efficient technique for performing multiple operations. Instead, consider using threads.

You can use the **System.Threading** namespace of .NET to create and manage threads, which operate within the bounds of the CLR. Threading in .NET is discussed in detail in the "Thread Management" section later in this chapter. The Win32 API provides a **CreateThread** method to achieve a similar functionality. A call to the Win32 API **CreateThread** is an unmanaged call, however, and the threads created by this method will operate outside the bounds of the CLR.

Managing Process Resource Limits

Developers often create processes that run with a specific set of resource restrictions. In some cases, they may impose the restrictions for stress testing or forced failure condition testing. In other cases, however, the limitations may be imposed to restrict runaway processes from using up all the available memory, CPU cycles, or disk space. In UNIX, the **getrlimit** function retrieves resource limits for a process, the **getrusage** function retrieves current usage, and the **setrlimit** function sets new limits. Table 4.1 lists the common limit names and their descriptions.

Table 4.1. Common Limit Names and Their Descriptions

Limit	Description
RLIMIT_CORE	The maximum size of a core file (in bytes) created by a process. If the core file is larger than RLIMIT_CORE, the write is terminated at this value. If the limit is set to 0, then no core files are created.
RLIMIT_CPU	The maximum CPU time (in seconds) that a process can use. If the process exceeds this time, the system generates SIGXCPU for the process.
RLIMIT_DATA	The maximum data segment size (in bytes) of a process. If the data segment exceeds this value, the functions **brk**, **malloc**, and **sbrk** will fail.
RLIMIT_FSIZE	The maximum size of a file (in bytes) created by a process. If the limit is 0, the process cannot create a file. If a write or truncation call exceeds the limit, further attempts will fail.
RLIMIT_NOFILE	The highest possible value for a file descriptor, plus one. This limits the number of file descriptors a process may allocate. If more files are allocated, functions allocating new file descriptors may fail with the error, EMFILE.

Limit	Description
RLIMIT_STACK	The maximum stack size (in bytes) of a process. The stack will not automatically exceed this limit; if a process tries to exceed the limit, the system generates SIGSEGV for the process.
RLIMIT_AS	The maximum total available memory (in bytes) for a process. If this limit is exceeded, the memory functions **brk**, **malloc**, **mmap**, and **sbrk** will fail with *errno* set to ENOMEM, and automatic stack growth will fail as described for RLIMIT_STACK.

In .NET, you can impose resource restrictions on a process by using the various properties and members of the **Process** class. Table 4.2 lists the properties and methods that you can use to enforce the restrictions.

Table 4.2. Process Class Properties and Their Description

Property	Description
MaxWorkingSet	Gets or sets the maximum allowable working set size for the associated process.
MinWorkingSet	Gets or sets the minimum allowable working set size for the associated process.
PriorityClass	Gets or sets the overall priority category for the associated process.
ProcessorAffinity	Gets or sets the processors on which the threads in this process can be scheduled to run.
Threads	Gets the set of threads that are running in the associated process.

Thread Management

This section describes the functionality of threads and their usage in the UNIX and .NET environments. In thread management, you need to consider the following core areas when migrating UNIX applications to the .NET environment:

- Creating a thread.
- Terminating a thread.
- Thread synchronization.
- Thread scheduling and priorities.

The following sections describe these areas in detail.

A thread is an independent path of execution in a process that shares the address space, code, and global data of the process. Time slices are allocated to each thread based on priority and consist of an independent set of registers, stack, input/output (I/O) handles, and message queue. Threads can usually run on separate processors on a multiprocessor computer. Win32 enables you to assign threads to a specific processor on a multiprocessor hardware platform.

An application using multiple processes usually must implement some form of interprocess communication (IPC). This can result in significant overhead and possibly a communication bottleneck. In contrast, threads share the process data among them and the interthread communication can be much faster. However, threads sharing data can lead to data access conflicts between multiple threads. You can address these conflicts using synchronization techniques, such as semaphores and mutexes.

In UNIX, developers implement threads by using the POSIX **pthread** functions. In .NET, developers implement threading by using the **System.Threading** namespace. The following section describes how you should go about converting UNIX threaded applications into .NET threaded applications. As discussed in the section on processes, you may also decide to convert some of the UNIX processes of an application into threads for better performance.

Note More information on thread management functions in the Threading namespace, **System.Threading**, is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreading.asp.

Creating a Thread

This section compares the UNIX and .NET functionalities for creating threads and provides examples. When creating a thread in UNIX, use the **pthread_create** function. This function has three arguments: a pointer to a data structure that describes the thread, an argument specifying the attributes (usually set to NULL indicating default settings) of the thread, and the function that the thread will run. The thread finishes execution with a **pthread_exit**, where (in this case) it returns a string. The process can wait for the thread to complete using the function **pthread_join**.

UNIX example: Creating a thread

The following code example shows how to create a thread in UNIX and waits for it to finish.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello World";

void *thread_function(void *arg) {
    printf("thread_function started. Arg was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("See Ya");
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n", (char
*)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}
```

(Source File: U_CreatingThread-UAMV4C4.01.c)

In .NET, the **Thread** class and its associates in the **System::Threading** namespace are used to create a thread.

To create and run a thread in .NET

8. Create a **ThreadStart** delegate that refers to the method that the thread would execute.
9. Create a **Thread** object using the **ThreadStart** delegate.
10. Call the start method of the **Thread** object.

Unlike UNIX, you must always start .NET threads explicitly after they are created. The **ThreadStart** delegate must refer to a **void** method that takes no parameters. This implies that you cannot start a thread using a method that takes parameters or obtain a return value from the method. To pass data to a thread, create an object to hold the data and the thread method, as illustrated by the code example that follows. To retrieve the results of a thread method, use a callback method.

.NET example: Creating a thread

The following Managed C++ example shows how to create a thread in the .NET Framework.

```
#using <mscorlib.dll>
 using namespace System;
 using namespace System::Threading;

 //The managed class ThreadExample
 public __gc class ThreadExample
 {
 private:
 String* tMessage;
 public:
 //Constructor of ThreadExample, which is used to pass arguments
 ThreadExample(String* argMessage)
 {
       tMessage = argMessage;
 }
 //The method that is executed as a seperate thread
 void thread_function()
 {
         Console::WriteLine(S"thread_function started");
        Console::WriteLine(S"Message is {0}",tMessage);
         Thread::Sleep(3000);
         tMessage = S"Bye!";
        Console::WriteLine(S"Message is {0}",tMessage);

 }
 };

 int main()
 {
       String *message = S"Hello World";
       ThreadExample *obTex = new ThreadExample(message);
       /*
Instantiates the Thread class to execute thread_function as a
seperate  thread
       */
Thread *oThread = new Thread(new
ThreadStart(obTex,&ThreadExample::thread_function));
        //Starts the thread
       oThread->Start();
       Console::WriteLine(S"Waiting for thread to finish...");
       oThread->Join();
       Console::WriteLine(S"Thread Joined");
       return 0;
 }
```

(Source File: N_CreatingThread-UAMV4C4.01.cpp)

Terminating a Thread

This section compares the UNIX and .NET functionalities for terminating threads and provides examples. UNIX uses the POSIX **pthread_cancel** function to terminate a thread. UNIX also offers facilities that allow a thread to specify if it is to be terminated immediately or deferred until it reaches a safe recovery point. Moreover, UNIX provides a facility known as cancellation cleanup handlers, which a thread can push and pop from a stack that is invoked in a last-in-first-out order when the thread is terminated. These cleanup handlers are coded to clean up and restore the resources before the thread is actually terminated.

UNIX example: Terminating a thread

The following code sample shows how to terminate a thread in UNIX.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg) {
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0) {
        perror("Thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0) {
        perror("Thread pthread_setcanceltype failed");
        exit(EXIT_FAILURE);
    }
    printf("thread_function is running\n");
    for(i = 0; i < 10; i++) {
        printf("Thread is running (%d)...\n", i);
        sleep(1);
    }
    pthread_exit(0);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(3);
    printf("Cancelling thread...\n");
    res = pthread_cancel(a_thread);
    if (res != 0) {
        perror("Thread cancellation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
```

```
}
```

(Source File: U_TerminatingThread-UAMV4C4.01.c)

In .NET, the **Thread.Abort** method is used to stop a thread permanently. When Abort is called, the CLR throws a **ThreadAbortException** in the thread on which Abort is invoked to begin the process of terminating the thread. Calling this method usually terminates the thread. The cleanup operations are handled by the CLR.

Thread.Abort does not abort the thread immediately. To ensure that the thread is stopped, you must call **Thread.Join** to wait on the thread. Join is a blocking call that does not return until the thread has actually stopped executing. After a thread is aborted, it cannot be restarted.

You can also call **Thread.Join** and pass a time-out period. If the thread dies before the timeout has elapsed, the call returns true. If the time expires before the thread dies, the call returns false. Other threads that call **Thread.Interrupt** can interrupt the threads that are waiting on a call to **Thread.Join**.

.NET example: Terminating a thread

The following sample code shows how to terminate a thread in .NET using the **Thread.Abort** method with managed C++.

```
#using <mscorlib.dll>
using namespace System;
using namespace System::Threading;

 //The managed class ThreadExample
 public __gc class ThreadExample
 {
 private:
 String* tMessage;
 public:
 //Constructor of ThreadExample, which is used to pass arguments
 ThreadExample(String* argMessage)
 {
       tMessage = argMessage;
 }
 //The method that is executed as a separate thread
 void thread_function()
 {
       try{
            Console::WriteLine(S"thread_function is running");
            Console::WriteLine(S"Message is {0}",tMessage);
             for(int i = 0; i < 10; i++)
            {
            Console::WriteLine(S"Thread is running
({0})...",i.ToString());
            Thread::Sleep(1000);
            }
            } catch(ThreadAbortException *e) {
```

```
            Console::WriteLine(e->Message);
            }
 }
 };

 int main()
 {
       String *message = S"Hello World";
      /*
Instantiates the Thread class to execute thread_function as a
seperate  thread
       */
      ThreadExample *obTex = new ThreadExample(message);
Thread *oThread = new Thread(new
ThreadStart(obTex,&ThreadExample::thread_function));
      oThread->Start();
      //Makes the main thread to sleep for 3 seconds
      Thread::Sleep(3000);
      Console::WriteLine(S"Cancelling Thread...");
      //Terminates the thread
      oThread->Abort();
      Console::WriteLine(S"Waiting for thread to finish...");
      oThread->Join();
      return 0;
 }
```

(Source File: N_TerminatingThread-UAMV4C4.01.cpp)

Suspend, Resume, and Sleep Methods

In .NET, the **Thread** class also provides a **Suspend** method, which can be used to temporarily halt the execution of a thread. The execution of the halted thread can be resumed by calling the **Resume** method. Use the **Sleep** method to stop execution of a thread for a short period and restart the thread when this period is over.

Thread Synchronization

This section describes usage of the thread synchronization mechanism and various thread synchronization techniques available in the UNIX and .NET environments. When more than one thread is executing simultaneously, you have to take the initiative to protect shared resources. For example, if the thread increments a variable, you cannot predict the result because the variable may have been modified by another thread before or after the increment. You cannot predict the result because the order in which threads access a shared resource is indeterminate. UNIX and Windows provide mechanisms, called synchronization techniques, for controlling resource access. These techniques are discussed in the following sections. The next section explains the indeterminate behavior of the threads when no synchronization techniques are used. The subsequent sections elaborate on how you can handle this scenario using the various synchronizing techniques.

Multiple Nonsynchronized Threads

The following example illustrates code that is, in principle, indeterminate. The parent represents the main thread in the example. It generates a “P” and the child or the secondary thread outputs a “T”.

Note This is a very simple example and, on most computers, the result would always be the same, but the important point to note is that this result is not guaranteed.

UNIX example: Threads with no synchronization

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg) {
    int count2;

    printf("thread_function is running. Argument was: %s\n",
(char *)arg);
    for (count2 = 0; count2 < 10; count2++) {
        sleep(1);
        printf("T");
    }
    sleep(3);
}

char message[] = "Hello I'm a Thread";

int main() {
    int count1, res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        sleep(1);
        printf("P");
    }

    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

(Source File: U_SyncTechnique-UAMV4C4.01.c)

.NET example: Threads with no synchronization

```
#using <mscorlib.dll>
 using namespace System;
 using namespace System::Threading;

 public __gc class ThreadExample
 {
 public:

     static void ThreadProc()
     {
         for (int i = 0; i < 10; i++)
         {
             Thread::Sleep(1000);
            Console::Write(S"T");
         }
        Thread::Sleep(3000);
     }
 };

 int main()
 {
     Thread *oThread = new Thread(new ThreadStart(0,
&ThreadExample::ThreadProc));
     oThread->Start();
     Console::WriteLine(S"entering loop");
     for (int i = 0; i < 10; i++)
     {
       Thread::Sleep(1000);
      Console::Write(S"P");
     }
     Console::WriteLine(S"Waiting for thread to finish");
     oThread->Join();
     Console::WriteLine(S"Thread Joined");
     return 0;
 }
```

(Source File: N_SyncTechnique-UAMV4C4.01.cpp)

In the UNIX example, access to **thread_function** has not been synchronized; and in the .NET example, the access to **ThreadProc** has not been synchronized. It is not possible to predict the output of these code examples. In most applications, unpredictable results are an undesirable feature. Therefore, it is important to control access to shared resources in threaded code. UNIX and .NET provide mechanisms for controlling resource access.

Synchronization Techniques

Synchronization techniques are used for the following reasons:

- To explicitly control the order in which code runs whenever tasks must be performed in a specific sequence.
- To prevent the problems that can occur when two threads share the same resource at the same time.

There are two approaches to synchronization: polling and using synchronization objects.

Note More information on polling is available at
http://msdn.microsoft.com/library/en-us/vbcn7/html/vaconthreadsynchronization.asp.

The next sections discuss the synchronization objects in detail.

Advanced Synchronization Techniques in .NET

The .NET Framework provides a number of objects that help in creating and managing multithreaded applications. **WaitHandle** objects help you respond to actions taken by other threads, especially when interoperating with unmanaged code. The **ThreadPool** provides the best basic thread creation and management mechanism for most tasks. **Monitor**, **Mutex**, **Interlocked**, and **ReaderWriterLock** objects provide mechanisms for synchronizing execution at a low level. **Timer** is a flexible way to raise activities at certain intervals and I/O asynchronous completion uses the thread pool to notify when the I/O work is completed, freeing you to do other things in the meantime.

Synchronization Using Interlocked Compare Exchange

The **Interlocked** methods **CompareExchange**, **Decrement**, **Exchange**, and **Increment** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The **Increment** and **Decrement** functions combine the operations of incrementing or decrementing the variable and checking the resulting value. This atomic operation is useful in a multitasking operating system in which the system can interrupt execution of one thread to grant a slice of processor time to another thread. Without such synchronization, one thread could increment a variable but be interrupted by the system before it could check the resulting value of the variable. A second thread could then increment the same variable. When the first thread receives its next time slice, it will check the value of the variable, which has now been incremented not once but twice. The **Interlocked** variable access functions prevent this kind of error.

The **Exchange** function atomically exchanges the values of the specified variables. The **CompareExchange** function combines two operations: comparing two values and storing a third value in one of the variables based on the outcome of the comparison. You can use **CompareExchange** to protect computations that are more complicated than simple increment and decrement. The following example demonstrates a thread-safe method that adds to a running total.

The following example code uses the **CompareExchange** method of the **Interlocked** class to synchronize the threads.

.NET example: Thread synchronization using Interlocked class

```
#using <mscorlib.dll>
 using namespace System;
 using namespace System::Threading;

 public __gc class ThreadExample
 {
 public:
       int run_now;
 public:
       ThreadExample()
       {
       }

       ThreadExample(int iRun)
       {
             run_now = iRun;
       }
 public:
    void ThreadProc()
     {
            Console::WriteLine(S"thread_function is running.");
      for (int i = 0; i < 10; i++)
       {
            int iLockCompare = 
Interlocked::CompareExchange(&run_now,1,2);
            if(iLockCompare ==2)
            {
              Console::Write(S"T-2");
            }
            else
            {
              Thread::Sleep(1000);
            }
      }
       Console::WriteLine(S"Child Thread Terminating");
      Thread::Sleep(3000);
     }
     void ParentThread()
     {
       Console::WriteLine(S"Main thread: Start a second
thread.");
 Thread *oThread = new Thread(new ThreadStart(this,
&ThreadExample::ThreadProc));
      oThread->Start();
      Console::WriteLine(S"entering loop");
      for (int i = 0; i < 10; i++)
      {
       int iLock = Interlocked::CompareExchange(&run_now,2,1);
       if(iLock==1)
       {
         Console::Write(S"P-1");
       }
       else
        Thread::Sleep(1000);
      }
      Console::WriteLine(S"Waiting for thread to finish");
      oThread->Join();
      Console::WriteLine(S"Thread joined");
     }
 };
 int main()
 {
       ThreadExample *objThreadEx = new ThreadExample(1);
       objThreadEx->ParentThread();
       return 0;
 }
```

(Source File: N_SyncTechnique-UAMV4C4.02.cpp)

Synchronization Using Semaphores

This section describes the usage of semaphore in UNIX applications and the implementation of the similar functionality in the .NET environment. In the following UNIX example, two threads are created that use a shared memory buffer. Access to the shared memory is synchronized using a semaphore. The primary thread (**main**) creates a semaphore object and uses this object to handshake with the secondary thread (**thread_function**). The primary thread instantiates the semaphore in a state that prevents the secondary thread from acquiring the semaphore while it is initiated.

After the user types in some text at the console and presses ENTER, the primary thread relinquishes the semaphore. The secondary thread then acquires the semaphore and processes the shared memory area. At this point, the main thread is blocked and is waiting for the semaphore; it will not resume until the secondary thread has given up control by calling **ReleaseSemaphore**. In UNIX, the semaphore object functions of **sem_post** and **sem_wait** are all that are required to perform the handshake.

UNIX example: Synchronizing threads using semaphores

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
sem_t bin_sem;

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("done", shared_area, 4) != 0) {
        printf("You input %d characters\n", strlen(shared_area) -
1);
        sem_wait(&bin_sem);
    }
    pthread_exit(NULL);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'done' to finish\n");
    while(strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        sem_post(&bin_sem);
    }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}
```

(Source File: U_SyncTechnique-UAMV4C4.03.c)

Semaphores are not readily available as part of the .NET Framework 1.1 class library that comes with Visual Studio .NET 2003. However, there are two ways to implement semaphores in .NET.

To implement semaphores with Win32 API using P/Invoke

1. Make P/Invoke calls to the semaphore functions in Win32, which are present in the header file <semaphore.h>. The following examples indicate how the Win32 semaphore functions present in kernel32.dll can be made available to the .NET classes using the **DllImport** attribute.

   CreateSemaphore

   ```
   [DllImport("Kernel32.dll", SetLastError = true)]
   static extern IntPtr CreateSemaphore( IntPtr
   lpSemaphoreAttributes,
   int lInitialCount, int lMaximumCount, string lpName );
   ```

   ReleaseSemaphore

   ```
   [DllImport("Kernel32.dll", SetLastError = true)]
   static extern bool ReleaseSemaphore( IntPtr hSemaphore, int
   lReleaseCount,
   out IntPtr lpPreviousCount );
   ```

   WaitForSingleObject

   ```
   [DllImport("Kernel32.dll", SetLastError = true)]
   static extern DWORD WaitForSingleObject(HANDLE hHandle,DWORD
   dwMilliseconds);
   ```

   With Win32, a combination of **WaitForSingleObject** and **ReleaseSemaphore** must be used in both the primary and the secondary threads to facilitate handshaking.

11. Write a semaphore class using the various existing synchronization objects that .NET Framework provides.

The following Managed C++ example shows how you can write a semaphore class in .NET by using the Win32 semaphore functions in kernel32.dll through **P/Invoke**.

.NET example: Implementing semaphore with Win32/Win64 API

```
#using <mscorlib.dll>
using namespace System;
using namespace System::ComponentModel;
```

```
using namespace System::Runtime::InteropServices;
using namespace System::Threading;

[DllImport("Kernel32.dll", CharSet=CharSet::Ansi)]
extern IntPtr __nogc* CreateSemaphore( IntPtr
lpSemaphoreAttributes,
int lInitialCount, int lMaximumCount, String* lpName );

[DllImport("Kernel32.dll", CharSet = CharSet::Ansi)]
 extern bool ReleaseSemaphore( IntPtr hSemaphore, int
lReleaseCount, IntPtr __nogc* lpPreviousCount );

public __gc __sealed class Semaphore : public WaitHandle
{
public:
      static int _thread = 0;
      Semaphore()
      {

      }
      Semaphore(int maxCount)
      {
Handle = CreateSemaphore(IntPtr::Zero, maxCount, maxCount, NULL);
               if(Handle == InvalidHandle)
               {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
               }
      }
      Semaphore(int initialCount, int maxCount)
      {
Handle = CreateSemaphore(IntPtr::Zero, initialCount, maxCount,
NULL);
              if(Handle == InvalidHandle)
              {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
               }
      }
      Semaphore(int maxCount, String* name)
      {
Handle = CreateSemaphore(IntPtr::Zero, maxCount, maxCount, name);
               if(Handle == InvalidHandle)
               {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
               }
      }
      Semaphore(int initialCount, int maxCount, String* name)
      {
Handle = CreateSemaphore(IntPtr::Zero, initialCount, maxCount,
name);
               if(Handle == InvalidHandle)
               {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
               }
      }

      int ReleaseWin32Semaphore()
      {
               void *p;
               IntPtr __nogc* previousCount = __nogc new
IntPtr(&p);
               if(!ReleaseSemaphore(Handle, 1, previousCount))
              {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
              }
              return previousCount->ToInt32();
      }
      int ReleaseWin32Semaphore(int count)
      {
               void *p;
               IntPtr __nogc* previousCount = __nogc new
IntPtr(&p);
               if(!ReleaseSemaphore(Handle, count,
previousCount))
               {
                  throw new
Win32Exception(Marshal::GetLastWin32Error());
               }
               return previousCount->ToInt32();
      }
     }
};
```

(Source File: N_SyncTechnique-UAMV4C4.03.cpp)

The latest release of .NET, Visual Studio .NET 2005, contains readymade classes for semaphores as part of the .NET Framework v2.0 class library. These semaphore classes allow you to use Win32 semaphores from managed code. More details on the semaphore class is available at http://msdn2.microsoft.com/en-US/library/system.threading.semaphore.aspx.

Synchronization Using Mutexes

A *mutex* is a kernel object that provides a thread with mutually exclusive access to a single resource. Any thread of the calling process can specify the mutex-object handle in a call to one of the **wait** functions. The single-object **wait** functions return when the state of the specified object is signaled. The state of a mutex object is signaled when no thread owns it. When the state of the mutex is signaled, one waiting thread is granted ownership. The state of the mutex changes to nonsignaled and the **wait** function returns. Only one thread can own a mutex at any given time. The owning thread uses the **ReleaseMutex** function to release its ownership.

The following example code illustrates the use of mutexes to coordinate access to a shared resource and to handshake between two threads. The logic is virtually identical to the semaphore example in the previous section. The only real difference is that this example uses a mutex instead of a semaphore.

UNIX example: Thread synchronization using mutexes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
pthread_mutex_t shared_mutex; /* protects shared_area */

void *thread_function(void *arg) {
    pthread_mutex_lock(&shared_mutex);
    while(strncmp("done", shared_area, 4) != 0) {
        printf("You input %d characters\n", strlen(shared_area) -
1);
        pthread_mutex_unlock(&shared_mutex);
        pthread_mutex_lock(&shared_mutex);
    }
    pthread_mutex_unlock(&shared_mutex);
    pthread_exit(0);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&shared_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_lock(&shared_mutex);
    printf("Input some text. Enter 'done' to finish\n");
    while (strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        pthread_mutex_unlock(&shared_mutex);
        pthread_mutex_lock(&shared_mutex);
    }

    pthread_mutex_unlock(&shared_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    pthread_mutex_destroy(&shared_mutex);
    exit(EXIT_SUCCESS);
}
```

(Source File: U_SyncTechnique-UAMV4C4.04.c)

In .NET, use **WaitHandle.WaitOne** to request ownership of a mutex. The thread that owns a mutex can request the same mutex in repeated calls to **Wait** without blocking its execution. However, the thread must call the **ReleaseMutex** method the same number of times to release ownership of the mutex. If a thread terminates normally while owning a mutex, the state of the mutex is set to signaled and the next waiting thread gets the ownership. If no one owns the mutex, the state of the mutex is signaled.

.NET example: Thread synchronization using mutexes

```
#using <mscorlib.dll>
using namespace System;
using namespace System::Threading;
__gc class Test
{
public:
      String* gStr;
```

```
public:
      Test()
      {
            gStr = S"";
      }
public:
    static Mutex* mut = new Mutex();
public:
    void UseResource()
    {
      mut->WaitOne();
      while(!gStr->Equals(S"done"))
      {
Console::WriteLine(S"The length of '{0}' is {1}", gStr,
__box(gStr->Length));
      mut->ReleaseMutex();
      mut->WaitOne();
      }
      mut->ReleaseMutex();
    }
    void Parent()
    {
      mut->WaitOne();
      Thread * myThread = new Thread(new ThreadStart(this,
Test::UseResource));
      myThread->Start();
      while(!gStr->Equals(S"done"))
      {
      gStr = Console::ReadLine();
      mut->ReleaseMutex();
      mut->WaitOne();
      }
      mut->ReleaseMutex();
      Console::WriteLine(S"Waiting for thread to finish...");
      myThread->Join();
      Console::WriteLine(S"Thread joined");
    }
};
int main()
{
      Test* objTest = new Test();
      Console::WriteLine(S"Input some text. Enter 'done' to
finish");
```

```
      objTest->Parent();
}
```

(Source File: N_SyncTechnique-UAMV4C4.04.cpp)

Note More information on the threading and synchronization objects in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-s/cpguide/html/cpconthreadingobjectsfeatures.asp.

Thread Scheduling and Priorities

This section describes the scheduling priority of a thread in UNIX and .NET. This section helps you convert the UNIX application with different thread priorities to .NET thread priorities. A thread in .NET can be assigned any one of the following five priority values:

- Highest
- AboveNormal
- Normal
- BelowNormal
- Lowest

The **Thread::Priority** property in .NET allows you to set or change the priority of the thread to any of the five priorities. The threads created within the CLR are initially assigned the priority of **ThreadPriority::Normal**. Threads created outside the runtime retain the priority that they had before they entered the managed environment.

The operating system uses the priority level of all the executable threads to determine which thread gets the next slice of CPU time. The scheduling algorithm used to determine the order in which the threads are executed varies with each operating system. UNIX offers both round-robin and FIFO (first-in-first-out) scheduling algorithms, whereas Windows uses only a round-robin algorithm. This does not mean that Windows is less flexible; it just means that any fine-tuning performed on thread scheduling in UNIX is implemented differently in Windows.

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. As long as a thread with a higher priority is available to run, lower priority threads are not executed. When there are no more executable threads at a given priority, the scheduler moves to the next lower priority and schedules the threads at that priority for execution. If a higher priority thread becomes executable, the lower priority thread is preempted and the higher priority thread is allowed to execute once again.

Managing Thread State and Priorities in .NET

The **Thread** class provides a number of members for managing the thread state and priorities. Some of these members are:

- **ThreadState**. Returns the current state of the thread. The initial thread state value is **Unstarted**. The other values are **Running**, **WaitSleepJoin**, **SuspendedRequested**, **Suspended**, and **Stopped**.
- **Priority**. Gets or sets the priority for the specified thread.
- **Name**. Gets or sets the name of the thread. If the name is not set, it returns null.
- **CurrentContext**. Gets the current context in which the thread is executing.
- **IsBackground**. Specifies whether the thread should execute in the background. Background threads are stopped automatically (if they are still running) when the program finishes. (Programs will wait for foreground threads to complete before terminating.)

Example of Converting UNIX Thread Scheduling into .NET

In this example, the thread priority level is set to the lowest level. For UNIX, lowering the thread priority level requires creating an attribute object before instantiating the thread, and then setting the policy of the attribute object. After this, the thread is created with the modified attribute. On successful instantiation of the thread, the priority level is adjusted to the lowest level within the designated policy and class. In UNIX, this is accomplished by a call to **pthread_attr_setschedparam**.

In .NET, the priority of the thread is set to lowest by setting the priority property to **ThreadPriority::Lowest.**

UNIX example: Thread scheduling

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello I’m a Thread";
int thread_finished = 0;

void *thread_function(void *arg) {
    printf("thread_function running. Arg was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting 
now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}

int main() {
    int count=0, res, min_priority, max_priority;
    struct sched_param scheduling_params;
    pthread_t a_thread;
    void *thread_result;
    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
    if (res != 0) {
        perror("Setting schedpolicy failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    max_priority = sched_get_priority_max(SCHED_OTHER);
    min_priority = sched_get_priority_min(SCHED_OTHER);
    scheduling_params.sched_priority = min_priority;
    res = pthread_attr_setschedparam(&thread_attr,
&scheduling_params);
    if (res != 0) {
        perror("Setting schedparam failed");
        exit(EXIT_FAILURE);
    }
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to finish (%d)\n", ++count);
        sleep(1);
    }
    printf("Other thread finished, See Ya!\n");
    exit(EXIT_SUCCESS);
}
```

(Source File: U_ThreadSchedule-UAMV4C4.01.c)

.NET example: Thread scheduling

```
#using <mscorlib.dll>
using namespace System;
using namespace System::Threading;
int thread_finished = 0;

public __gc class ThreadExample
{
 private:
 String* tMessage;
 public:
 ThreadExample(String* argMessage)
 {
       tMessage = argMessage;
 }
 void thread_function()
 {
       Console::WriteLine(S"thread_function running.");
       Console::WriteLine(S"Message is {0}",tMessage);
       Thread::Sleep(4000);
       Console::WriteLine(S"Second thread finished, setting flag,
and exiting now\n");
     thread_finished = 1;

 }
 };

 int main()
 {
     int count = 0;
       String *message = S"Hello! I am Thread";
       ThreadExample *obTex = new ThreadExample(message);
 Thread *oThread = new Thread(new
ThreadStart(obTex,&ThreadExample::thread_function));
       oThread->Start();
       oThread->Priority = ThreadPriority::Lowest;
       while(!thread_finished)
       {
             ++count;
Console::WriteLine("Waiting for the other thread to finish
({0})",count.ToString());
             Thread::Sleep(1000);
       }
       Console::WriteLine(S"Other thread finished, bye!");
       return 0;
   }
```

(Source File: N_ThreadSchedule-UAMV4C4.01.cpp)

Managing Multiple Threads

In the following UNIX example, numerous threads are created that terminate at random times. Their termination and display messages are then caught to indicate their termination status.

UNIX example: Managing multiple threads

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
#define NUM_THREADS 5

void *thread_function(void *arg) {
    int t_number = *(int *)arg;
    int rand_delay;

    printf("thread_function running. Arg was %d\n", t_number);
//  Seed the random-number generator with current time so that
//  the numbers will be different each time function is run.
    srand( (unsigned)time(NULL));
//  random time delay from 1 to 10
    rand_delay = 1+ 9.0*(float)rand()/(float)RAND_MAX;
    sleep(rand_delay);
    printf("See Ya from thread #%d\n", t_number);
    pthread_exit(NULL);
}

int main() {
    int res;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    int multiple_threads;

    for(multiple_threads = 0; multiple_threads < NUM_THREADS;
multiple_threads++) {
        res = pthread_create(&(a_thread[multiple_threads]), NULL,
thread_function, (void *)&multiple_threads);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    printf("Waiting for threads to finish…\n");
    for(multiple_threads = NUM_THREADS - 1; multiple_threads >=
0; multiple_threads--) {
        res = pthread_join(a_thread[multiple_threads],
&thread_result);
        if (res == 0) {
            printf("Another thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}
```

(Source File: U_MultipleThreads-UAMV4C4.01.c)

.NET example: Managing multiple threads using ThreadPool class

.NET provides a **ThreadPool** class to manage multiple threads. When you use the **ThreadPool** class, the thread management is done by the infrastructure.

```
#using <mscorlib.dll>
using namespace System;
using namespace System::Threading;
// TaskInfo holds state information for a task that will be
// executed by a ThreadPool thread.

public __gc class TaskInfo
{
    // State information for the task.
public:
    int Value;

    // Public constructor provides an easy way to supply all
    // the information needed for the task.
    TaskInfo(int number)
    {
        Value = number;
    }
};

__gc class Example
{
public:
// This thread procedure performs the task.
    static void thread_function(Object* stateInfo)
    {

      TaskInfo* ti = dynamic_cast<TaskInfo*>(stateInfo);
Console::WriteLine(S"Thread function running.The arg was {0}",ti-
>Value.ToString());
      Random *objRand = new Random();
      int rand_delay = 1 + (objRand->Next() % 10);
```

```
        Thread::Sleep(rand_delay*1000);
        Console::WriteLine(S"See ya from thread #{0}",__box(ti-
>Value));
    }
};

int main()
{
    for(int i=0;i<4;i++)
      {
            TaskInfo *ti = new TaskInfo(i);
ThreadPool::QueueUserWorkItem(new WaitCallback(0,
Example::thread_function),ti);
            Thread::Sleep(1000);
      }
      Console::WriteLine(S"Waiting for threads to finish");
      Thread::Sleep(10000);
      Console::WriteLine(S"All done");
      return 0;
}
```

(Source File: N_MultipleThreads-UAMV4C4.01.cpp)

Chapter 5: Developing Phase: Memory and File Management

This chapter discusses the programming differences between the Microsoft® .NET Framework and the UNIX environment in the following two categories:

- Memory management
- File management

In addition, this chapter outlines the various options available for converting the UNIX code to Microsoft .NET in each of these categories and illustrates the options with appropriate source code examples.

Memory Management

This section explains the process of memory allocation, de-allocation on the managed heap, and the garbage collection mechanism in the .NET environment. The following topics are explained in detail:

- Allocating memory
- Releasing memory
- Garbage collection
- Releasing unmanaged resources
- Thread local storage

UNIX provides the standard heap management functions for memory management. The standard C runtime on UNIX includes such functions as **calloc()** and **malloc()** for allocating memory and **free()** for de-allocating memory. The programmer, however, has to delete the references that are no longer required. Simply put, the programmer has to free the allocated memory when references to that memory are no longer required. Otherwise, it could lead to memory leaks.

In the .NET Framework, memory management is automatic. The garbage collector, which is a part of the common language runtime (CLR), manages the allocation and release of memory for managed code. Automatic memory management in .NET eliminates some common problems, such as forgetting to free a reference that causes memory leaks or attempting to access memory for an object that has already been freed. The following section describes how the garbage collector allocates and releases the memory for an application.

Allocating Memory

When a new process is initialized, the runtime reserves a contiguous region of address space for the process. This reserved address space is called the *managed heap*. The managed heap maintains a pointer to the address space, which will be allocated to the next object in the heap. Initially, this pointer is set to the base address of the managed heap. All reference types are allocated on the managed heap. When an application creates the first reference type, memory is allocated for the type at the base address of the managed heap. When the application creates the next object, the garbage collector allocates memory for it in the address space immediately following the first object. As long as address space is available, the garbage collector continues to allocate space for new objects in this manner.

Allocating memory from the managed heap is faster than the unmanaged memory allocation. Because the runtime allocates memory for an object by adding a value to a pointer, it is almost as fast as allocating memory from the stack. In addition, because new objects that are allocated address spaces consecutively are stored contiguously in the managed heap, an application can access the objects very quickly.

Releasing Memory

The optimizing engine of the garbage collector determines the best time to perform a collection based on the allocations made. When the garbage collector performs a collection, it releases the memory for the objects that are no longer being used by the application. How the garbage collector identifies the objects that are no longer being used and how the memory is released is discussed in detail in the “Garbage Collection” section later in this chapter.

To improve performance, the runtime allocates memory for large objects in a separate heap. The garbage collector automatically releases the memory allocated for large objects.

Releasing Memory for Unmanaged Resources

The garbage collector automatically performs the necessary memory management tasks for the majority of the objects that an application creates. However, unmanaged resources require an explicit cleanup. When you create an object that encapsulates an unmanaged resource, it is recommended that you provide the necessary code to clean up the unmanaged resource in a public **Dispose** method. A **Dispose** method enables users of an object to explicitly free the memory when they are finished with the object.

Garbage Collection

Each time the **new** operator is used to create an object, the runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate memory for new objects. However, memory is not infinite.

Eventually, the garbage collector must perform a collection to free some memory. The optimizing engine of the garbage collector determines the best time to perform a collection, based upon the allocations made. When the garbage collector performs a collection, it checks the managed heap for objects that are no longer being used by the application and performs the necessary operations to reclaim the memory.

The garbage collector determines which objects are no longer being used by examining the roots of the application. Every application has a set of roots and each root refers to an object on the managed heap. The runtime maintains a list of all the active roots. The garbage collector accesses the list to identify the unused objects, marks these objects for release, and releases the memory allocated for them.

Irrespective of the managed language used, the garbage collector of the .NET Framework provides automatic memory management. It allocates and releases the memory for the managed objects and, when necessary, executes the **Finalize** methods and destructors to properly clean up the unmanaged resources.

Automatic memory management simplifies development by eliminating the common bugs that arise from the manual memory management schemes. The following steps describe the life cycle of the object from its creation to destruction:

- **Type initialization**. When the first instance of an object is created, it executes any shared initialization and shared constructor code.
- **Instance initialization**. When an instance of your component is created, data members that have initialization code are initialized, and the appropriate constructor overload is executed.
- **Disposing of resources**. If the object overrides the **Dispose** method, it frees all system resources it may have allocated, releases references to other objects, and renders itself unusable.
- **Instance destruction**. When garbage collection detects that there are no remaining references to the component, the runtime calls your component's destructor and frees the memory.

Forcing Garbage Collection

The garbage collection GC class provides the **GC.Collect** method, which is used to give an application some direct control over the garbage collector. In general, avoid calling any of the collect methods and allow the garbage collector to run independently. In most cases, the garbage collector is better at determining the best time to perform a collection. However, in certain rare situations, forcing a collection might improve the performance of an application.

Use the **GC.Collect** method in a situation where there is a significant reduction in the amount of memory being used at a defined point in the application code. For example, an application might use a document that references a significant number of unmanaged resources. When the application closes the document, the resources the document has been using are no longer needed. To improve the performance of the application, consider releasing the unused resources.

Note More information on GC.Collect method is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemgcclasscollecttopic.asp.

Releasing Unmanaged Resources

The most common type of an unmanaged resource is an object that wraps an operating system resource, such as a file, a window, or a network connection. Although the garbage collector can track the lifetime of an object that encapsulates an unmanaged resource, it cannot clean up the resource. For these types of objects, the .NET Framework provides the **Object.Finalize** method, which allows an object to clean up its unmanaged resources properly when the garbage collector reclaims the memory used by the object. In C#, finalizers are expressed using the destructor syntax.

To properly dispose of the unmanaged resources, consider implementing a public **Dispose** method, which executes the necessary cleanup code for the object. The **IDisposable** interface in the **System** namespace provides the **Dispose** method for the resource classes that implement the interface. This method is public and hence the users of an application can call the **Dispose** method directly to free the memory used by the unmanaged resources. If a **Dispose** method is properly implemented, the **destructor** or the **finalize** method becomes a safeguard to clean up the unmanaged resources in case the **Dispose** method is not called.

The **Dispose** method of an object type should release all the resources that it owns. In addition, it should also release all the resources owned by its base types by calling the **Dispose** method of its parent type. The **Dispose** method of the parent type should release all the resources that it owns and in turn call the **Dispose** method of its parent type.

A **Dispose** method calls the **GC.SuppressFinalize** method for the object that it is disposing. If an object is currently on the finalization queue, **GC.SuppressFinalize** prevents the **Finalize** method from being called. Executing the **Finalize** method affects performance. Therefore, if the **Dispose** method has already done the work to clean up an object, then it is not necessary for the garbage collector to call the **Finalize** method of the object.

The **BaseResource** class of the .NET Framework implements the **IDisposable** interface and defines a public **Dispose** method. The cleanup code for the object is executed in this **Dispose** method. The **Dispose** method takes either a true or a false as an argument depending on the identity of the caller. If the argument is true, it means that the method has been called by the code; hence, all the managed and unmanaged resources can be disposed. If the argument is false, it means that the method has been called from the runtime; hence, only the unmanaged resources can be disposed. The **BaseResource** class also provides a destructor as a safeguard mechanism in case the **Dispose** method is not called.

The following example code illustrates a design pattern for implementing the **Dispose** method for classes that encapsulate unmanaged resources. This pattern is implemented throughout the .NET Framework.

In this example, the class **MyResourceWrapper** illustrates how to derive from a class that implements resource management using the **Dispose** method. **MyResourceWrapper** overrides the virtual **Dispose(bool)** method and provides clean-up code for the managed and unmanaged resources that it creates. **MyResourceWrapper** also calls the **Dispose** method on its base class, **BaseResource,** to ensure that its base is properly cleaned up.

.NET example: Code for implementing the Dispose method

```
using System;
using System.ComponentModel;
// Design pattern for the base class.
// By implementing IDisposable, you are announcing that instances
// of this type allocate scarce resources.
public class BaseResource: IDisposable
{
   // Pointer to an external unmanaged resource.
   private IntPtr handle;
   // Other managed resource this class uses.
   private Component Components;
   // Track whether Dispose has been called.
   private bool disposed = false;

   // Constructor for the BaseResource object.
   public BaseResource()
   {
      // Insert appropriate constructor code here.
   }

   // Implement IDisposable.
   // Do not make this method virtual.
   // A derived class should not be able to override this method.
   public void Dispose()
   {
      Dispose(true);
      // Take yourself off the Finalization queue
      // to prevent finalization code for this object
      // from executing a second time.
      GC.SuppressFinalize(this);
   }

   // Dispose(bool disposing) executes in two distinct scenarios.
   // If disposing equals true, the method has been called 
directly
   // or indirectly by a user's code. Managed and unmanaged 
resources
   // can be disposed.
   // If disposing equals false, the method has been called by 
the
   // runtime from inside the finalizer and you should not 
reference
   // other objects. Only unmanaged resources can be disposed.
   protected virtual void Dispose(bool disposing)
   {
      // Check to see if Dispose has already been called.
      if(!this.disposed)
      {
         // If disposing equals true, dispose all managed
         // and unmanaged resources.
         if(disposing)
         {
            // Dispose managed resources.
            Components.Dispose();
         }
         // Release unmanaged resources. If disposing is false,
         // only the following code is executed.
         CloseHandle(handle);
         handle = IntPtr.Zero;
         // Note that this is not thread safe.
         // Another thread could start disposing the object
         // after the managed resources are disposed,
         // but before the disposed flag is set to true.
         // If thread safety is necessary, it must be
         // implemented by the client.

      }
      disposed = true;
   }

   // Use C# destructor syntax for finalization code.
   // This destructor will run only if the Dispose method
   // does not get called.
   // It gives your base class the opportunity to finalize.
   // Do not provide destructors in types derived from this
class.
   ~BaseResource()
   {
      // Do not recreate Dispose clean-up code here.
      // Calling Dispose(false) is optimal in terms of
      // readability and maintainability.
      Dispose(false);
   }

   // Allow your Dispose method to be called multiple times,
   // but throw an exception if the object has been disposed.
   // Whenever you do something with this class,
   // check to see if it has been disposed.
   public void DoSomething()
   {
      if(this.disposed)
      {
         throw new ObjectDisposedException();
      }
   }
public void CloseHandle(IntPtr h)
      {
            //cleanup of handle
            // Write code here to cleanup your unmanaged resource
      }
      public void CloseHandle(NativeResource n)
      {
            CloseHandle(n.handle);

      }
}
public class ManagedResource:BaseResource
{
};
public class NativeResource:BaseResource
{

};
// Design pattern for a derived class.
// Note that this derived class inherently implements the
// IDisposable interface because it is implemented in the base
class.
public class MyResourceWrapper: BaseResource
{
   // A managed resource that you add in this derived class.
      private ManagedResource addedManaged = new
ManagedResource();
// A native unmanaged resource that you add in this derived
class.
      private NativeResource addedNative = new NativeResource();

   private bool disposed = false;

  // Constructor for this object.
   public MyResourceWrapper()
   {
      // Insert appropriate constructor code here.
   }

   protected override void Dispose(bool disposing)
   {
      if(!this.disposed)
      {
         try
         {
            if(disposing)
            {
               // Release the managed resources you added in
               // this derived class here.
               addedManaged.Dispose();
            }
            // Release the native unmanaged resources you added
            // in this derived class here.
            CloseHandle(addedNative);
            this.disposed = true;
         }
         finally
         {
            // Call Dispose on your base class.
            base.Dispose(disposing);
         }
      }
   }
}

// This derived class does not have a Finalize method
// or a Dispose method without parameters because it inherits
// them from the base class.
```

(Source File: N_MemMgt-UAMV4C5.01.cs)

Thread Local Storage

The Thread Local Storage (TLS) mechanism enables storing of data in a thread and accessing the data anywhere the thread exists. The **System.Threading** namespace allows developers to use TLS within their multithreaded applications.

Whenever a process is created, the CLR allocates a multislot data store array to each and every process. Threads, with the help of the flexible access methods, can use these data slots within the data stores to store and retrieve information that is unique to a thread and an application. There are two types of data slots: named slots and unnamed slots. The named slots can use a mnemonic identifier. However, other components can, intentionally or unintentionally, modify them by using the same name for their own thread-relative storage. However, if an unnamed data slot is not exposed to other code, it cannot be used by any other component. To use managed TLS, create a data slot using **Thread.AllocateNamedDataSlot** or **Thread.AllocateDataSlot**, and use the appropriate methods to set or retrieve the information placed there.

The following is an example of a console application in .NET illustrating the use of TLS for storing information specific to a thread. A dispatcher object is created, which keeps calling the **ProcessSignal()** method of the receiver in a loop. The receiver object generates a random number for every call from the dispatcher and writes this number on the TLS, which the dispatcher can read.

.NET example: Using Thread Local Storage

```
using System;
using System.Threading;
namespace TLSExample
{
// This Console Application is to demonstrate how TLS can be used
to store
// information specific to a thread

class TLSExample
{
      public static void Main(string[] args)
      {
            // Allocate a named data slot on all threads

            Thread.AllocateNamedDataSlot ("receivervalue");
            Despatcher desOb =new Despatcher ();
            // Thread out to the other classes
            ThreadStart myThreadStart = new ThreadStart
(desOb.DespatchWork);
            Thread myThread = new Thread(myThreadStart);
            // Start the thread here.
            myThread.Start();
            // free the memory on all threads
            Thread.FreeNamedDataSlot("receivervalue");

      }
}

public class Despatcher
{
      // This is class implementation that despatches down into
other classes
          private Receiver recOb;
      public Despatcher()
      {
            recOb = new Receiver();
      }
      public void DespatchWork()
      {
            // Looping to simulate processing
            for(int i=0;i<50;i++)
            {
            // The Despatcher despatches to receiver by calling
            // the processSignal method of the Receiver
            recOb.ProcessSignal();

            //After processing by the Receiver, the calling class
needs to
            //access the information on TLS
            int valueFromReceiver = DetermineVFR();
Console.WriteLine ("The value pulled from TLS " +
valueFromReceiver);
            }
      }
            public int DetermineVFR()
      {
            LocalDataStoreSlot vfcTLS;
            vfcTLS = Thread.GetNamedDataSlot("receivervalue");
            int vfc = (int) Thread.GetData(vfcTLS);
            return(vfc);
      }


}
```

```
public class Receiver
{

// This class processes work from the despatcher and returns back
to
// despatcher with information on thread

      private Random ranTime;
      private int max;
      public Receiver()
      {
            ranTime = new Random();
            max = 500;
      }
      public bool ProcessSignal()
      {
      // Generate some random number to store different values on
all threads
      int rndValue=ranTime.Next(max);

      LocalDataStoreSlot myData;
      myData = Thread.GetNamedDataSlot("receivervalue");
      // Set the named data slot equal to the random number
created above
      Thread.SetData(myData,rndValue);
      return true;
      }

}
}
```

(Source File: N_MemMgt-UAMV4C5.02.cs)

File Management

This section details the file management techniques in the UNIX and .NET environments. The following topics are explained in detail:

- File access mechanisms
- File open and access modes
- Migrating using interoperability strategies
- Working with directories

Every program that runs from the UNIX shell opens three standard files: standard input, standard output, and standard error. These files have the integer file descriptors and provide the primary means of communication between the programs. These file descriptors are 0, 1, and 2 respectively for standard input, standard output, and standard error. These files exist as long as the process runs. UNIX provides two kinds of file access: low-level file access and standard, or stream, file access.

File Access Mechanisms

This section describes the various file access mechanisms using low-level file input/output routines and the stream file access routines on the UNIX and .NET environments.

UNIX File Access

UNIX file access is mainly classified as two types: low-level file access and stream file access.

Low-Level File Access

The low-level input/output (I/O) functions invoke the operating system more directly for low-level operations than that provided by standard (or stream) I/O. Function calls relating to low-level input and output do not buffer or format data. They deal with bytes of information, which means that you are using the binary files instead of the text files. The low-level file handles or file descriptors, which give a unique integer number to identify each file, are used instead of file pointers.

Stream File Access

The standard, or stream, I/O functions process data in different sizes and formats, ranging from a single character to large data structures. They also provide buffering, which can improve performance. Using the stream file access functions, you can open a file either in the binary mode or in the text mode. In the binary mode, a program can access every byte in the file, whereas the text mode is normally used for text files in which some characters may be "hidden" from the program.

.NET File Access

In .NET, the **System::IO** namespace provides a **FileStream** method to access the contents of a file. If a low-level file access or the binary mode in a stream file access is used in UNIX, then consider using the **BinaryReader** and **BinaryWriter** classes in .NET. These .NET classes enable you to read from and write into binary files. The **System::IO** namespace also provides the **StreamReader** and **StreamWriter** classes for processing text files, which can be used for migrating the nonbinary mode file access code.

File Access Through the FileStream Class

The **FileStream** class provides access to files, including the standard input, output, and error devices. Use the **FileStream** class to read from, write to, open, and close files on a file system, as well as to manipulate other file-related operating system handles such as pipes, standard input, and standard output. The **FileStream** class also buffers input and output for better performance.

There are different types of **FileStream** constructors that you can use depending on your requirement. One common usage is as follows.

```
FileStream (“File name”, FileMode, FileAccess)
```

You can also use the **FileInfo** class, which helps in the creation of **FileStream** objects.

Note More information on the usage of the **FileStream** class in the **System.IO** namespace is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemIOFileStreamClassTopic.asp.

File Access Through BinaryReader and BinaryWriter

The **BinaryReader** class is used for reading strings and elementary data types as binary values, whereas the **BinaryWriter** class is used for writing elementary types in binary data. When writing to files, an application may have to create a file if the file to which the application is trying to write does not exist. To do so, the application requires permission for the directory in which the file is to be created. However, if the file already exists, the application only requires write permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant write permission to that file instead of granting permission to the entire directory. It is also more secure to write data to the user directories than to the root directory or the Program Files directory.

File Access Through StreamReader and StreamWriter

The **System.IO** namespace class allows you to read and write characters to and from files as streams, or contiguous groups of data, using specific encoding to convert characters to and from bytes. It includes the **StreamReader** and **StreamWriter** classes, which enable you to read or write a sequential stream of characters to or from a file. The **StreamReader** and **StreamWriter** classes mirror the functionality of the **BinaryReader** and **BinaryWriter** classes, but they read and write information as text rather than as binary data.

The following code sample reads up to 1 KB of characters from the input file and writes that information to the output file. If any input/output errors occur, an error message is output to the standard error file descriptor.

UNIX example: Reading and writing files using low-level functions

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char block[1024];
    int in, out;
    int num_read;

    in = open("input_file", O_RDONLY);
    if (in == -1) {
        write(2, "An error has occurred opening the file:
‘input_file’\n", 52);
        exit(1);
    }
    out = open("output_file", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    if (out == -1) {
        write(2, "An error has occurred opening the file:
‘output_file’\n", 53);
        exit(1);
    }
    while((num_read = read(in,block,sizeof(block))) > 0)
        write(out, block, num_read);

    exit(0);
}
```

(Source File: U_FileMgt-UAMV4C5.01.c)

The Managed C++ code sample reads up to 1 KB of characters from the input file and writes them to the output file.

.NET example: Reading and writing files using BinaryReader and BinaryWriter

```
#using <mscorlib.dll>
#define BUF_SIZE 1024

using namespace System;
using namespace System::IO;

int main(int argc, char *argv[])
{
  try
  {
    Byte block[] = new Byte[BUF_SIZE];
    FileStream *in =
        new FileStream("input_file", FileMode::Open,
FileAccess::Read);
    FileStream *out =
        new FileStream("output_file", FileMode::Create,
FileAccess::ReadWrite);

    BinaryReader *source = new BinaryReader(in);
    BinaryWriter *dest = new BinaryWriter(out);

    while (int numBytes = source->Read(block, 0, BUF_SIZE))
    {
      dest->Write(block, 0, numBytes);
    }

    source->Close();
    dest->Close();
    in->Close();
    out->Close();
  }
  catch (Exception *e)
  {
    Console::WriteLine(e->get_Message());
  }
}
```

(Source File: N_FileMgt-UAMV4C5.01.cpp)

The following code sample reads characters from the input file opened with the standard file I/O library function **fopen()** and writes that information to the output file, also opened with **fopen()**. Then it uses a loop of the **fgetc** and **fputc** calls to transfer the contents of "input_file" to "output_file".

UNIX example: Copying a file using stream file functions

```
#include <stdio.h>
int main()
{
    int c;
    FILE *in, *out;

    in = fopen("input_file","r");
    if (in == NULL) {
        write(2, "An error has occurred opening the file:
‘input_file’\n", 52);
        exit(1);
    }
    out = fopen("output_file","w");
    if (out == NULL) {
        write(2, "An error has occurred opening the file:
‘output_file’\n", 53);
        exit(1);
    }

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    fclose(in);
    fclose(out);
    exit(0);
}
```

(Source File: U_FileMgt-UAMV4C5.02.c)

.NET provides a simpler way of copying the contents of one file into another. The following code sample uses the **ReadToEnd()** method of the **StreamReader** class to read the contents of the input file to a string and later writes the read string to the output file.

.NET example: Copying files using StreamReader and StreamWriter

```
#using <mscorlib.dll>
using namespace System;
using namespace System::IO;

int main()
{
  try
  {
    FileStream *in =
        new FileStream("input_file", FileMode::Open,
FileAccess::Read);
    FileStream *out =
        new FileStream("output_file", FileMode::Create,
FileAccess::ReadWrite);


    StreamReader *source = new StreamReader(in);
    StreamWriter *dest = new StreamWriter(out);
    String *str = source->ReadToEnd();
    dest->Write(str);
    source->Close();
    dest->Close();
    in->Close();
    out->Close();
  }
  catch (Exception *e)
  {
    Console::WriteLine(e->get_Message());
  }
}
```

(Source File: N_FileMgt-UAMV4C5.02.cpp)

File Open and Access Modes

Table 5.1 lists the file access modes used in the **fopen()** statement in UNIX and its equivalent in the .NET Framework.

Table 5.1. Mapping of File Modes in fopen() to System::IO Namespace

File Modes in fopen()	File Modes in System::IO	Operation
r	FileMode::Open FileAccess:Read	Opens a text file for reading.
w	FileMode::OpenOrCreate FileAccess:Write	Opens a text file for writing. If the file does not exist, a new file is created.
a	FileMode::Append FileAccess::Write	Opens a text file for appending. The text that is written to is added at the end of the file. If the file does not exist, a new file is created.
rb	FileMode::Open FileAccess::Read Use the **BinaryReader** class for reading.	Opens a binary file for reading.
wb	FileMode::OpenOrCreate FileAccess::Write Use the **BinaryWriter** class for writing.	Opens a binary file for writing. If the file does not exist, a new file is created.
ab	FileMode::Append FileAccess::Write Use the **BinaryWriter** class for writing.	Opens a binary file for appending. If the file does not exist, a new file is created.
r+	FileMode::Open FileAccess::ReadWrite	Opens a text file for reading and writing.
w+	FileMode::OpenOrCreate FileAccess::ReadWrite	Creates a new text file for reading and writing. If a file with the same name already exists, it is over-written. If the file does exist, a new file is created.
a+	Append access can be requested only in write mode. To write use the following: FileMode::Append FileAccess::Write	Opens a text file for reading and appending.

File Modes in fopen()	File Modes in System::IO	Operation
r+b	FileMode::Open FileAccess::ReadWrite Use the **BinaryReader** and **BinaryWriter** classes for reading and writing.	Opens a binary file for reading and writing.
w+b	FileMode::OpenOrCreate FileAccess::ReadWrite Use the **BinaryReader** and **BinaryWriter** classes for reading and writing.	Creates a binary file for reading and writing. If a file with the same name already exists, it is overwritten.
a+b	Append access can be requested only in write mode. Use the following modes: FileMode::Append FileAccess::ReadWrite Use the **BinaryReader** and **BinaryWriter** classes for reading and writing.	Opens a binary file for reading and appending.

Migrating Using Interoperability Strategies

File I/O calls are provided by the standard I/O library stdio.h. This library is a part of ANSI standard C; hence it ports directly to Microsoft Windows. Similarly, many C and C++ file-related programs on UNIX can be recompiled on Windows with minimal changes. These programs can also be directly recompiled in a .NET Managed C++ project using the /CLR compiler switch (IJW mechanism).

You can also use the other .NET interoperability strategies, such as wrapping the unmanaged classes, or Platform Invocation services (DllImport) to migrate the file-related code to .NET. However, as explained in the “.NET Interoperability Mechanisms” section in Chapter 3, “.NET Interoperability” of this volume, the file-operations code will run unmanaged in the .NET environment.

Working with Directories

This section describes various routines to perform actions related to directories in UNIX and .NET such as accessing the current working directory, changing a directory, and deleting a directory.

Accessing the Current Working Directory

Directory operations involve calling the appropriate functions to traverse a directory hierarchy and to list the contents of a directory. The **Directory** class of the **System.IO** namespace in .NET contains all such appropriate directory access and manipulation functions.

UNIX provides the **_getcwd()**, **_get_current_dir_name()**, and **_getwd()** functions to get the current working directory.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
char *getwd(char *buf);
```

This code sample prints out the current working directory using the **get_current_dir_name()** function.

UNIX example: Print the current working directory

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    char *cwd;
    cwd = (char *)get_current_dir_name();
    printf("Current working directory: %s", cwd);
    exit(0);
}
```

(Source File: U_WorkingWithDir-UAMV4C5.01.c)

The **Directory** class in the **System.IO** namespace provides a static method called **GetCurrentDirectory()** to print the current directory. The following Managed C++ code sample prints out the current working directory using the **GetCurrentDirectory()** method.

.NET example: Print the current working directory

```
#using <mscorlib.dll>
using namespace System;
using namespace System::IO;
 int main()
 {
   try
     {
         String* cwd;
         cwd = Directory::GetCurrentDirectory();
         Console::WriteLine(S"Current working 
directory:{0}",cwd);
       }
   catch (Exception *e)
   {
      Console::WriteLine(S"The process failed:{0}",e);
   }
  }
```

(Source File: N_WorkingWithDir-UAMV4C5.01.cpp)

Most of the methods in the **Directory** class provide an object-oriented mechanism to access the underlying Microsoft Win32® application programming interface (API). For example, the **Directory::GetCurrentDirectory()** internally calls the Win32 API function **GetCurrentDirectory()**, which has been discussed in Volume 3: *Migrate Using Win32/Win64* of this guide. Similarly, the **Directory** class has a method called **GetLogicalDrives ()**, which returns all available drives in the system as an array of strings. Internally, this method calls **kernel32.dll GetLogicalDrives()**, which is a file I/O function in the kernel32.dll library.

Similarly, all **Directory** class methods internally invoke the Win32 native functions. In addition, all methods in the **Directory** class are static and need not be instantiated.

Accessing Directories

UNIX provides two system—**opendir()** and **readdir()**—to open and display the contents of a UNIX directory. The following code example in UNIX takes the name of the directory from the command line and displays its contents using the earlier system calls.

UNIX example: Accessing directories

```
#include <dirent.h>
#include <stdio.h>
#include <iostream>

void main(int argc, char *argv[])
{
  struct dirent *entryp;
  DIR *dp;
  char *directory = argv[1];

  if ((dp = opendir(directory)) == NULL)
  {
    perror("opendir failed");
    exit(1);
  }
  while ((entryp = readdir(dp)) != NULL)
    cout << entryp->d_name << endl;

  exit(0);
}
```

(Source File: U_WorkingWithDir-UAMV4C5.02.c)

The same operation can be achieved in .NET by using the **Directory** class and the **GetFiles()** method. The following Managed C++ code example shows how the earlier UNIX example is migrated to .NET.

.NET example: Accessing directories

```
#using <mscorlib.dll>

using namespace System;
using namespace System::IO;

void _tmain(int argc, char *argv[])
{
  try
  {
       String *fileNames[];
      String *directoryName = argv[1];
      fileNames = Directory::GetFiles(directoryName);
      for (int i = 0; i < fileNames->Length; i++)
        Console::WriteLine(fileNames[i]);
  }
  catch (Exception *e)
  {
    Console::WriteLine(e->get_Message());
  }
}
```

(Source File: N_WorkingWithDir-UAMV4C5.02.cpp)

Other Directory Operations

The following examples show how some common directory operations, such as creating a directory, changing the current directory, and deleting a directory, are performed using the **System::IO** namespace of .NET.

.NET example: Creating a new directory

The following Managed C++ creates a new directory called Test.

```
#using <mscorlib.dll>
using namespace System;
using namespace System::IO;

int main()
 {
   try
     {
      String* newDir = S"Test";
      Directory::CreateDirectory (newDir);
      Console::WriteLine (S"New Directory created");
     } catch(Exception *e){ Console::WriteLine(S"The process
failed:{0}",e); }
}
```

(Source File: N_WorkingWithDir-UAMV4C5.03.cpp)

UNIX example: Changing the current directory

The following code example in UNIX changes the current directory to a directory called Test if such a directory exists within the current directory. It uses the function **chgdir()** for this purpose.

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    char chgDir[] = "Test";
    int res;
    res = chdir(chgDir);
    if(res == 0)
    {
          printf("Change Directory successful");
    }
    else
    {
          printf("Error in Change Directory");
          exit(1);
    }
}
```

(Source File: U_WorkingWithDir-UAMV4C5.04.c)

.NET example: Changing the current directory

The following Managed C++ sample code changes the current directory to a directory called Test if such a directory exists within the current directory.

```
#using <mscorlib.dll>
using namespace System;
using namespace System::IO;
int main()
 {
   try
     {
      String* chgDir = "Test";
      if(Directory::Exists(chgDir))
      {
            Directory::SetCurrentDirectory(chgDir);
            Console::WriteLine(S"The Directory changed");
      }
      else
      {
            Console::WriteLine(S"The Directory does not exist");
      }
     } catch(Exception *e){Console::WriteLine(S"The process
failed:{0}",e); }
   }
```

(Source File: N_WorkingWithDir-UAMV4C5.04.cpp)

UNIX example: Deleting a directory

The following code example in UNIX deletes a directory named Test if such a directory exists within the current directory. It uses the **remove()** function for this purpose.

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    char delDir[] = "Test";
    int res;
    res = remove(delDir);
    if(res == 0)
    {
          printf("The directory successfully deleted");
    }
    else
    {
          printf("Unable to delete the directory");
    }
}
```

(Source File: U_WorkingWithDir-UAMV4C5.05.c)

.NET example: Deleting a directory

The following Managed C++ code example deletes a directory named Test if such a directory exists within the current directory.

```
#using <mscorlib.dll>
using namespace System;
using namespace System::IO;

int main() {
    // Specify the directories you want to manipulate.
    String* delDir = S"Test";
    try {
        // Determine whether the directory exists.
        if (Directory::Exists(delDir))
      {
            Directory::Delete(delDir);
            Console::WriteLine(S"The Directory Deleted");
        }
      else
      {
            Console::WriteLine(S"Directory does not exist");
      }


    } catch (Exception* e) {
        Console::WriteLine(S"The process failed: {0}", e);
    }
}
```

Chapter 6: Developing Phase: Infrastructure Services

This chapter discusses the programming differences between UNIX and Microsoft® .NET Framework. These differences are addressed in the following categories:

- Signals and events
- Exception handling in .NET
- Sockets and networking
- Interprocess communication
- Daemons versus services
- Database connectivity

In addition, this chapter outlines the various options available for converting the UNIX code to .NET in each of these categories and illustrates the options with appropriate source code examples. This information will assist you in choosing the appropriate approach for migrating your application to .NET. You can also use the examples in this chapter as a basis for constructing your .NET application.

Signals and Events

This section discusses the signal-specific implementations in UNIX and their suitable replacement mechanisms in .NET. It provides different alternatives for converting the signal-specific code on UNIX to event-specific code on .NET and suggests the pros and cons for each alternative.

The UNIX operating system supports a wide range of signals. UNIX signals are software interrupts that catch or indicate different types of events. Microsoft .NET Framework, on the other hand, supports a mechanism called events to provide the same functionality as UNIX signals.

Introduction to Events in .NET

An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic. The object that raises (triggers) the event is called the event sender. The object that captures the event and responds to it is called the event receiver.

Frequently, objects register themselves with another object to receive notifications for certain events. For example, consider an object that requires a notification when a Button object is clicked. The Button object offers a Click event that enables other objects to receive notification of the Click event. Objects that need to receive the notification create delegates (callback methods) and register themselves with the event. When the user clicks the button, the Button object fires the event, which sends the notification to all objects registered with the Button object.

An event handler is a method that is bound to an event. When the event is raised, the code within the event handler is executed.

In event communication, the event sender class does not know which object or method will receive (handle) the events it raises. What is needed is an intermediary (or a pointer mechanism) between the source and the receiver. The .NET Framework defines a special type (delegate) that meets this purpose and provides the functionality similar to function pointers in C++. Delegates are mainly used to handle the events in the .NET Framework.

Delegate

The delegate in .NET Framework is a class that can hold references to methods that match its signature. It is a type-safe function pointer or a callback method. Events and delegates are closely associated in .NET. An event is a message sent by an object based on the occurrence of some external interaction. Delegate is a form of object-oriented function pointer to invoke the function indirectly through its reference. Events are implemented using delegates. Event delegates are multicast, which means that the delegates can hold references to more than one event handling method, thus enabling flexibility and fine-grained control in event handling. It also acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

Delegates can be bound to a single method or to multiple methods, referred to as multicasting. Multicasting allows multiple events to be bound to the same method, thus allowing a many-to-one notification. The binding mechanism used with delegates is dynamic, which means that a delegate can be bound at run time to any method whose signature matches with that of the event handler. It can also be used on static methods and to call instance methods on objects. Delegates can also be chained together into a linked-list, so that calling through a delegate calls all the callback methods in the linked-list chain.

A more common use of delegates is to enable an object to implement a callback mechanism that other objects can "register." Implementing event notification registration typically entails having the callback method placed on a list of callback methods that is invoked when an event occurs. When events occur, the "event generating" object calls all the callback methods in its callback list. The event-generating object only looks for the signature of its callback method and does not care about any other details of the called object.

Event Model in .NET

This section describes the elements of the .NET event model and describes the various actions that can be performed:

- Raising events
- Raising multiple events
- Consuming events

The event model in .NET has the following four elements:

- A class that provides event data.

```
public class EventNameEventArgs:EventArgs
{
     //Declaration of Event Data
     //Properties for the Event Data
}
```

- An event delegate.

```
public delegate void EventNameEventHandler(object sender,
EventNameEventArgs e)
```

- A class that raises the event.

```
public class ControlName
{
     //An event declaration
     public event EventNameEventHandler EventName;

     //A method named onEventName that raises the event
protected virtual void OnAlarm(AlarmEvent e){...}
}
```

- A class that contains the event handler.

```
public class EventTest
{
    //Wires the handler method to the event
    Control.Event += new EventNameEventHandler(EventHandlerMethod);

    //Method that handles the event
    public void EventHandlerMethod(object sender, EventNameEventArgs
e)
    {…}
}
```

Raising Events

To raise an event, the following elements are required:

- A class that holds event data. This class must derive from **System.EventArgs**.
- A delegate for the event.
- A class that raises the event. This class must provide:
  - An event declaration.
  - A method that raises the event.

The event data class and the event delegate class might already have been defined in the .NET Framework class library or in a third-party class library. In that case, you do not have to define these classes.

Raising Multiple Events

If a class needs to raise multiple events, the .NET Framework provides a construct called *event properties* that can be used with another data structure (of your choice) to store event delegates.

Event properties consist of event declarations accompanied by *event accessors*. Event accessors are methods that allow event delegate instances to be added or removed from the storage data structure. Note that the event properties are slower than event fields because each event delegate has to be retrieved before it can be invoked. The trade-off is between memory and speed. If the class defines many events that are infrequently raised, consider implementing event properties. Windows Forms controls and ASP.NET server controls use event properties instead of event fields.

The .NET Framework provides a data structure for storing event delegates, the **System.ComponentModel.EventHandlerList** class, which is used by classes in the .NET Framework that raise multiple events. You can use this class or define your own data structure for storage.

Consuming Events

To consume an event in an application, you must provide an event handler (an event-handling method) that executes program logic in response to the event and registers the event handler with the event source. This process is referred to as event wiring.

Each event handler provides two parameters that allow you to handle the event properly.

```
//Method that handles the event
public void EventHandlerMethod(object sender, EventNameEventArgs e)
{…}
```

The first parameter, sender, provides a reference to the object that raised the event. The second parameter, e in the earlier example, passes an object specific to the event that is being handled. By referencing the properties—and, sometimes—the methods of the object, you can obtain information such as the location of the mouse for mouse events or the data being transferred in drag-and-drop events.

You can create an event handler at design time within the Windows Forms Designer. Double-click the design surface (either the form or a control) to create an event handler for the default action for that item.

You can create an event handler at run time. This allows you to connect event handlers based on conditions in code at run time instead of having the event handlers connected when the program initially starts.

In the application design, it may be necessary to have a single event handler used for multiple events or the multiple events fire the same procedure. For example, it is a powerful time-saver to have a menu command fire the same event as a button on the form does if they expose the same functionality.

SIGINT Implementation

The following example is a simple instance of catching SIGINT to detect CTRL-C.

UNIX example: Managing SIGINT signal

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
/* The intrpt function reacts to the signal passed in the parameter
signum. This function is called when a signal occurs. A message is
output, and then the signal handling for SIGINT is reset (by default
generated by pressing CTRL-C) back to the default behavior.
*/
void intrpt(int signum)
{
printf("I got signal %d\n", signum);
(void) signal(SIGINT, SIG_DFL);
}
/* main intercepts the SIGINT signal generated when Ctrl-C is input.
Otherwise, sits in an infinite loop, printing a message once a second.
*/
int main()
{
(void) signal(SIGINT, intrpt);
while(1) {
printf("Hello World!\n");
sleep(1);
}
}
```

(Source File: U_EventHandling-UAMV4C6.01.c)

.NET example: SIGINT implementation

SIGINT is implemented in .NET by using the **SetConsoleCtrlHandler,** the Windows API that adds or removes an application-defined **HandlerRoutine** function from the list of handler functions for the calling process.

```
using System;
using System.Threading;
using System.Runtime.InteropServices;
namespace KeyPressEvents
{
/// <summary>
/// Class to catch console control events (Ctrl+C, Ctrl+Break, etc) in
C#.
/// Calls SetConsoleCtrlHandler() in Win32 API
/// </summary>
public class ConsoleAppCtrl: IDisposable
{
/// <summary>
/// Declaration of an enumeration (An enumeration list that consists of
a set of named constants)
/// The events that can be captured by SetConsoleCtrlHandler()
/// </summary>
public enum ConsoleEvent
{
CTRL_C = 0,
CTRL_BREAK = 1,
CTRL_CLOSE = 2,
CTRL_LOGOFF = 5,
CTRL_SHUTDOWN = 6
}
/// <summary>
/// Event Handler to be called when a console event occurs.
/// </summary>
public delegate void ControlEventHandler(ConsoleEvent consoleEvent);
/// <summary>
/// Event fired when a console event occurs
/// </summary>
public event ControlEventHandler ControlEvent;
```

```
ControlEventHandler eventHandler;
/// <summary>
/// Create a new instance.
/// </summary>
public ConsoleAppCtrl()
{
eventHandler = new ControlEventHandler(Handler);
SetConsoleCtrlHandler(eventHandler, true);
}
~ConsoleAppCtrl()
{
Dispose(false);
}
public void Dispose()
{
Dispose(true);
}
/// <summary>
/// Disposing the handler
/// </summary>
/// <param name="disposing"></param>
void Dispose(bool disposing)
{
if (eventHandler != null)
{
SetConsoleCtrlHandler(eventHandler, false);
eventHandler = null;
}
}
/// <summary>
/// Event Handler method that is called when the event occurs
/// </summary>
/// <param name="consoleEvent"></param>
private void Handler(ConsoleEvent consoleEvent)
{
if (ControlEvent != null)
ControlEvent(consoleEvent);
}
/// <summary>
/// PInvoke usage of SetConsoleCtrlHandler present in kernel32.dll
/// </summary>
/// <param name="e"></param>
/// <param name="add"></param>
/// <returns></returns>
[DllImport("kernel32.dll")] // PInvoke usage of Windows API
//SetConsoleCtrlHandler
static extern bool SetConsoleCtrlHandler(ControlEventHandler e, bool
add);
}
class ConsoleClass
{
/// <summary>
/// The Event handler method to capture the Console events that occur
/// </summary>
/// <param name="consoleEvent"></param>
public static int iFlag = 0;
public static void MyHandler(ConsoleAppCtrl.ConsoleEvent consoleEvent)
{
switch(consoleEvent)
{
case ConsoleAppCtrl.ConsoleEvent.CTRL_C:
Console.WriteLine("Ctrl+C Captured");
iFlag = iFlag+1;
break;
case ConsoleAppCtrl.ConsoleEvent.CTRL_BREAK:
Console.WriteLine("Ctrl+Break Captured");
break;
default:
break;
}
}
public static void Main()
{
ConsoleAppCtrl cc = new ConsoleAppCtrl();
cc.ControlEvent += new ConsoleAppCtrl.ControlEventHandler(MyHandler);
while (true)
{
Console.WriteLine("Hello World!");
Thread.Sleep(1000);
if(iFlag == 2)
{
break;
}
}
}
}
```

```
}
```

(Source File: N_EventHandling-UAMV4C6.01.cs)

Replacing UNIX Signals Within .NET

This section explains how to replace the UNIX signal with the .NET event mechanism. UNIX uses signals to send alerts to processes when specific events occur. A UNIX application uses the **kill** function to activate signals internally. .NET does not support signals. Therefore, you have to rewrite the existing code to use another form of event notification in .NET.

The following example code illustrates how you can redevelop the UNIX code in .NET. It shows a simple main process that forks a child process, which issues the SIGALRM signal. The parent process catches the alarm and outputs a message when the signal is received.

UNIX example: Managing the SIGALRM signal

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
static int alarm_fired = 0;
/* The alrm_bell function simulates an alarm clock. */
void alrm_bell(int sig)
{
alarm_fired = 1;
}
int main()
{
int pid;
/* Child process waits for 5 sec's before sending SIGALRM to its
parent. */
printf("alarm application starting\n");
if((pid = fork()) == 0) {
sleep(5);
kill(getppid(), SIGALRM);
exit(0);
}
/* Parent process arranges to catch SIGALRM with a call to signal
and then waits for the child process to send SIGALRM. */
printf("waiting for alarm\n");
(void) signal(SIGALRM, alrm_bell);
pause();
if (alarm_fired)
printf("Ring...Ring!\n");
printf("alarm application done\n");
exit(0);
}
```

(Source File: U_EventHandling-UAMV4C6.02.c)

.NET example: Managing the SIGALRM signal

The **Timer** class in .NET can function as the SIGALARM signal in UNIX. The following example in Managed C++ illustrates how the UNIX example is implemented in .NET.

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;
using namespace System::Threading;
__gc class CppTimer
{
public:
CppTimer()
{

}
void execTimer(Object *stateInfo)
{
AutoResetEvent* objAutoEvt = dynamic_cast<AutoResetEvent*>(stateInfo);
Console::WriteLine(S"Ring...Ring!");
//Sets the state of the specified event to signaled.
objAutoEvt->Set();
}
};
void main()
{
CppTimer* objCppTimer = new CppTimer();
//The object of the AutoResetEvent class notifies the waiting thread
that an event has occured.
AutoResetEvent* objAutoReset = new AutoResetEvent(false);
//TimerCallback delegate represents the method that handles the calls
from a Timer object.
TimerCallback* objTimerCallBack = new TimerCallback(objCppTimer,
&CppTimer::execTimer);
Console::WriteLine(S"alarm application starting");
//The Timer class represents a mechanism for executing a method at
specific intervals
Timer* objTimer = new Timer(objTimerCallBack,objAutoReset,5000,0);
//WaitOne method blocks the current thread until the current WaitHandle
receives a signal.
objAutoReset->WaitOne(-1,false);
objTimer->Dispose();
Console::WriteLine(S"alarm application done");
}
```

(Source File: N_EventHandling-UAMV4C6.02.cpp)

Note More information on event handling and code samples for handling and raising events is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconevents.asp.

Exception Handling in .NET

This section describes the exception handling mechanism in .NET and its advantages and provides some examples of the exception handling process.

An exception is any error condition or unexpected behavior encountered by an executing program. Exceptions can be raised because of a fault in your code or in the code that you call (such as a shared library), unavailable operating system resources, and unexpected conditions that the common language runtime (CLR) encounters (such as code that cannot be verified). Applications can recover from some of these conditions, but not others. Generally, applications can recover from most application exceptions, but not from most run-time exceptions.

In .NET Framework, an exception is an object that inherits from the **Exception** class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates. All .NET Framework operations indicate failure by throwing exceptions.

Traditionally, error-handling models relied on either the specific language's unique way of detecting errors and locating handlers for them or on the error handling mechanism provided by the operating system. The runtime implements exception handling with the following features:

- Handles exceptions without regard for the language that generates the exception or the language that handles the exception.
- Does not require any particular language syntax for handling exceptions, but allows each language to define its own syntax.
- Allows exceptions to be thrown across processes and even computer boundaries.

Exceptions offer several advantages over other methods of error notification, such as return codes. These advantages include:

- Failures do not go unnoticed.
- Invalid values do not continue to propagate through the system.
- Programmers do not have to check return codes.
- Exception-handling code can be easily added to increase program reliability.
- The exception handling of the runtime is faster than Windows-based C++ error handling.

Exception Handling Model

The runtime uses an exception-handling model based on Exception objects and protected blocks of code. An Exception object is created to represent an exception when it occurs.

.NET supports structured exception handling (SEH), which helps you create and maintain programs with robust, comprehensive error handlers. The SEH code is designed to detect and respond to errors during execution by combining a control structure known as try-catch-finally. Using the try-catch-finally statements, you can protect blocks of code that have the potential to raise errors.

During execution, the CLR creates an exception information table for each executable. Each method of the executable has an associated array of exception handling information (which can be empty) in the exception information table. Each entry in the array describes a protected block of code, any exception filters associated with that code, and any exception handlers (catch statements). This exception table is extremely efficient in terms of performance, use of processor time, and memory use when an exception does not occur. The resources are used only when an exception occurs. The exception information table represents the following four types of exception handlers for protected blocks:

- A "finally" handler that executes whenever the block exits, whether that occurs by typical control flow or by an unhandled exception. This is useful when a cleanup or closure of resources is critical before continuing to the next section of code.

- A fault handler that must execute if an exception occurs, but does not execute on completion of typical control flow.
- A type-filtered handler that handles any exception of a specified class or any of its derived classes.
- A user-filtered handler that runs user-specified code to determine whether the exception should be handled by the associated handler or should be passed to the next protected block.

Each language implements these exception handlers according to its specifications. For example, Microsoft Visual Basic® .NET provides access to the user-filtered handler through a variable comparison (using the When keyword) in the catch statement; C# does not implement the user-filtered handler.

When an exception occurs, the runtime begins the following two-step process:

1. The runtime searches the array for the first protected block that:
   a. Protects a region that includes the currently executing instruction.
   b. Contains an exception handler or contains a filter that handles the exception.
2. If a match occurs, the runtime creates an Exception object that describes the exception. The runtime then executes all finally or fault statements between the statement where the exception has occurred and the statement handling the exception. The order of exception handlers is important; the innermost exception handler is evaluated first. The exception handlers can access the local variables and local memory of the routine that catches the exception; but intermediate values, generated at the time the exception is thrown, are lost.

   If no match occurs in the current method, the runtime searches each caller of the current method and continues this path all the way up the stack. If no caller has a match, the runtime allows the debugger to access the exception. If the debugger does not attach to the exception, the runtime raises the **UnhandledException** event. If there are no listeners for the **UnhandledException** event, the runtime dumps a stack trace and ends the program.

UNIX example: Exception handling

```
#include <iostream.h>
int main () {
  try
  {
    char * str;
    str = new char [100];
    if (str == NULL) throw "New was unable to allocate memory";
     int i = 0;
    for (;;)
    {
      if (i>9) throw i;
      str[i]='z';
      i++;
    }
  }
  catch (char * strEx)
  {
    cout << "Exception Caught: " << strEx << endl;
  }
  catch (int i)
  {
    cout << "Exception Caught: " <<"Index No." ;
    cout << i << " is out of range" << endl;
  }

  return 0;
}
```

(Source File: U_ExcepHandle-UAMV4C6.01.cpp)

.NET example: Exception handling

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;

int main()
{
      try
{
// managed char data type declaration
char __gc* str;
str = new char[100];
int i=0;
for(;;)
{
// throwing a user defined exception if i exceeds 9
if(i>9)
throw i;
str[i] = 'Z';
i++;
}
}
//Catching the user defined exception
catch(int i)
{
      Console::WriteLine(S"User defined exception thown :{0} ",
i.ToString());
}
//Catching the Sytem thrown exceptions - For example in the statement
//"str = new char[100]"
//if new was unable to allocate memory, then a NullReferenceException
//would have thrown.
catch(Exception* objExcp)
{
Console::WriteLine(S"Exception Caught : {0} ", objExcp->Message);
}
__finally
{
Console::WriteLine(S"Statement in finally block – always executed");
}
return 0;
}
```

(Source File: N_ExcepHandle-UAMV4C6.01.cpp)

Note More information on exception handling in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconExceptionHandlingFundamentals.asp.

Sockets and Networking

This section describes the sockets- and networking-related implementation mechanisms in .NET and details how to implement sockets- and networking-related code in the .NET application.

The .NET Framework class library includes two namespaces that help you with networking; these are the **System.Net** and **System.Net.Sockets** namespaces.

The **System.Net** classes:

- Provide a simple, yet complete solution for writing networked applications in managed code.

  Using a layered approach, the **System.Net** classes enable applications to access the network with varying levels of control, depending upon the needs of the application. The spectrum covered by these levels includes nearly every scenario on the Internet today, from a fine-grained control over sockets to a generic request/response model. In addition, the model is extensible; thus the model can continue to work with your application as the Internet evolves.
- Expose a robust implementation of the Hypertext Transfer Protocol (HTTP).

  With a large share of Web traffic today going over the HTTP, the importance of HTTP as an application protocol is significant. The **System.Net** classes support most of the HTTP 1.1 features. The advanced features include pipelining, chunking, authentication, preauthentication, encryption, proxy support, server certificate validation, connection management, and HTTP extensions.
- Are designed for writing scalable, high-performance middle-tier applications.

  The **System.Net** classes were designed specifically for a common Web server scenario; a single client browser request makes multiple requests to back-end or external servers. This scenario requires a robust middle-tier networking stack that can stand up to a high load. Such features as connection management, pipelining, keep-alive, and asynchronous send and receive ensure strong support for the middle tier. In addition, because **the System.Net** classes are part of an overall framework, integration with ASP+ features such as impersonation and caching is seamless.

WebRequest/WebResponse classes and **HTTP** classes are found in the **System.Net** namespace, while **TCP/UDP** and **Sockets** are found in the **System.Net.Sockets** namespace.

UNIX example: Server using sockets

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
void error(char *);
int main(int argc,char *argv[])
{
      int sockfd,newsockfd,portno,clilen;
      char buffer[256];
      struct sockaddr_in serv_addr,cli_addr;
      int n;
      printf("Server Running........\n");
      if(argc<2)
      {
            fprintf(stderr,"ERROR, no port provided\n");
            exit(1);
      }
      sockfd=socket(AF_INET,SOCK_STREAM,0);
      if(sockfd<0)
            error("ERROR Opening Socket");
      bzero((char *) &serv_addr,sizeof(serv_addr));
      portno=atoi(argv[1]);
      serv_addr.sin_family=AF_INET;
      serv_addr.sin_addr.s_addr=INADDR_ANY;
      serv_addr.sin_port=htons(portno);
      if(bind(sockfd,(struct sockaddr *)
&serv_addr,sizeof(serv_addr))<0)
            error("Error in Binding");
      listen(sockfd,5);
      clilen=sizeof(cli_addr);
      newsockfd=accept(sockfd,(struct sockaddr *) &cli_addr,&clilen);
      if(newsockfd<0)
            error("ERROR on Accept");
      bzero(buffer,256);
      n=read(newsockfd,buffer,255);
      if(n<0)
            error("ERROR Reading from Socket");
      printf(" %s\n",buffer);
      printf("the above message was received from the client\n");
      char szBuf1[256];
      bzero(szBuf1,256);
      strcpy(szBuf1,"from server to client: hi client\0");
      n=write(newsockfd,szBuf1,256);
      printf("n:%d",n);
      if(n<0)
            error("ERROR Writing to Socket");
      return 0;
}
void error(char *msg)
{
      perror(msg);
      exit(0);
}
```

(Source File: U_Sockets-UAMV4C6.01.c)

.NET example: Server using sockets

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;
using namespace System::Net;
using namespace System::Net::Sockets;
using namespace System::Text;
using namespace System::IO;
int main()
{
      String* args[] = Environment::GetCommandLineArgs();
      String* serverName = Dns::GetHostName();
      Char asciiChars[] = new Char[S"From the Server- Hi Client"-
>Length];
      asciiChars = S"From the Server- Hi Client"->ToCharArray();
      TcpListener *objTCP = new TcpListener(Convert::ToInt32(args[1]));
      objTCP->Start();
Console::WriteLine("Server named: {0} waiting on port: {1} ",
serverName,args[1]);
      Socket *objSock = objTCP->AcceptSocket();
      int byteNum = 0;
      String* strRec = "";
      if(objSock->Connected)
      {
                  Console::WriteLine(S"Client Connected");
                  NetworkStream *objNs = new
NetworkStream(objSock,true);
                  StreamReader *objReader = new StreamReader(objNs);
                  String* readLine = objReader->ReadLine();
Console::WriteLine("Message from the client {0}", readLine);
                  StreamWriter *objWriter = new StreamWriter(objNs);
                  objWriter->Write(asciiChars);
                  objWriter->Flush();
      }
      objSock->Close();
      return 0;
}
```

(Source File: N_Sockets-UAMV4C6.01.cpp)

For High-Performance Computing (HPC) applications, Microsoft provides Windows Compute Cluster Server 2003 as a member of the Windows Server family. It is a specialized 64-bit version server running Windows Server 2003 to support high-performance software. It enables workers to perform multinode workload computing and also supports execution of parallel applications based on the Message Passing Interface (MPI) standard. The latest versions of Visual Studio are designed for parallel computing and parallel debugging capability with MPI support.

Note More information on the Windows Compute Cluster Server 2003 is available at http://www.microsoft.com/windowsserver2003/hpc/.

More information on the **System.Net** namespace is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemnet.asp.

More information on the **System.Net.Sockets** namespace is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemNetSockets.asp.

Interprocess Communication

The operating system designed for multiprocessing or multitasking provides a mechanism for sharing the data between different applications, known as interprocess communication (IPC). This section describes various IPC mechanisms available in the UNIX and .NET environments and provides examples of these processes. The different forms of IPC that are described are:

- Process pipes
- Named pipes
- Shared memory
- Memory-mapped files
- Message queues

Sockets, discussed in the previous section, can also be used for IPC. .NET provides extensive support for implementation of sockets. Process pipes, named pipes, memory-mapped files, and shared memory can be implemented in .NET by invoking their respective Microsoft Win32® APIs through **P/Invoke**. .NET also supports the other forms of IPC, including message queuing (such as Microsoft Message Queuing and IBM MQSeries), .NET Remoting, and COM+.

MPI can also be used as an IPC mechanism for high-performance applications. MPI is a standard application programming interface (API) and specification for message passing, designed for high-performance computing scenarios. The Microsoft MPI in Windows Compute Cluster Server 2003 uses the Winsock Direct protocol for best performance and CPU efficiency.

Note More information on the Microsoft MPI is available at http://technet2.microsoft.com/WindowsServer/en/Library/4cb68e33-024b-4677-af36-28a1ebe9368f1033.mspx.

This section describes how you can convert UNIX code that uses the different forms of IPC. It also introduces new methods of IPC that are not available in UNIX but might provide a better solution that meets the IPC requirements of your application.

Process Pipes

Process pipes are not supported in .NET but can be implemented in .NET using the P/Invoke services. The following example in UNIX shows the implementation of process pipes in UNIX.

UNIX example: Implementation of process pipes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
```

```
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = (FILE *) popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output is:\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

(Source File: U_ProcessPipes-UAMV4C6.01.c)

For implementing this example in .NET, the code has to be first implemented in Win32 by changing the header files and the function names for **popen** and **pclose**. After writing the code successfully in Win32, it is compiled into a DLL (ProcessPipes.dll), and then it can be accessed from any .NET project, using the P/Invoke services. The following example shows how the earlier code is written using the Win32 API and then accessed from a C# project using the **DllImport** attribute.

Win32 example: Implementation of process pipes

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

processPipes()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    size_t chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = (FILE *)_popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output is:\n%s\n", buffer);
        }
        _pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

(Source File: W_ProcessPipes-UAMV4C6.01.cpp)

.NET example: Using Win32 implementation of process pipes

```
# using <mscorlib.dll>
using namespace System;
using namespace System.Runtime.InteropServices;

[DllImport("ProcessPipes.dll",CharSet=CharSet::Ansi)]
extern void processPipes();

int main()
{
      processPipes();
return 0;
}
```

(Source File: N_ProcessPipes-UAMV4C6.01.cpp)

Named Pipes

In this section, examples are given to illustrate how named pipes are implemented in UNIX and .NET. Named pipes are sometimes referred to as first-in-first-out (FIFO) and are generally half-duplex pipes as only one-way communication is possible.

Note More information on the implementation of named pipes for IPC in .NET is available at http://support.microsoft.com/?kbid=871044.

UNIX example: Implementation of named pipes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int fd;
    int i;

if (argc < 2) {
    fprintf(stderr, "Usage call once with \"r\" and then with
\"w\"\n");
    exit(EXIT_FAILURE);
}

// Check if the FIFO exists and create it if necessary.
    if (access(FIFO_NAME, F_OK) == -1) {
        fd = mkfifo(FIFO_NAME, 0777);
        if (fd != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }
    printf("Process %d opening FIFO\n", getpid());

// Open FIFO and output status result
    if (strncmp(argv[1], "r", 1) )
        fd = open(FIFO_NAME, O_RDONLY | O_NONBLOCK);
    else if (strncmp(argv[1], "w", 1) )
        fd = open(FIFO_NAME, O_WRONLY | O_NONBLOCK);
    else {
        fprintf(stderr, "Usage call once with \"r\" and then with
\"w\"\n");
        exit(EXIT_FAILURE);
    }

    printf("Process %d file descriptor: %d\n", getpid(), fd);
    sleep(3);
// Close FIFO
    if (fd != -1) close(fd);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

(Source File: U_NamedPipes-UAMV4C6.01.c)

The following example creates a named pipe from .NET by making a P/Invoke call to the Win32 API function **CreateNamedPipe()**.

.NET example: Implementation of named pipes

```
using namespace System;
using namespace System::Runtime::InteropServices;
using namespace System::Diagnostics;
using namespace System::Threading;
[DllImport("Kernel32.dll", CharSet=CharSet::Ansi)]
```

```
extern int CreateNamedPipe(char* lpName, int dwOpenMode, int
dwPipeMode, int nMaxInstances, int nOutBufferSize, int nInBufferSize,
int nDefaultTimeOut, IntPtr lpSecurityAttributes);
[DllImport("msvcrt.dll", CharSet=CharSet::Ansi)]
extern int _open(char * lpName, int rdOnly);
[DllImport("msvcrt.dll", CharSet=CharSet::Ansi)]
extern void _close(int fd);
[DllImport("msvcrt.dll", CharSet=CharSet::Ansi)]
extern int _getpid();

int main()
{
      const short FILE_ATTRIBUTE_NORMAL = 0x00000080;
      const int FILE_FLAG_NO_BUFFERING = 0x20000000;
      const int FILE_FLAG_WRITE_THROUGH = 0x80000000;
      const short PIPE_ACCESS_DUPLEX = 0x00000003;
      const short PIPE_READMODE_MESSAGE = 0x00000002;
      const short PIPE_TYPE_MESSAGE = 0x00000004;
      const short PIPE_WAIT = 0x00000000;
      const short INVALID_HANDLE_VALUE = -1;
      const int RDONLY = 0x0000;
      char* pipeName = "\\\\.\\pipe\\mynamedpipe";

      int pipeHandle = 0;
      int fd = -1;
      int processId = _getpid();
pipeHandle =
CreateNamedPipe(pipeName,PIPE_ACCESS_DUPLEX|FILE_FLAG_WRITE_THROUGH,PIP
E_WAIT|PIPE_TYPE_MESSAGE|PIPE_READMODE_MESSAGE,10,10000,2000,5000,
IntPtr::Zero);
      if(pipeHandle == INVALID_HANDLE_VALUE)
      {
Console::WriteLine(S"Could not create FIFO {0}",
Convert::ToString(pipeName));
            return 0;
      }
      Console::WriteLine(S"Process {0} opening FIFO",
processId.ToString() );
      fd = _open(pipeName,RDONLY);
Console::WriteLine(S"Process {0} file descriptor: {1}",
processId.ToString(),fd.ToString());
      Thread::Sleep(5000);
      if(fd!=-1)
      {
            (void)_close(fd);
            Console::WriteLine(S"Pipe {0} closed",
Convert::ToString(pipeName));

      }
      Console::WriteLine(S"Process {0} finished",
processId.ToString());
      return 0;
}
```

(Source File: N_NamedPipes-UAMV4C6.01.cpp)

Shared Memory and Memory-Mapped Files

Shared memory allows two or more threads or processes to share a region of memory. It is generally considered the most efficient method of IPC because data is not copied as part of the communication process. Instead, both the client and the server access the same physical area of memory. The System V IPC mechanisms for shared memory include the **shm*()** APIs, namely **shmat**, **shmctl**, **shmdt** and **shmget**. The signatures of these methods are given as follows:

```
Int shmget(key_t key, size_t size, int shmflg)
void *shmat(int shmid, const void *shmaddr, int shmflg)
int shmdt(const void *shmaddr)
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

UNIX example: Creating a shared memory area and mapping it

A simple example of creating a shared memory area and mapping it in UNIX is as follows:

```
if ( (fd = open("/dev/zero", O_RDWR)) < 0)
      err_sys("open error");
if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0)) == (caddr_t) -1)
      err_sys("mmap error");
close(fd);  // can close /dev/zero now that it's mapped
```

The .NET Framework class library does not provide any direct implementation for shared memory. Win32/Win64 does not support the **shm*()** APIs. However, MKS and Cygwin support the **shm*()** APIs for Win32/Win64. Win32/Win64 supports memory-mapped files and memory-mapped page files, which can be used to translate the UNIX code with the **shm*()** APIs. UNIX **shm***() API calls identify the shared memory section with an identification number and Windows memory mapping calls identify with a character string name. The Win32 API functions **CreateFileMapping** and **MapViewOfFile** are used for this. It involves making P/Invoke calls to these functions for the .NET implementation of the shared memory. How they can be imported in a C# or Managed C++ .NET project using the **DllImport** attribute is shown in the following code example.

CreateFileMapping

```
[DllImport("Kernel32.dll", SetLastError = true)]
extern IntPtr CreateFileMapping(IntPtr hFile,IntPtr
lpFileMappingAttributes,
IntPtr flProtect,IntPtr dwMaximumSizeHigh,IntPtr
dwMaximumSizeLow,IntPtr lpName);
```

MapViewOfFile

```
[DllImport("Kernel32.dll", SetLastError = true)]
extern IntPtr MapViewOfFile(IntPtr hFileMappingObject,IntPtr
dwDesiredAccess,
IntPtr dwFileOffsetHigh,IntPtr dwFileOffsetLow,IntPtr
dwNumberOfBytesToMap);
```

Note More information on the **CreateFileMapping** and the **MapViewOfFile** functions is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/createfilemapping.asp and http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/mapviewoffile.asp.

Message Queues

Microsoft Windows out-of-the box does not support UNIX System V style message queues. However, Windows supports message queues through the use of Microsoft Message Queuing (MSMQ). Additionally, System V-style support is available from third-party products like MKS or Cygwin. MSMQ is used in Windows to perform message queuing in your application. Message queuing is covered comprehensively in other Microsoft documentation and, therefore, is only briefly described here.

Note More information on Microsoft Message Queuing is available at http://www.microsoft.com/windows2000/technologies/communications/msmq/default.mspx.

Message queuing technology enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline. Applications send messages to queues and read messages from queues. Message Queuing provides guaranteed message delivery, efficient routing, security, and priority-based messaging. It can be used to implement solutions for both asynchronous and synchronous scenarios requiring high performance.

Note More information on implementation of MSMQ technology in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetasync1.asp.

Daemons vs. Services

This section provides an overview of the UNIX daemons and Windows services, and explains their similarities and differences.

A daemon in UNIX is a process that runs in the background to provide service to other applications and does not require a user interface. A service on Windows is the equivalent of an UNIX daemon. Normally, a daemon is started when the system is booted and runs without supervision until the system is shut down. Similarly, Windows services enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots to continue across the logon sessions, can be paused and restarted, and do not show any user interface. Services are ideal for use on a server or for long-running functionality that does not interfere with other users who are working on the same computer. It is possible to run services in the security context of a specific user account that is different from the logged-on user or the default computer account.

Note More information about services and Windows sessions is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/about_services.asp.

Using Microsoft Visual Studio .NET or the Microsoft .NET Framework SDK, you can easily create services by creating an application that is installed as a service. This type of application is called a Windows Service application. With .NET Framework features, you can create services, install services, start or stop services, and otherwise control their behavior.

The **System.ServiceProcess** namespace provides classes that allow you to implement, install, and control Windows Service applications. Implementing a service involves inheriting from the **ServiceBase** class and defining the specific behavior to be processed when start, stop, pause, and continue commands are passed as well as custom behavior and actions when the system shuts down.

Services are installed using an installation tool, such as **InstallUtil.exe**. The **System.ServiceProcess** namespace provides installation classes that write service information to the registry. The **ServiceProcessInstaller** class provides an encompassing class, which installs components common to all the services in an installation. For each service, create an instance of the **ServiceInstaller** class to install service-specific functionality.

The **ServiceController** class enables you to connect to an existing service and manipulate or get information about it. This class is typically used in an administrative capacity and enables you to start, stop, pause, continue, or perform custom commands on a service. The **ServiceBase** class defines the processing that a service performs when a command occurs; the **ServiceController** is the agent that enables you to call those commands on the service.

Note More information on creating, installing, and controlling Windows Service applications is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconintroductiontontserviceapplications.asp.

Database Connectivity

Database connectivity in UNIX applications is typically achieved using either ODBC (Open Database Connectivity) drivers or OCI (Oracle Call Interface, which is used for Oracle databases only).

ODBC provides access to an RDBMS (Relational Database Management System) using a standard data access interface. ODBC uses an RDBMS-specific driver to interact with the underlying data source.

OCI is a native library with a set of low-level APIs that allows fast access to the Oracle database.

.NET ships with ADO.NET, a data access technology that utilizes the features of CLR and the .NET Framework classes. The ADO.NET object model provides two basic components: the DataSet and DataProvider.

.NET data providers are available for all ODBC-compliant data sources (such as Oracle, Sybase, and DB2), Microsoft SQL Server™, and other OLE DB data sources.

To port the database access code in your application to .NET, you could rewrite the code to make use of ADO.NET. However, if your UNIX application uses direct OCI calls to connect to the Oracle database, consider making P/Invoke calls to OCI to reuse the existing code.

Note More information on access to the Oracle database in ADO.NET is available at http://www.fawcette.com/vsm/2003_01/magazine/features/beauchemin/.

Chapter 7: Developing Phase: Migrating the User Interface

This chapter describes how to migrate from a UNIX-based user interface (UI) to a Microsoft® Windows® UI using Windows Forms provided by Microsoft .NET. Windows Forms is a framework for building Windows rich-client applications that use the common language runtime (CLR). Because the overwhelming majority of UNIX graphical interfaces are built on X Windows and Motif, this chapter focuses on porting code from X Windows to .NET using Windows Forms.

This chapter covers the following topics:

- Architectural and visual differences between the UNIX and .NET environments.
- Programming principles used by X Windows and Windows Forms.
- Migrating different types of graphical constructs from one environment to the other.

This chapter also includes sections on migrating from other UNIX UI types, including text-based and OpenGL-based interfaces, and provides code examples illustrating how to migrate the UIs.

.NET Forms, Drawing, and GDI+

This section covers Windows Forms that are used for developing rich client applications, Web Forms that are used for developing Web applications, and Drawing and GDI+ namespaces and classes provided by the .NET Framework for drawing and graphics.

Windows Forms

Windows Forms is the new platform for Windows application development, based on the .NET Framework. This framework provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows applications.

The **System.Windows.Forms** namespace contains the group of classes that form the Windows Forms technology. These classes include the following:

- The **Control** class
- The **Form** class
- The **Component** class
- The **CommonDialog** class

These classes are discussed in detail later in this chapter, where a particular X Windows functionality is mapped to its corresponding functionality in .NET.

Web Forms

Web Forms can be used to create programmable Web pages that serve as the user interface of your Web application. It presents information to the user in any browser or client device and implements application logic using server-side code.

The **System.Web.UI** namespace provides classes and interfaces that allow you to create various controls and pages that will appear in your Web applications as user interface elements. The namespace provides the following important features by its classes:

- The **Control** class provides all the server controls with a common set of functionality.
- The **Page** class represents the Web Forms page that is requested from the Web server and contains the related methods and properties.
- The **data binding** classes provide the server controls with data binding functionality.
- Classes like **BaseParser** provide parsing functionality for the controls.

Drawing and GDI+

GDI stands for Graphics Device Interface. GDI+ is a class-based application-programming interface (API) for C and C++ programs. GDI+ enables programmers to build applications that use graphics and formatted text on the screen as well as the printer. When using GDI+, you do not need to access the graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by the 64-bit Windows operating system.

In the Microsoft .NET library, all the classes related to GDI+ are grouped under six namespaces and reside in the **System.Drawing.dll** assembly. The drawing namespace and the five subnamespaces are as follows:

- **System.Drawing**
- **System.Drawing.Design**
- **System.Drawing.Printing**
- **System.Drawing.Imaging**
- **System.Drawing.Drawing2D**
- **System.Drawing.Text**

These namespaces and the important classes associated with them are discussed in detail later in this chapter, where a particular X Windows functionality is mapped to its corresponding functionality in .NET.

Windows Forms Designer

Windows Forms, when used in conjunction with the Microsoft Visual Studio® .NET 2003 integrated development environment (IDE), enable Rapid Application Development (RAD) techniques for building applications. RAD enables you to just drag and drop controls onto your form, double-click that particular control, and write the code that corresponds to that particular event in the code editor. The Windows Forms Designer and the Visual Studio .NET 2003 IDE greatly facilitate the process of replicating the look of your graphical user interface (GUI) in the Windows environment.

Comparing X Windows and Windows Forms

The main UI type in use on the UNIX platform today builds on the X Windows set of standards, protocols, and libraries. To understand how to migrate such a UI, it is worth comparing the UI architecture and the resulting "look and feel" in the X Windows and Windows models. It is also useful to understand the similarities and differences in windowing terminology used in the two environments.

X Windows and Windows Forms are compared in the following concepts, which are explained in detail in the later subsections:

- User interface architecture
- Look and feel
- Window types

User Interface Architecture

The architecture of X Windows-based interfaces differs significantly from Windows architecture. The first and most fundamental difference is the orientation of client and server. For X Windows, the client is the application that requests services and receives information from the UI. The user-facing elements of the interface are based on what is termed the X Server.

In the X Windows-based system, the client application sends requests to the server to display graphics and to send mouse and keyboard events. The X Server is responsible for doing all the work on behalf of the client. The client might run on a remote system with no graphics hardware or on the same physical computer as the server. In either case, the client does not interact with the display, mouse, or keyboard. Figure 7.1 shows the X Windows client-server architecture.

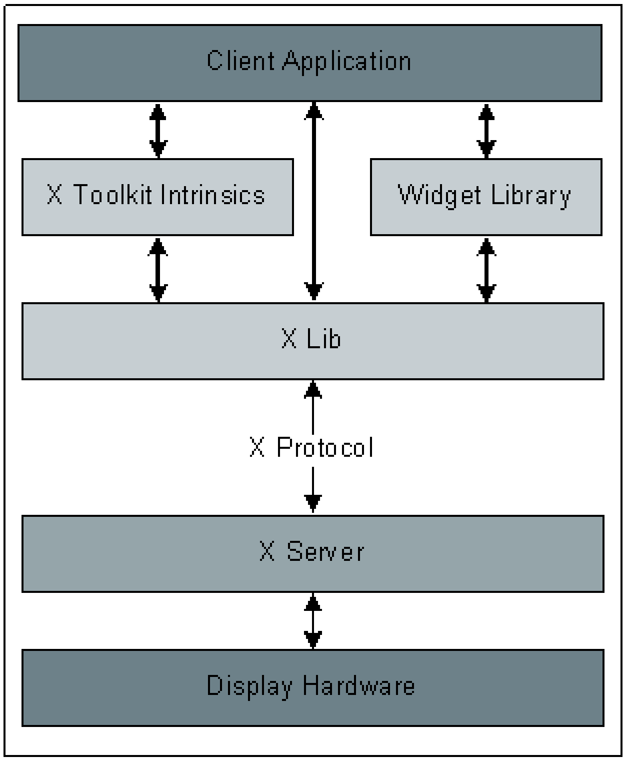


Figure 7.1. X Windows architectural model

A standard .NET-based, rich client application is not responsible for dealing with the display, mouse, or hardware. Figure 7.2 shows the Windows UI architecture using Windows Forms. It shows the path from an application, through the layers, to the hardware in the .NET environment.

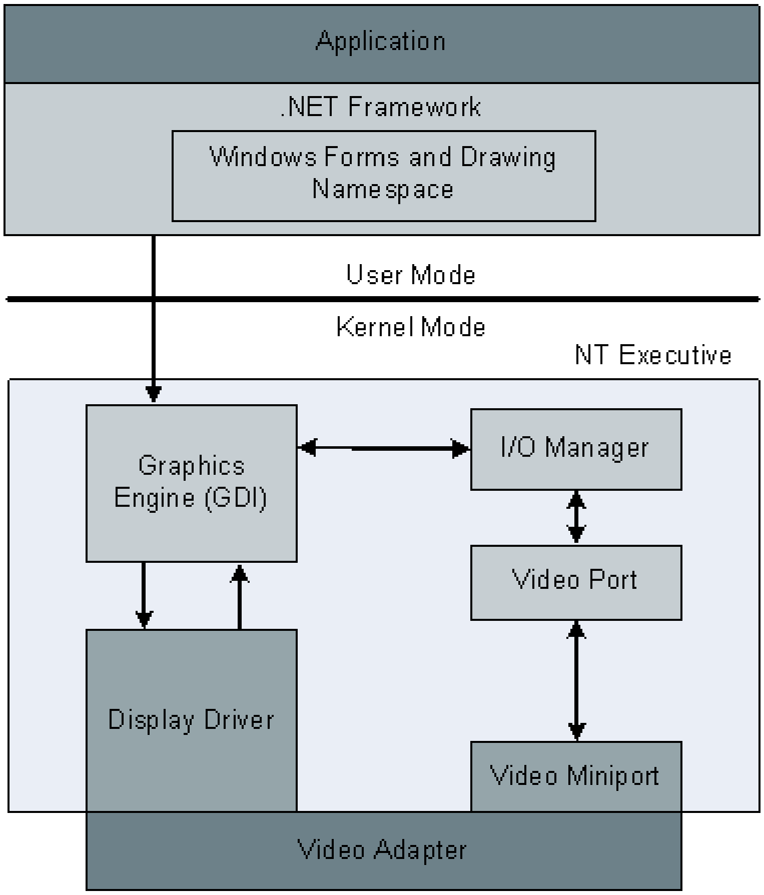


Figure 7.2. Windows UI architecture using Windows Forms

Look and Feel

X Windows is normally used with the Motif widget library, which is a library of UI components (such as scroll bars, buttons, drop-down lists, and dialog boxes) that can be used off the shelf. Motif is the most commonly used library to develop the UI component of X Windows applications.

According to the *Motif Programming Manual* and the *Microsoft Official Guidelines for User Interface Developers and Designers*, all applications that a user can run on the desktop should have a consistent "look and feel" as well as functional design. Otherwise, it is likely to confuse users, possibly to such an extent that they will not use the application.

Although there are many differences, UIs of both Microsoft Windows and X Windows with Motif have roots in the IBM Common User Access (CUA) guidelines. The resulting similarity in look and feel is not too surprising. Every windowing system needs to perform the same tasks, which are as follows:

- Determine the font to be used to display text.
- Determine the background color.
- Specify the location where a check box appears.
- Show that a user has clicked a particular button.

Given the task list, it just becomes a matter of the methodology used for accomplishing these tasks. Figure 7.3 and Figure 7.4 are examples of a Motif dialog box and a Windows dialog box respectively. Notice the similarities in terminology and appearance of dialog box elements in these two figures.

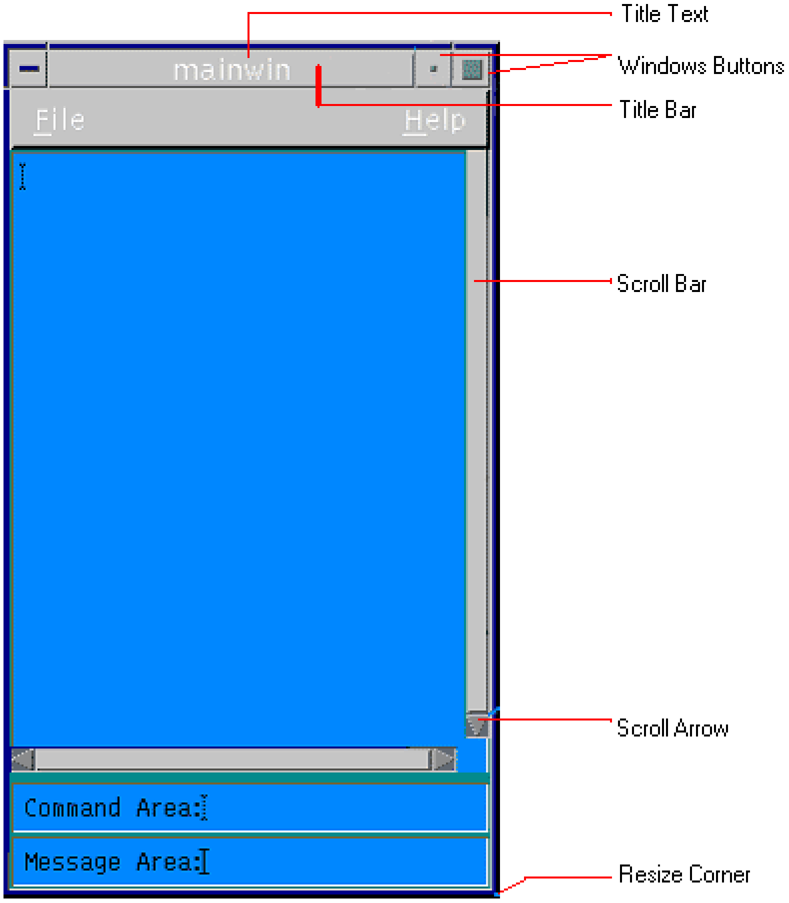


Figure 7.3. An example Motif dialog box

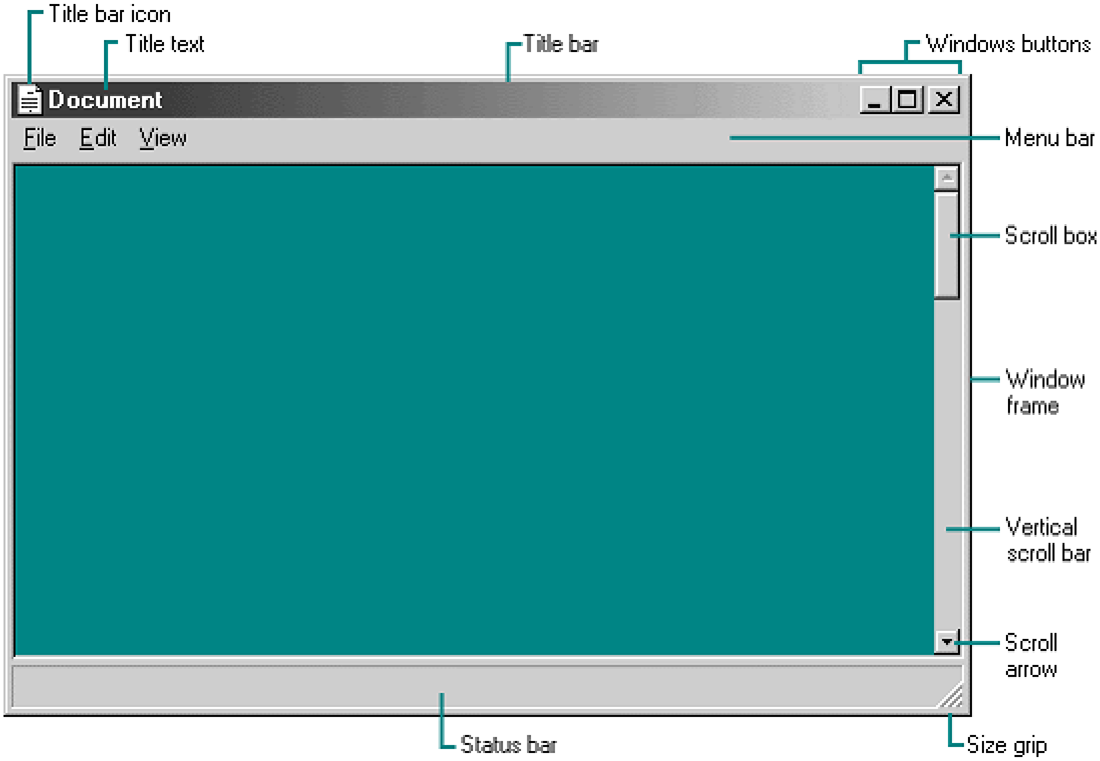


Figure 7.4. An example Windows dialog box

Note To ensure a consistent look and feel with other Windows-based applications, the development of a GUI on Windows using Windows Forms should be governed by the official Microsoft guidelines.

Window Types

Window types are very similar between the X Windows and Windows environments, as detailed in the following sections.

Desktop Window

The X Windows system automatically creates the desktop window. This is a system-defined window that is the base for all windows displayed by all applications. In X Windows, it can be thought of in the same general terms as the root window.

Application Window

The application window is the interface between the user and the application. Such elements as a menu bar, window menu, minimize and maximize buttons, Close button, title bar, sizing border, client area, and scroll bars typically appear in the application window.

Dialog Boxes

A dialog box is a temporary window, typically used by a user to create some additional input. A dialog box contains one or more controls, such as buttons and check boxes, to elicit user input.

Modeless Dialog Box

A modeless dialog box becomes the active window when the system creates it. The modeless dialog box neither disables its parent window nor sends messages to its parent window. However, it stays at the top of the z-order, even when the parent window becomes the active window. You use this mainly when you want to perform activities in the parent window and dialog box in any order. Applications can create a modeless dialog box by using the **System.Windows.Forms.Form** constructor and its various properties.

Modal Dialog Box

A modal dialog box becomes the active window when the system creates it and remains active until a call to **System.Windows.Forms.Form.Close** is made, which destroys the modal dialog box. Neither the application nor the user can make the parent window active. **Form.Close** must be called to close the modal dialog box. You use this mainly when you want to take some necessary action from the dialog box and want to make the decision before doing any other action.

An application uses the **System.Windows.Forms.Form** constructor, with the modal property set to true, to create a modal dialog box in Windows Forms.

Message Box

A message box is a special dialog box that displays a note, caution, or warning to the user. For example, a message box can inform the user of a problem that the application has encountered while performing a task.

An application uses **System.Windows.Forms.MessageBox.Show** to create a message box.

Reference Material

Table 7.1 lists the reference materials available for X Windows/Motif and Microsoft Windows. All of these Microsoft documents are available on the MSDN® Web site.

Table 7.1. References for X/Motif and Microsoft Windows

X Windows/Motif Reference	Microsoft Windows Reference
Motif Style Guide	*Official Guidelines for User Interface Developers and Designers*
Motif Programming Manual	<To be provided>
Motif Reference Manual	<To be provided>

Note The Microsoft .NET Framework SDK is available for download at http://www.microsoft.com/downloads/details.aspx?FamilyID=9b3a2ca6-3647-4070-9f41-a333c6b9181d&displaylang=en.

User Interface Programming in X Windows and Windows Forms

The core functionality offered by the X Windows environment is similar to the one that the Windows Forms environment offers. This section discusses the programming principles for developing UIs in both environments and provides information about libraries and include files. This section helps you in understanding Windows Forms programming and explains how to use these programming concepts to replace the X Windows UI.

Programming Principles

The basic structure of an X Windows–based application that uses Motif is very similar to the structure of a Microsoft Windows-based application.

To initiate an X Windows–based interface

3. Initialize the toolkit.
4. Create widgets.
5. Manage widgets.
6. Set up callbacks.
7. Display widgets.
8. Enter the main program event handler.

The following example code illustrates these steps.

```
topWidget = XtVaAppInitialize();
frame = XtVaCreateManagedwidget("frame",xmFrameWidgetFrams,
topWidget,,,);
button = XmCreatePushButton( frame, "EXIT", NULL, 0 );
XtManageChild(button)
XtAddCallback( button, XmNactivateCallback, myCallback, NULL );
XtRealizeWidget( topWidget );
XtAppMainLoop();
```

To initiate a Windows Forms-based application

1. Write your own form class that inherits from the **System.Windows.Forms.Form** class.
2. Instantiate the form class you have written.
3. Call the static method **Run()** of the **Application class** with the instance of your class as argument to launch your Windows application.

The following C# example code illustrates these steps.

```
using System;
using System.Windows.Forms;
namespace FirstWinFormAppl
{
      public class Form1 : System.Windows.Forms.Form
      {
            static void Main()
            {
                  Application.Run(new Form1());
            }


      }
}
```

After launching your application, you can instantiate new forms and display them using the **show()** method. The following example code illustrates this.

```
Form obForm = new Form();
obForm.Show();
```

Libraries and Include Files

Despite differences in their underlying architectures, many of the graphical functions used in X Windows and Windows Forms perform similar tasks. These include the core libraries and common dialog boxes.

Core Libraries

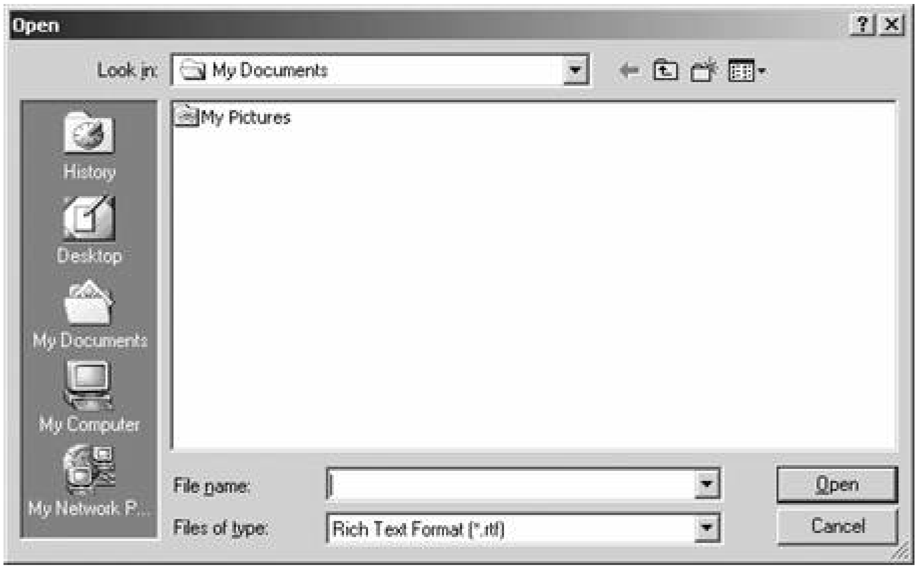
A number of functions exist to support the core API used in a GUI. X Windows includes the libraries X and Xlib and the X Windows Intrinsics toolkit. The .NET equivalent is the **System.Windows.Forms** namespace, which includes extensive UI-related classes. The namespace can be compared to the X Library and X Windows Intrinsics toolkit because they provide nearly all of the basic window management and two-dimensional graphics APIs.

Motif and Windows Forms Common Dialog Boxes

Dialog box functionality is provided by the Motif library in UNIX and by the Windows Forms dialog classes in .NET. If you migrate code from Motif, there is probably an equivalent Windows Forms common dialog box for each Motif function.

For example, the Motif function **XmCreateFileSelectionDialog()** is very similar to the **OpenFileDialog.ShowDialog()** function. The X Motif code must include the Xm/FileSB.h header file. The Windows Forms application must include the **System.Windows.Forms** namespace.

Calling the **OpenFileDialog.ShowDialog()** function displays the **Open File** dialog box, as shown in Figure 7.5.


Look in: My Documents

My Pictures

Files of type: Rich Text Format (*.rtf)

Figure 7.5. The OpenFileDialog.ShowDialog() common dialog box

Table 7.2 lists the common dialog boxes and related methods in .NET.

Table 7.2. Common Dialog Box and Related Methods in the .NET Framework API

Function	Description
System.Windows.Forms.ColorDialog constructor	Creates a **Color** dialog box that enables the user to select a color.
System.Windows.Forms.FontDialog constructor	Creates a **Font** dialog box that enables the user to choose attributes for a logical font.
System.IO.FileInfo.Name	Retrieves the name of the specified file.
System.Windows.Forms.OpenFileDialog.ShowDialog	Creates an **Open** dialog box that enables the user to specify the drive, the directory, and the name of a file or set of files to open.
System.Windows.Forms.SaveFileDialog.ShowDialog	Creates a **Save** dialog box that enables the user to specify the drive, the directory, and the name of a file to save.
System.Windows.Forms.PageSetupDialog.ShowDialog	Creates a **Page Setup** dialog box that enables the user to specify the attributes of a printed page.
System.Windows.Forms.PrintDialog.ShowDialog	Displays a **Print** dialog box.
PrintDlgEx()	Displays a **Print** property sheet that enables the user to specify the properties of a particular print job.

Note The **Find and Replace** dialog box, which enables the user to search for a text and replace it, is not readily available as part of the .NET Framework API.

The API for the **Find and Replace** dialog box is available for download at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/userinput/commondialogboxlibrary/aboutcommondialogboxes/findandreplacedialogboxes.asp.

Window Management

Window management functions cover the creation, initialization, management, and eventual removal of dialog boxes and other window types. This section discusses the following activities in window management:

- Creating windows.
- Creating controls.
- Identifying the control.
- Communicating with the control.

You can use the information in this section to create windows, create and identify controls, and communicate with controls using Windows Forms.

Creating Windows

The sample code provided in this section illustrates some of the X Windows and .NET implementations of window management. It is unlikely that any large-scale X Windows client or .NET Windows Forms–based application would actually be implemented as these code examples are. However, it will help you understand the conceptual similarities and some differences between the X Windows and .NET implementation of window management.

An X Windows X11 client might use **XtAppInitialize()**, **XtVaAppInitialize()**, **XtOpenApplication()**, or **XtVaOpenApplication** to get a top-level widget to create a window, as shown in the following example code.

X Windows example: Creating a window

```
main (int argc, char *argv[] )
{
  Widget toplevel;     /* Conceptual Application Window */
  XtAppContext  app;   /* context of the app */

  toplevel = XtVaAppInitialize( &app,
                                "myClassName",
                                NULL,0,&argc,argv,NULL,NULL );
```

The following is an alternative example code to get a top-level widget to create a window.

```
  toplevel = XtOpenApplication( &app,
                                "myClassName",
                                NULL,0,&argc,argv,NULL,
                                whateverWidgetClass, NULL,0);
  …
  …
```

In the following example code, a .NET Windows Forms application creates a main window.

.NET example: Creating a window

```
private void createWindow()
{
System.Windows.Forms.Form objForm = new System.Windows.Forms.Form();
objForm.Name = "myFormName";
objForm.Text = "myWindowName";
objForm.Height = 145;
objForm.Width = 500;
objForm.Show();
}
```

An X Windows client can create a control or widget, as shown in the following example code.

X Windows example: Creating a window with a control

```
main (int argc, char *argv[] )
{
  Widget toplevel;     /* Conceptual Application Window */
  Widget button;
  XtAppContext  app;   /* context of the app */
  toplevel = XtVaAppInitialize( &app, "Example",
NULL,0,&argc,argv,NULL,NULL );
  button   = XtVaCreateManagedWidget( "command",
                                       commandWidgetClass,  /* class
*/
                                       toplevel,           /* parent */
                                       XtNheight, 50,
                                       XtNwidth, 100,
                                       XtNlabel, "Press To Exit",
                                       NULL );
```

A .NET Windows Forms application can create a control, as shown in the following example code.

.NET example: Creating a window with a control

```
private void createWindow()
{
System.Windows.Forms.Form objForm  = new System.Windows.Forms.Form();
objForm.Name = "myFormName";
objForm.Text = "myWindowName";
objForm.Height = 145;
objForm.Width = 500;
objForm.Controls.Add(createButton());
objForm.Show();
}
private System.Windows.Forms.Button createButton()
{
System.Windows.Forms.Button objButton = new
System.Windows.Forms.Button();
objButton.Name = "myButton";
objButton.Text = "Press To Exit";
objButton.Top = 50;
objButton.Left = 150;
objButton.Height = 20;
objButton.Width = 100;
return objButton;
}
```

Creating Controls

Controls, such as X Windows widgets, come in all shapes, sizes, colors, and functions. There are two ways to create controls in a .NET environment. The first and simplest method is by using the form designer in Visual Studio .NET 2003 or by writing the code to generate a control.

Using Visual Studio .NET 2003, you can drag and drop controls onto a form, which in X Windows is a widget itself.

Using a form designer in Visual Studio .NET 2003 produces the following code in the code editor.

Note The following code is generated by the form designer in the code editor for a sample application that has a window, a text box, and a button.

.NET example: Creating controls

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Configuration;
namespace WindowAppTest
{
      public class WindowsAppTestForm : System.Windows.Forms.Form
      {
            private System.Windows.Forms.TextBox Display;
            private System.Windows.Forms.Button ClickMe;
            private System.ComponentModel.Container components = null;
            #region WindowsAppTestForm
            public WindowsAppTestForm()
            {

                  InitializeComponent();

            }
            protected override void Dispose( bool disposing )
            {
                  if( disposing )
                  {
                        if (components != null)
                        {
                              components.Dispose();
                        }
                  }
                  base.Dispose( disposing );
            }

            /// <summary>
            /// Required method for Designer support - do not modify
            /// the contents of this method with the code editor.
            /// </summary>
            private void InitializeComponent()
            {
                  this.Display = new System.Windows.Forms.TextBox();
                  this.ClickMe = new System.Windows.Forms.Button();
                  this.SuspendLayout();
                  //
                  // Display
                  //
                  this.Display.Location = new System.Drawing.Point(40,
32);
                  this.Display.Name = "Display";
                  this.Display.Size = new System.Drawing.Size(224, 20);
                  this.Display.TabIndex = 0;
                  this.Display.Text = "";
this.Display.TextChanged += new
System.EventHandler(this.Display_TextChanged);
                  //
                  // ClickMe
                  //
                  this.ClickMe.Location = new System.Drawing.Point(88,
80);
                  this.ClickMe.Name = "ClickMe";
                  this.ClickMe.Size = new System.Drawing.Size(96, 40);
                  this.ClickMe.TabIndex = 1;
                  this.ClickMe.Text = "Click Me";
this.ClickMe.Click += new System.EventHandler(this.button1_Click);
                  //
                  // WindowsAppTestForm
                  //
                  this.AutoScaleBaseSize = new System.Drawing.Size(5,
13);
                  this.ClientSize = new System.Drawing.Size(292, 141);
                  this.Controls.Add(this.ClickMe);
                  this.Controls.Add(this.Display);
                  this.Name = "WindowsAppTestForm";
                  this.Text = "WindowsAppTestForm";
                  this.ResumeLayout(false);

            }
            #endregion
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main()
            {
                  Application.Run(new WindowsAppTestForm());
            }
private void ClickMe_Click(object sender,System.EventArgs e)
            {
                  System.Windows.Forms.MessageBox.Show(Display.Text);

            }

private void Display_TextChanged(object sender, System.EventArgs e)
            {
                  String s = "Text Entered: ";
                  s += Display.Text;
System.Windows.Forms.MessageBox.Show(s,"Now entered");
            }
      }
}
```

The second method is to instantiate the objects of the corresponding classes, such as **System.Windows.Forms.Form** and **System.Windows.Forms.Button**. To produce the desired control at the desired location inside a parent window, the properties of the objects can be set.

A sample code to produce a control in X Windows is as follows:

```
thisButton =
    XtVaCreateManagedWidget("Fire Phasers", ← button text
                          commandWidgetClass, ← the type of widget
                          parentWidget, ← parent widget
                          NULL );
```

A sample code to produce a control in .NET Windows Forms is as follows:

```
private System.Windows.Forms.Button createButton()
{
System.Windows.Forms.Button objButton = new
System.Windows.Forms.Button();
objButton.Name = "myButton";
objButton.Text = "Press To Exit";
objButton.Top = 50;
objButton.Left = 150;
objButton.Height = 20;
objButton.Width = 100;
return objButton;
}
```

Identifying a Control

To communicate with or to respond to a control, it is necessary to identify the control. Controls are identified using the name property associated with the control object. For example:

```
System.Windows.Forms.Button objButton = new
System.Windows.Forms.Button();
objButton.Name = "myButton";
```

The name property of the object can be used to get or set the name of the object.

Communicating with a Control

After the application identifies a control, it can communicate with it. The following are some examples of sending and receiving commands or messages from controls in .NET and X Windows.

.NET example: Adding strings to a list box

```
//+
//  programmatically add strings to the list box
//-
// .NET
//
private System.Windows.Forms.ListBox mylistBox;
mylistBox.Items.Add("one");
mylistBox.Items.Add("two");
mylistBox.Items.Add("three");
```

X Windows example: Adding strings to a list box

```
// X11/Motif
//
XmString newString;
newString = XmStringCreateLocalized(“String One”);
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
newString = XmStringCreateLocalized(“String Two”);
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
newString = XmStringCreateLocalized(“String Three”);
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
```

.NET example: Setting the current focus

```
//+
//  programmatically force the current focus
//  to this control, i.e., make the user select
//  something now!
//-
// .NET
//
mylistBox.Focus();
```

X Windows example: Setting the current focus

```
// Xlib
//
XSetInputFocus( display, listWidget, RevertToParent, timeNow );
```

.NET example: Selecting a list box item

```
//+
//  programmatically pick a string for them!
//-
// .NET
//
mylistBox.SelectedIndex = 2;
```

X Windows example: Selecting a list box item

```
// X11/Motif
//
XmListSelectPos( listWidget, 2, 0 );
```

The “Signals and Events” section in Chapter 6, “Infrastructure Services” of this volume has a detailed discussion on communicating with controls and handlers for the events generated. The next section also describes keyboard and mouse event handling in .NET.

Event Handling

UI devices include keyboards and mouse devices, tablets, touchpads, and other devices. The most frequently used devices in many applications are the mouse and the keyboard. The following sections cover the mouse and keyboard events in Windows Forms.

Capturing Mouse Events

Like X Windows, mouse events can be associated with controls in Windows Forms. These events occur depending on the various mouse operations over the controls. Table 7.3 lists the various mouse events that are available in Windows Forms.

Table 7.3. Mouse Events in Windows Forms

Event	Argument	Description
Click	EventArgs	Occurs when a control is clicked.
DoubleClick	EventArgs	Occurs when a control is double-clicked.
MouseEnter	EventArgs	Occurs when a mouse cursor enters a control.
MouseLeave	EventArgs	Occurs when a mouse cursor leaves a control.
MouseHover	EventArgs	Occurs when a mouse cursor hovers over a control.
MouseDown	MouseEventArgs	Occurs when the mouse cursor is over a control and the mouse button is pressed.
MouseUp	MouseEventArgs	Occurs when the mouse cursor is over a control and the mouse button is released.
MouseMove	MouseEventArgs	Occurs when the mouse cursor is moved over a control.
MouseWheel	MouseEventArgs	Occurs when the mouse wheel is rotated.

The **MouseEventArgs** class provides two properties X and Y, which return the x-coordinate and the y-coordinate of the mouse pointer in pixels relative to the window's client area. There are two types of mouse events: the client-area mouse events and the nonclient-area mouse events. Client-area mouse events are those events that occur within the client area of the window or the form. Nonclient-area mouse events are those events that occur within the window but outside the client area. The nonclient area includes border, title bar, menu, scroll bars, and minimize and maximize buttons.

The following example code shows how the coordinates of the mouse are determined, whenever the MouseMove event occurs, using the **MouseEventArgs** class.

.NET example: Using the mouse coordinates

```
private void panel1_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)      {
            // Update the mouse path that is drawn onto the Panel.
            int mouseX = e.X;
            int mouseY = e.Y;

        }
```

Capturing Keyboard Events

In most applications, it may be necessary to capture keystrokes and perform some actions based on those keystrokes. Like X Motif, Windows Forms also provides a number of keyboard events. The **KeyDown** event occurs when a key is pressed and the **KeyUp** event occurs when the key is released. Table 7.4 lists the three keyboard events in Windows Forms.

Table 7.4. Key Events in Windows Forms

Event	Event Data	Description
KeyDown	KeyEventArgs	Occurs when a key is pressed.
KeyUp	KeyEventArgs	Occurs when a key is released.
KeyPress	KeyPressEventArgs	Occurs between the **KeyDown** and the **KeyUp** events and only when a character-generating key is pressed.

You can use the **KeyEventArgs** event data associated with the **KeyDown** and **KeyUp** events to obtain low-level information about the keystroke, such as determine the keycode (code that is used to uniquely identify a particular key) of the key that was pressed. It is also used to determine such things as whether any modifier keys (Alt, Ctrl, Shift keys, and the various combinations of these keys) were pressed or to determine whether a lowercase or an uppercase character was pressed.

You can use the **KeyPressEventArgs** event data associated with the **KeyPress** event to obtain a type that contains the ASCII character of the pressed key. The following example code shows how to catch the key events when text is entered in a text box control, named textBox1.

.NET example: Handling the KeyDown event

```
// Handle the KeyDown event to determine the type of character entered
into the control.
private void textBox1_KeyDown(object sender,
System.Windows.Forms.KeyEventArgs e)
{
      //Sample Handler code

      switch(e.KeyCode)
      {
      case Keys.D0:
      // Keystroke is number 0
      ..
      ..

      case Keys.Back:
      //Keystroke is backspace
      ..
      ..
      }
}

//Handle the KeyUp event
private void textBox1_KeyUp(object sender,
System.Windows.Forms.KeyEventArgs e)
```

```
{
      //Handler code

}
//Handle the KeyPress event
private void textBox1_KeyPress(object sender,
System.Windows.Forms.KeyPressEventArgs e)
{
      //handler code.

}
```

For most purposes, the standard **KeyUp**, **KeyDown**, and **KeyPress** events are sufficient to capture and handle keystrokes. However, not all controls raise these events for all keystrokes under all conditions. Follow the steps in the following paragraph to capture keystrokes, regardless of the state of the control.

To capture keystrokes in a Windows Forms control, you must derive a new class that is based on the class of the control that you want and override the **ProcessCmdKey** method. In this overridden method, place the code to process the keystrokes that you want to capture.

The following sample code is an example of the basic structure for such a class. The code demonstrates how to catch the keystrokes UP ARROW, DOWN ARROW, TAB, CTRL+M, and ALT+Z. The code given here is written to work with the DataGrid because this feature is most frequently requested for the DataGrid control. You can use the same approach with other .NET controls as well.

.NET example: Processing keystrokes

```
protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
{
   const int WM_KEYDOWN = 0x100;
   const int WM_SYSKEYDOWN = 0x104;
   if ((msg.Msg == WM_KEYDOWN) || (msg.Msg == WM_SYSKEYDOWN))
   {
      switch(keyData)
      {
         case Keys.Down:
            this.Parent.Text="Down Arrow Captured";
            break;

         case Keys.Up:
            this.Parent.Text="Up Arrow Captured";
            break;

         case Keys.Tab:
            this.Parent.Text="Tab Key Captured";
            break;

         case Keys.Control | Keys.M:
            this.Parent.Text="<CTRL> + M Captured";
            break;

         case Keys.Alt | Keys.Z:
            this.Parent.Text="<ALT> + Z Captured";
            break;
      }
   }

   return base.ProcessCmdKey(ref msg,keyData);
}
```

The system passes two parameters to the **ProcessCmdKey** method: msg and keyData. The msg parameter contains the Windows message, such as **WM_KEYDOWN**. The keyData parameter contains the key code of the key that was pressed. If CTRL or ALT was also pressed, the keyData parameter contains the ModifierKey information.

Using the msg parameter is not mandatory. However, it is a good practice to test the message. In this example, test **WM_KEYDOWN** to verify that this is a keystroke event. Also, test **WM_SYSKEYDOWN** to ensure that it is possible to catch keystroke combinations that include control keys (primarily ALT and CTRL).

To capture specific keys, you can evaluate the keyCode by comparing it to the keys enumeration. The code demonstrates how to catch the keystrokes UP ARROW, DOWN ARROW, TAB, CTRL+M, and ALT+Z.

Keyboard Focus

Keyboard focus is a temporary property of a control or a widget. The window or widget that is listening is said to have the current focus, keyboard focus, or focus.

Setting focus in Windows Forms–based applications involves calling the focus method on a particular control for which focus is requested. For example, if you want to set focus to a particular text box, then you need to call the focus method on that text box. The **Control.GotFocus()** is the event that is raised when a control gains focus and the **Control.LostFocus()** is the event that is raised when a control loses focus. This is similar to using **XmNfocusCallback** and **XmNlosingFocusCallback** for focus callbacks set up within X Motif.

The following example code shows how **GotFocus** and **LostFocus** events are handled for a text box named Textbox1.

.NET example: Textbox focus events

```
// This method handles the GotFocus event for TextBox1
private void TextBox1_GotFocus(object sender, System.EventArgs e)
{
      //Perform operations that needs to be done when TextBox1 gains
focus
}

// This method handles the LostFocus event for TextBox1
private void TextBox1_LostFocus(object sender, System.EventArgs e)
{
      //Perform operations that needs to be done when TextBox1 loses
focus.
}
```

You can also manually raise the **GotFocus** event by using the **Control.OnGotFocus()** method. Table 7.5 lists the events for getting current focus in the X Windows and .NET Framework environments.

Table 7.5. Getting Current Focus in X Windows and .NET Framework API

X Windows	.NET Framework API
No equivalent	**System.Windows.Forms.Form.ActiveForm**
XGetInputFocus()	**System.Windows.Forms.Form.ActiveForm**
XmGetFocusWidget()	**System.Windows.Forms.Control.Controls[].Focused**

Table 7.6 lists the events for setting current focus in the X Windows and .NET Framework environments.

Table 7.6. Setting Current Focus in X Windows and .NET Framework API

X Windows	.NET Framework API
No equivalent	**System.Windows.Forms.Form.Focus**
No equivalent	**System.Windows.Forms.Activate**
XSetInputFocus()	**System.Windows.Forms.Form.Activate**

Graphics Device Interface

The Graphics Device Interface (GDI) is a set of classes and functions used to generate graphics for devices such as displays and printers. These classes help you in creating graphics objects such as Pens, Brush, and Palette and in drawing shapes such as lines, circles, and rectangles. This section describes the GDI-specific routines and functions used in X Windows applications and their corresponding replacements in the .NET environment. Using this information, you can identify the best approach to migrate the GDI-specific routines in your UNIX application to the .NET environment.

Getting the Graphics Object

Applications on both platforms use a context to control the drawing functions. In X Windows systems, this context is known as the graphics context (GC) and in Microsoft Win32®-based GDI systems, this context is known as the device context (DC). Windows Forms provides a **Graphics** class that encapsulates the GDI+ drawing surface and maintains its state.

You can obtain the Windows Forms graphics object in several ways.

To obtain the Windows Forms graphics object

- Override the **OnPaint** event handler method. This method takes **PaintEventArgs** as a parameter. **PaintEventArgs** has a property that returns a graphics object. This is illustrated by the following example code:

  ```
  protected override void OnPaint (PaintEventArgs e)
  {
     Graphics g = e.Graphics;
  }
  ```

```
Or
```

- Call the **CreateGraphics** method on a control or a form, which would return the graphics object. The following example code illustrates this.

```
Graphics g = this.CreateGraphics();
```

Device Context

Applications on both UNIX and Windows use a context that controls the behavior of drawing functions. Windows Forms provides a **System.Drawing** namespace, which includes the **Graphics** class that encapsulates the GDI+ drawing surface and maintains its state.

However, there are some differences in UNIX and Windows regarding the use of context. The first difference is the location where the operating system stores and manages drawing attributes such as the width of lines or the current font. In X Windows, these values belong to the graphics context. When using **XCreateGC()** or **XtGetGC()**, it is necessary to provide a values mask and values structure. These values are used to store settings such as line width, foreground color, background color, and font style.

The following code is an example of the process of setting the foreground and background colors in X Windows.

X Windows example: Setting the foreground and background colors

```
GC             gcRedBlue;
XGCValues      gcValues;
unsigned long  gcColorRed;
unsigned long  gcColorBlue;
unsigned long  gcColorWhite;
Widget         myWidget;

int main (int args, char **argv)
{
  // initialize colors - widget - etc.

  gcValues.foreground = gcColorRed;
  gcValues.background = gcColorBlue;

  gcRedBlue = XtGetGC ( myWidget, GCForeground | GCBackground,
&gcValues);
}
```

The following code shows a .NET based method to set the foreground and background color of a form.

.NET example: Setting the foreground and background color of a form

```
private void setColor()
{
System.Windows.Forms.Form objForm = new Form();
objForm.ForeColor = System.Drawing.Color.White;
objForm.BackColor = System.Drawing.Color.Black;
objForm.Show();
}
```

.NET-based applications use a different approach. Apart from the **Graphics** class, the **System.Drawing** namespace also includes the classes for displaying text, a pen for line drawing, and a brush for painting and filling and drawing lines, rectangles, and text.

The following code shows a .NET based application that creates a pen and then uses it to draw lines.

.NET example: Creating a pen and drawing lines

```
private void DrawBlueLine()
{
System.Drawing.Pen objBlue = new Pen(System.Drawing.Color.Blue);
System.Drawing.Graphics objGraph = this.CreateGraphics();
objGraph.DrawLine(objBlue,0,0,100,200);
objBlue.Dispose();
objGraph.Dispose();
}
```

Windows Character Data Types

This section describes various routines and functions related to fonts and character sets used in X Windows applications and their alternatives in the .NET environment. The information provided in this section will enable you to identify the font- and text-specific routines in your UNIX applications and implement their replacements in the .NET environment. The subsections discuss the following topics in detail:

- Displaying text
- Text and drawing operations
- Calculating text metrics
- Text widgets and controls

Displaying Text

Ensuring that the visual display of text is as readable as possible requires creating and using the appropriate fonts and selecting the best mapping modes.

Using Fonts

Fonts control the display characteristics of text. An X Windows client application can use the **XLoadQueryFont()** and **XSetFont()** functions to apply a font to a given graphics context (GC), as shown in the following code.

X Windows example: Using the fonts

```
#define FONT1 "-*-lucida-medium-r-*-*-12-*-*-*-*-*-*-*"

Font           font1;
XFontStruct   *font1Info;

main() {

  Display  *pDisplay;
  int       iScreen;
  GC        gc;

  pDisplay  = XOpenDisplay(“myDisplay”);
  iScreen   = DefaultScreen(pDisplay);

  //+
  //  get the Graphics Context
  //-
  gc        = DefaultGC(pDisplay,iScreen);

  //+
  //  attempt to load the font
  //-
  font1Info = XLoadQueryFont( pDisplay,FONT1 );
  font1     = font1Info->fid;

  //+
  //  Set the font in the GC
  //-
  XSetFont( pDisplay, gc,  font1 );
   …
```

In .NET, the **System.Drawing** namespace provides a **Font** class, which defines the format and characteristics for text, including font face, size, and style attributes. In .NET, the graphics object is a high-level encapsulation of the graphics context. The text drawing function (**DrawString ()**) displays a text with a particular font on the form by taking that **Font** class object as one of its arguments.

All controls have a Font property, which defines the typeface, size, and style for any text displayed by that particular control. In addition, the Windows default font will be the default font for any control. To change the Windows default font, go to Control Panel, click **Display**, select **Properties,** and then click the **Settings** tab.

Note Windows provides several tools for adding and editing fonts.

The **Eudcedit.exe** tool, which comes with the operating system, allows the user to create unique characters such as logos and special characters. Eudcedit Help provides guidance on how to create, store, and use these characters in the font library.

The **Charmap.exe** tool, which comes with the operating system, allows the user to view, find, and copy characters from the Windows, MS-DOS®, and Unicode character sets. Charmap Help provides guidance on how to copy these characters.

The **Fontedit.exe** tool, which comes with Visual Studio .NET 2003, allows the user to create and edit raster fonts.

Creating Fonts

The **System.Drawing.Font** constructor is used for creating logical fonts, based on the fonts loaded in the system. The following example code duplicates the Times New Roman Windows font of size 14 pt and Bold style.

.NET Windows example: Creating the fonts

```
Public Font CreateTimesNewRomanFont()
{
      int size = 14;
      FontFamily fontFamily = new FontFamily("TimesNewRoman");
Font font = new Font(fontFamily,size, FontStyle.Regular,
GraphicsUnit.Point);
}
```

There are 13 overloaded types of the font constructor and you can use any one of these constructors to create logical fonts in your application. For further information on the **Font** class, refer to **System.Drawing.Font** in MSDN.

Device vs. Design Units

An application can retrieve font metrics for a particular family/style combination using the **FontFamily** class. The methods supported by this class are as follows:

- **GetEmHeight(FontStyle)**
- **GetCellAscent(FontStyle)**
- **GetCellDescent(FontStyle)**
- **GetLineSpacing(FontStyle)**

The numbers returned by these methods are in font design units; therefore, they are independent of the size and units of a particular Font object. The font metrics specific to the device are known as *device units*. Portable metrics in fonts are known as *design units*. To apply the design units to a specified device, convert design units to device units by using the following formula.

DeviceUnits = **(***DesignUnits***/***unitsPerEm***) * (***PointSize***/72) *** *DeviceResolution*

Note For a full explanation of device units, design units, and pixels, refer to the **System.Drawing** namespace at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDrawing.asp.

Fonts Example

The following example code overrides the OnPaint event of the form, creates a Font object (based on the Arial family) with size 16 pixels and displays the metrics (in pixels) for that particular Font object.

.NET example: Handling the OnPaint event

```
protected override void OnPaint (PaintEventArgs e)
{
      string infoString = "";  // enough space for one line of output
      int  ascent;             // font family ascent in design units
      float  ascentPixel;      // ascent converted to pixels
      int  descent;            // font family descent in design units
      float  descentPixel;     // descent converted to pixels
      int  lineSpacing;        // font family line spacing in design
      units
      float  lineSpacingPixel; // line spacing converted to pixels

      FontFamily fontFamily = new FontFamily("Arial");
      Font font = new Font(
         fontFamily,
         16, FontStyle.Regular,
         GraphicsUnit.Pixel);
      PointF pointF = new PointF(10, 10);
      SolidBrush   solidBrush = new SolidBrush(Color.Black);

      // Display the font size in pixels.
      infoString = "font.Size returns " + font.Size + ".";
      e.Graphics.DrawString(infoString, font, solidBrush, pointF);

      // Move down one line.
      pointF.Y += font.Height;

      // Display the font family em height in design units.
      infoString = "fontFamily.GetEmHeight() returns " +
         fontFamily.GetEmHeight(FontStyle.Regular) + ".";
      e.Graphics.DrawString(infoString, font, solidBrush, pointF);

      // Move down two lines.
      pointF.Y += 2 * font.Height;

      // Display the ascent in design units and pixels.
      ascent = fontFamily.GetCellAscent(FontStyle.Regular);

      // 14.484375 = 16.0 * 1854 / 2048
      ascentPixel =
         font.Size * ascent /
      fontFamily.GetEmHeight(FontStyle.Regular);
      infoString =  "The ascent is " + ascent + " design units, " +
      ascentPixel +
         " pixels.";
      e.Graphics.DrawString(infoString, font, solidBrush, pointF);

      // Move down one line.
      pointF.Y += font.Height;

      // Display the descent in design units and pixels.
      descent = fontFamily.GetCellDescent(FontStyle.Regular);

      // 3.390625 = 16.0 * 434 / 2048
      descentPixel =
         font.Size * descent /
      fontFamily.GetEmHeight(FontStyle.Regular);
      infoString = "The descent is " + descent + " design units, " +
         descentPixel + " pixels.";
      e.Graphics.DrawString(infoString, font, solidBrush, pointF);

      // Move down one line.
      pointF.Y += font.Height;

      // Display the line spacing in design units and pixels.
      lineSpacing = fontFamily.GetLineSpacing(FontStyle.Regular);

      // 18.398438 = 16.0 * 2355 / 2048
      lineSpacingPixel =
      font.Size * lineSpacing /
      fontFamily.GetEmHeight(FontStyle.Regular);
      infoString = "The line spacing is " + lineSpacing + " design
      units, " +
         lineSpacingPixel + " pixels.";
      e.Graphics.DrawString(infoString, font, solidBrush, pointF);
}
```

Text and Drawing Operations

This section describes how you can perform the following text and drawing operations in the .NET environment.

- Drawing text.
- Filling shapes.

c. Obtaining the color of the display elements.

- Drawing a gray text at the specified location.

This section also discusses the classes and related methods that help achieve these functionalities.

Drawing Text

There are two methods of drawing text: the X Windows **XDrawString()** function in X Windows and the **DrawString()** function of **Graphics** class in .NET. The **XDrawString()** requires a context to draw on, the x and y coordinates, the string, and the string length in characters. Similarly, the **DrawString()** method requires an object of the **Graphics** class to call it, the x and y coordinates, and the string to be drawn.

The code examples provided in this section draw the "Hello World" string in the current font and colors at the specified coordinates. It is often desirable to set a particular font or color before writing the text. The following code examples show how this task is accomplished in the two systems.

A programmer can code font and text display in X Windows as follows.

X Windows example: Drawing the text

```
Font = XLoadQueryFont (display,"fixed");
XSetFont (display, gc, font->fid);
XSetBackground (display, gc, WhitePixel (display,screen));
XSetForeground (display, gc, BlackPixel (display, screen));
XDrawString (display, d, gc, x, y, "Hello World", 11);
```

The **DrawString()** function in the **Graphics** class draws a formatted text in the specified location. The following code example creates a text, creates a font (Arial 16 pt), creates a solid black brush to draw with, and draws the text at the specified location in vertical format.

.NET example: Drawing the text

```
public void DrawStringFloatFormat(PaintEventArgs e)
{
      // Create string to draw.
      String drawString = "Sample Text";
      // Create font and brush.
      Font drawFont = new Font("Arial", 16);
      SolidBrush drawBrush = new SolidBrush(Color.Black);
      // Create point for upper-left corner of drawing.
      float x = 150.0F;
      float y =  50.0F;
      // Set format of string.
      StringFormat drawFormat = new StringFormat();
      drawFormat.FormatFlags = StringFormatFlags.DirectionVertical;
      // Draw string to screen.
      e.Graphics.DrawString(drawString, drawFont, drawBrush, x, y,
drawFormat);
}
```

One drawback of using **XDrawString()** is that the background is not erased before drawing the text string. Continually outputting strings to the same x and y coordinates results in a jumble of unreadable text strings, one upon the other. The X Windows library provides the **DrawImageString()** function, which calculates a rectangle containing the string and fills it with the background pixel color before drawing the text in the foreground pixel color.

However, in .NET, this is taken care of by an overloaded version of **DrawString()**. The syntax of that overloaded version is as follows:

```
public void DrawString(
   string s,
   Font font,
   Brush brush,
   RectangleF layoutRectangle
);
```

Filling Shapes

The constructor of **System.Drawing.SolidBrush** defines a brush of a single color. Brushes are used to fill graphics shapes, such as rectangles, ellipses, pies, polygons, or paths. The following example code shows the **SolidBrush** constructor. Color is a structure that represents the color of the brush.

```
public SolidBrush(
   Color color
);
```

Obtaining the Color of the Display Elements

The **System.Drawing.SystemColors** class contains several properties; each of these properties is a Color structure that represents the color of a Windows display element. Display elements are the parts of a window and the Windows display that appear on the system display screen. Some of the properties of the **SystemColors** class are explained in the following table.

Table 7.7. Properties of SystemColors Class

Property Name	Use of Property
SystemColors.ActiveBorder	Gets a Color structure that is the color of the active window's border.
SystemColors.Desktop	Gets a Color structure that is the color of the desktop.
SystemColors.Menu	Gets a Color structure that is the color of a menu's background.
SystemColors.ScrollBar	Gets a Color structure that is the color of the background of a scroll bar

The **SetTextColor()** function sets the text color for the specified device context to the specified color, as shown in the following example:

```
COLORREF SetTextColor(
  HDC      hdc,           // handle to DC
  COLORREF crColor        // text color
);
```

Drawing a Gray Text at the Specified Location

To draw a gray string at the specified location, you must pass **Drawing.Brushes.Gray** as the Brushes argument in the constructor of the **DrawString** function. The following example code draws a gray string at the specified location.

.NET example: Drawing the text with Brushes at specified location

```
public void DrawGrayString(PaintEventArgs e)
{
      // Create string to draw.
      String drawString = "Gray Text";
      // Create font and brush.
      Font drawFont = new Font("Arial", 16);
      // Create point for upper-left corner of drawing.
      float x = 150.0F;
      float y =  50.0F;
      // Draw string to screen.
      e.Graphics.DrawString(drawString, drawFont, Brushes.Gray, x, y);
}
```

Calculating Text Metrics

The X Windows programmer can rely on **XTextWidth()** to get the length of a character string in pixels. The Windows Forms programmer has an equivalent method called **MeasureString()** available in the **Graphics** class. This method returns a **SizeF** structure that represents the size, in pixels, of the specified string drawn with the specified font. From the **SizeF** structure, both the width and height of the string can be obtained. **SizeF.Width** represents the width of the string and **SizeF.Height** represents the height of the string.

The **MeasureString()** method is as follows.

```
public SizeF MeasureString(
   string text,
   Font font,
   int width
);
```

The following example code shows the use of **MeasureString**. It creates a string to measure and a font object set to Arial (16 pt). It then sets the maximum width of the string, creates a size object, and then uses the font object and the maximum string width to measure the size of the string. Finally, it draws a red rectangle using the measured size of the string and draws the string within the drawn rectangle.

.NET example: Measuring the string

```
 public void MeasureStringWidth(PaintEventArgs e)
 {
// Set up string.
string measureString = "Measure String";
Font stringFont = new Font("Arial", 16);
// Set maximum width of string.
int stringWidth = 200;
// Measure string.
SizeF stringSize = new SizeF();
stringSize = e.Graphics.MeasureString(measureString, stringFont,
stringWidth);
// Draw rectangle representing size of string.
e.Graphics.DrawRectangle(
new Pen(Color.Red, 1),
0.0F, 0.0F, stringSize.Width, stringSize.Height);
// Draw string to screen.
e.Graphics.DrawString(
measureString,
stringFont,
Brushes.Black,
new PointF(0, 0));
}
```

Text Widgets and Controls

A text widget or control is used to display, enter, and edit text. The exact functionality of a text widget or control depends upon how its resources are set.

In X Windows, the widget functionality is set as shown in the following example.

X Windows example: Setting widget functionality

```
text = XtVaCreateManagedWidget ( "myTextWidget",
                                 asciiTextWidgetClass,
                                 parentWidget,
                                 XtNfromHoriz,
                                 quit,
                                 XtNresize,
                                 XawtextResizeBoth,
                                 XtNresizable,
                                 True,
                                 NULL);
```

In Motif, the widget functionality is set as shown in the following example.

Motif example: Setting widget functionality

```
main (int argc, char *argv[])
{
  Widget       mainWidget;
  Widget       textWidget;
  XtAppContext appContext;

  mainWidget =

    XtVaOpenApplication ( &appContext,
                          "TextExample",
                          NULL,
                          0,
                          &argc,
                          argv,
                          NULL,
                          sessionShellWidgetClass,
                          NULL);
  (…)

  textWidget =

    XmCreateText ( mainWidget,"textWidget",NULL,0);

  (…)

  XtAppMainLoop( appContext );

}
```

In Windows Forms, a text box is used to display, enter, and edit text. A text box can be inserted into a form using the drag and drop feature of the IDE or it can be programmatically inserted into a form. The following example code shows how a text box is programmatically added to a form and displayed to the user.

.NET example: Using text box

```
public class FrmText : System.Windows.Forms.Form
{
      private TextBox txtDisplay;
      public Form1()
      {
            txtDisplay = new TextBox();
            txtDisplay.Location = new Point(50,50);
            txtDisplay.Text = "Default Text";
            Controls.Add(txtDisplay);

      }
      static void Main()
      {
            FrmText obFrm = new FrmText();
            Application.Run(obForm);
      }
}
```

Drawing

Drawing functions present graphical information on the screen. These range from primitive functions, such as turning a pixel on or off, to complex two-dimensional and three-dimensional drawing functions. The following are some notable differences in how drawing functions work on the X Windows and .NET platforms:

- Display and color management.
- Drawing two-dimensional lines.
- Drawing shapes and rectangles.

You can use the information in this section to assist you in understanding the programming differences in the drawing areas between the UNIX and .NET environments and in replacing the UNIX drawing routines with the equivalent .NET classes.

Display and Color Management

X Windows and Win32-based GDI are both constrained by the physical limitations of the available display hardware. One such limitation is the number of colors a display adapter is capable of showing.

All X Windows applications use a color map. This map can be shared or private. A shared color map is used by all applications that are not using a private map. Using a private map gives an application better control over color and a greater number of colors. However, there is one problem with private maps: When the mouse moves on or off the client area using a private map, the screen colors change.

.NET-based applications typically use color with no regard for the display device. If the application uses a color that is beyond the capabilities of the display device, the system approximates that color within the limits of the hardware. On display devices that support a color palette, applications sensitive to color quality can create and manage one or more logical palettes.

The **System.Drawing.Image.Palette** property is used to create a logical palette in .NET. The **Palette** property gets or sets the color palette (object of **System.Drawing.Imaging.ColorPalette**) used for the Image object. The **System.Drawing.Imaging.ColorPalette** defines an array of colors that make up a color palette. The colors are 32-bit ARGB colors.

To determine the capabilities of the hardware and calculate the best possible behaviors of the display, an X Windows program can use such functions as **DefaultColorMap()**, **DefaultVisual()**, **DisplayCells()**, **DisplayPlanes()**, **XGetVisualInfo(),** and **XGetWindowAttributes()**.

A .NET-based application can rely on **System.Drawing.Graphics** properties. The **Graphics** properties get or set device context information.

Note More information on the **ColorPalette** class and its members is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemdrawingimagingcolorpaletteclasstopic.asp.

Drawing Two-Dimensional Lines

Two sets of line and curve drawing functions are provided in the .NET environment. **System.Drawing.Graphics.DrawLine** and **System.Drawing.Graphics.DrawLines** are used for drawing lines. **System.Drawing.Graphics.DrawArc** and **System.Drawing.Graphics.DrawBeziers** are used for drawing curves.

The following table describes the preceding functions (one of overloaded functions) and their usage.

Table 7.8. Drawing lines in .NET

Function Name	Function Signature	Usage

System.Drawing.Graphics.DrawLine	`public void DrawLine(`
	`Pen, // Object of Pen class`
	`int, // x1 co-ordinate`
	`int, // y1 co-ordinate`
	`int, // x2 co-ordinate`
	`int  // y2 co-ordinate`
	`);`
		To draw a line from the current position up to, but not including, a specified point.

System.Drawing.Graphics.DrawLines	`public void DrawLines(`
	`Pen, // Objetc of Pen class`
	`Point[] //Array of Point Structure`
	`);`
		To draw a series of line segments by connecting the points in the specified array.

System.Drawing.Graphics. DrawArc	`public void DrawArc(`
	`   Pen pen,`
	`   Rectangle rect,`
	`   float startAngle,`
	`   float sweepAngle`
	`);`
		To draw an arc representing a portion of an ellipse specified by a Rectangle structure.

System.Drawing.Graphics.DrawBeziers	`public void DrawBeziers(`
	`   Pen pen,`
	`   Point[] points`
	`);`
		To draw a series of Bézier splines from an array of Point structures.

Note More information on drawing curves is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDrawingGraphicsMethodsTopic.asp.

The following example shows the use of **XDrawLine() in** X Windows.

X Windows example: Using XDrawLine() to draw a line

```
int main (int argc, char **argv)
{
  XtToolkitInitialize ();

  myApplication = XtCreateApplicationContext ();

  myDisplay     = XtOpenDisplay( myApplication,
                                 NULL,
                                 NULL,
                                 "XBlaat",
                                 NULL,
                                 0,
                                 &argc,
                                 argv);

  myWindow = RootWindowOfScreen(DefaultScreenOfDisplay (mydisplay));

  //+
  // now we need a surface to draw on
  //-
  myMap = XCreatePixmap ( myDisplay,myWindow,64,64, 1 );

  values.foreground =
    BlackPixel (myDisplay, DefaultScreen (myDisplay));

  myGC = XCreateGC (myDisplay, mySurface, GCForeground, &values);

  //+
  //  draw two diagonal lines across the 64x64 surface
  //
  XDrawLine( myDisplay,mySurface,myGC,0,0,63,63 );

  XDrawLine( myDisplay,mySurface,myGC,0,63,63,0 );

  …

}
```

The following example shows the use of **DrawLine** method in .NET to draw lines.

.NET example: Using DrawLine to draw lines

```
private void DrawX()
{
System.Drawing.Pen objPen = new Pen(System.Drawing.Color.Black);
System.Drawing.Graphics objGraph = this.CreateGraphics();
objGraph.DrawLine(objPen,0,0,63,63);
objGraph.DrawLine(objPen,0,63,63,0);
objPen.Dispose();
objGraph.Dispose();
}
```

Drawing Shapes and Rectangles

Filled shapes are geometric forms that the current pen can outline and the current brush can fill.

There are five filled shapes:

- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

The X Windows version of **XDrawRectangle()** uses an upper-left–corner point and the width and height.

In .NET, **System.Drawing.Graphics.DrawRectangle()** draws a rectangle specified by a coordinate pair: a width, and a height. **System.Drawing.Graphics.FillRectangle()** fills the interior of a rectangle specified by a pair of coordinates: a width, and a height.

Rectangle functions that fill the rectangle are:

- X Windows:**XFillRectangle()**
- .NET: **System.Drawing.Graphics.FillRectangle()**

Rectangle functions that draw the outline only are:

- X Windows: **XDrawRectangle()**
- .NET: **System.Drawing.Graphics.DrawRectangle()**

The following example demonstrates rectangle functions in X Windows.

X Windows example: Drawing a rectangle

```
void drawSomeRectangles()
{

  //+
  // fill the rectangle and then draw a black border around it
  //-
  XFillRectangle (myDisplay, mySurface, myWhiteGC, 0, 0, 31, 31);
  XDrawRectangle (myDisplay, mySurface, myBlackGC, 0, 0, 31, 31);



  //+
  //  draw an empty rectangle ten pixels square
  //-
  XDrawRectangle( myDislay, mySurface, myBlackGC, 0,0, 10,10 );
}
```

The following example demonstrates rectangle functions in .NET.

.NET example: Drawing a rectangle

```
private void DrawRec()
{
System.Drawing.Pen objPen = new Pen(System.Drawing.Color.Black);
System.Drawing.SolidBrush objBrush = new
SolidBrush(System.Drawing.Color.White);
System.Drawing.Graphics objGraph = this.CreateGraphics();
objGraph.FillRectangle(objBrush,0,0,31,31);
objGraph.DrawRectangle(objPen,0,0,31,31);
objGraph.DrawRectangle(objPen,0,0,10,10);
objPen.Dispose();
objBrush.Dispose();
objGraph.Dispose();
}
```

Timers

Timers are required to determine and act on delays in user input. The functionality and differences between X Windows timeouts and .NET timers are discussed in the following sections.

X Windows Timer

An X Windows client program can use the **XtAddAppTimeOut()** and **XtRemoveTimeOut()** functions with a callback to perform processing based on an interval specified in milliseconds. The following code shows an example of this.

X Windows example: Handling timer events

```
//+
// perform task every one second
//-
void myTimerProc( Widget    w,
                  XEvent   *event,
                  String   *pars,
                  Cardinal *npars)
{
      …
      …
}

//+
// start a 1 second timer
//-
void startTimer( XtIntervalId *timer )
{
  (*timer) = XtAppAddTimeOut( gContext, 1000, myTimerProc, NULL);
}

//+
// stop the timer
//-
void stopTimer( XtIntervalId * timer )
{
  if( *timer ) {
    XtRemoveTimeOut( *timer );
    (*timer) = NULL;
  }
}
```

.NET Timers

You can use .NET timers in two scenarios. First, as in the X Windows approach, a callback function can be identified to execute at each timer interval. Second, an event can be raised to process timer intervals.

The **System.Threading.Timer** class provides a mechanism for executing a method at specified intervals.

The following example shows the callback version using **System.Threading.Timer.**

.NET example: Handling timer events with System.Threading

```
using System;
using System.Threading;

namespace AlarmSignal
{
      /// <summary>
      /// Class to simulate the alarm
      /// </summary>
      class TimerSample
      {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            static void Main()
            {
                  CallBackClass objCallBack = new CallBackClass();

//The object of the AutoResetEvent class notifies the waiting thread
that an event has occured.
                  AutoResetEvent objState =  new AutoResetEvent(false);

//TimerCallback delegate represents the method that handles the calls
from a Timer object.
TimerCallback objTimerCallBack = new
TimerCallback(objCallBack.execTimer);
                  Console.WriteLine("alarm application starting");

//The Timer class represents a mechanism for executing a method at
specific intervals
Timer objTimer = new Timer(objTimerCallBack,objState,5000,0);
                  Console.WriteLine("waiting for alarm");

//WaitOne method blocks the current thread until the current WaitHandle
receives a signal.
                  objState.WaitOne(-1,false);
                              objTimer.Dispose();
                  Console.WriteLine("alarm application done");
            }
      }
      /// <summary>
      /// Class that has the method for executing the alarm.
      /// </summary>
      class CallBackClass
      {
            public void execTimer(Object stateInfo)
            {
                  AutoResetEvent objAutoEvt = (AutoResetEvent)
stateInfo;
                  Console.WriteLine("Ring...Ring!");
                  //Sets the state of the specified event to signaled.
                  objAutoEvt.Set();
            }
      }
}
```

(Source File: N_Timers-UAMV4C7.01.cs)

The **System.Timers** namespace provides the **Timer** component, which allows you to raise an event on a specified interval.

The following example shows the raising of an event to process time intervals.

.NET example: Handling timer events with System.Timers

```
using System;
using System.Timers;

public class Timer1
{
      public static void Main()
      {
            System.Timers.Timer aTimer = new System.Timers.Timer();
            aTimer.Elapsed+=new ElapsedEventHandler(OnTimedEvent);
            // Set the Interval to 5 seconds.
            aTimer.Interval=5000;
            aTimer.Enabled=true;

            Console.WriteLine("Press \'q\' to quit the sample.");
            while(Console.Read()!='q');
      }
      // Specify what you want to happen when the Elapsed event is
raised.
      private static void OnTimedEvent(object source, ElapsedEventArgs
e)
      {
            Console.WriteLine("Hello World!");
      }
}
```

(Source File: N_Timers-UAMV4C7.02.cs)

Migrating Character-based User Interfaces

Not all UNIX-based interfaces are graphical. Character-based interfaces were the original mainstay of UNIX computing, long before the graphical workstation was developed. The character-based UIs can be replaced with a Windows Forms–based or HTML-based UIs using .NET.

Porting OpenGL Applications

OpenGL was originally developed by Silicon Graphics as a platform-independent set of graphics APIs. This has made OpenGL an attractive option for developers who want to target multiple platforms. The interface of the OpenGL library is standardized by specifications that are available publicly. New functionalities called OpenGL Extensions can also be added without updating the entire specification.

However, OpenGL is not a set of windowing libraries. An OpenGL application with windows uses either the windowing system of the target platform (X Windows or Win32) or a cross-platform library such as the OpenGL Graphics Library Utility Kit (GLUT). Because of licensing concerns, however, most commercial applications incorporate the target platform windowing system.

When moving a UNIX application that uses OpenGL to .NET, the .NET interoperability strategies are used. While migrating the OpenGL libraries using .NET interoperability mechanism, the migrated libraries are treated as unmanaged code. Some third-party open source .NET classes are available, which wrap access to OpenGL and allow .NET to act as a container to access OpenGL. Internally, these classes have used the interoperability mechanisms such as P/Invoke only. An example of this is the C# OpenGL library.

Note More information on this library is available at
http://csgl.sourceforge.net/.

More information on OpenGL and platform-specific examples is available at
http://www.sgi.com/software/opengl/ and
http://www.opengl.org/.

Mapping X Windows Terminology to Windows Forms

The graphical models of UNIX and Windows are very different. There are conceptual similarities, but little side-by-side mapping is possible. The following sections describe the possible mappings.

In the headings of the following sections, the Win32 GDI term is followed by the corresponding X Windows term in the following format: *X Windows term vs. .NET term*.

Callback vs. Event Handlers

Windows Forms uses the **Event Handler** functions in the same capacity as **Callback** in X Windows.

Client vs. Client Window

X Windows consists of a protocol that describes how a client interacts with a server that could be running on a remote computer. How objects are drawn is the responsibility of the server. This provides device-independence for the client application because it is not responsible for knowing anything about the physical hardware.

In the Windows environment, the Windows Forms provide a layer of device-independence and are not required to access the graphics hardware directly.

A single Windows-based application can contain any number of separate windows. Each of these windows can have a window frame, caption bar, system menu, minimize and maximize buttons, and its own main display area, which is referred to as the client window.

Multiple document interface (MDI) applications have three kinds of windows: a frame window, an MDI client window, and a number of child windows. The term "iclient window" takes on a special meaning in this case. iClient is special kind of an application with visualization, input ctrl, and local models.

Note More information about MDI and the MDI documentation is available at http://msdn2.microsoft.com/en-us/library/ms171654.aspx.

Console Mode vs. Command Window

If X Windows or some other GUI is not running on a UNIX system, a user must work in text only or in console mode.

Windows is exactly the opposite. If a console is not running, the user must work in GUI mode. Windows text-based mode is provided by running the **Cmd.exe** tool. This environment is also referred to as a command window or the MS-DOS prompt. To run the **Cmd.exe** tool, click **Start**, and then click **Run**. Type **cmd**, and then click **OK**. Alternatively, on the keyboard, press the Windows key and then press R. The **Run** dialog box appears.

DPI vs. Screen Resolution

When starting an X Windows session, using the **-dpi** (dots per inch) option can improve appearance on displays with larger resolutions, such as 1600 × 1200. The **-dpi** option also helps to work around possible font issues.

A Windows-based application is usually built with no assumptions about the capabilities of the system that the application will run on. The .NET Framework API provides various classes to access system and device-specific information. The properties of **System.Windows.Forms.SystemInformation** are used to retrieve system metrics and configuration settings. **System.Drawing.Graphics** properties and **System.Drawing.Printing.PrinterSettings** and **System.Management** classes are used to retrieve device-specific information for the specified device.

Graphics Context vs. Graphics Class Object

The X Windows graphics context (GC) contains required information about how drawing functions are to be executed. In .NET, the Graphics class object provides similar information. The **Graphics** class here is an encapsulation of the GDI+ drawing surface, which includes the Win32 device context (DC). Table 7.9 lists X Windows graphic context and the corresponding Win32 device context.

Table 7.9. X Windows GC and Win32 DC Comparable Functions

Xlib	.NET Framework API
XtGetGC	To retrieve a Graphics class object, use any of the following: **System.Drawing.Graphics.FromHwnd(Windows.Forms.Control.Handle)** **System.Drawing.Printing.PrintPageEventArgs.Graphics** **System.Windows.Forms.PaintEventArgs.Graphics** To explicitly retrieve a handle, use the following: **System.Drawing.Graphics.GetHdc**
XtReleaseGC	**System.Drawing.Graphics.ReleaseHdc**

Resources vs. Properties

In X Windows terminology, a widget is defined by its *resources*. Width, height, color, and font are examples of resources. Resources can be managed by using the **XtVaCreateManagedWidget()** method, by using resource files, or by using **XtVaGetValues()** and **XtVaSetValues()**.

In Windows Forms terminology, a control is defined by its properties. For example, a text control has the following properties: Center Vertically, No Wrap, Transparent, Right-Aligned Text, and Visible.

Resource Files vs. Configuration Files

X Windows systems use configuration files, referred to as resource files, to store information about system settings or preferences for a particular X Windows client.

In .NET, these settings and preferences for applications are generally stored in application configuration files, which are in XML format. The following is a sample configuration file, which an application reads.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="LabelColor" value="Yellow" />
    <add key="MaximumWeight" value="150" />
    <add key="Title" value="MyApplication" />
  </appSettings>
</configuration>
```

The configuration file should be in the same directory as the application. The name of the configuration file should be the same as the application, with .config at the end. For example, an application called MyApplication.exe should have a configuration file called MyApplication.exe.config.

Root Window vs. Desktop Window

All X Windows windows are descendents of the root window. In the Windows environment, the desktop window is a system-defined window that is the base for all windows displayed by all applications.

/bin vs. /System32

In Windows, the /System32 directory is roughly equivalent to the /bin directory on a UNIX system. This is where the system executable files are located. The /System32 directory is located in the system root directory. To find the system root, type **set** at the command prompt and press ENTER. This displays a listing of the current environment. In the list, locate SYSTEMROOT. Under SYSTEMROOT, there is an entry similar to SYSTEMROOT=C:\WINNT. This is the system directory and under that directory is the /System32 directory.

/usr/bin vs. Program Files

The Program Files directory on a Windows-based system is similar to the /usr/bin directory on a UNIX system. This is the default location for user applications. In Windows, a user can create more than one such directory. Each drive, for example, might have a Program Files directory. The system environment variable, ProgramFiles, contains the path of one default location. For example, ProgramFiles=C:\ProgramFiles.

/usr/lib vs. LIB Environment Variable

In Windows, the path to user libraries can be anywhere. To manage this relationship, retrieve or set the system environment variable LIB.

/usr/include vs. INCLUDE Environment Variable

In Windows, the path to user include files can be anywhere. To manage this relationship, retrieve or set the system environment variable INCLUDE.

Pixmap or Bitmap vs. Bitmap

In X Windows, bitmap and pixmap have the same use as Windows bitmaps. For example, they can be used as pictures, fill patterns, icons, and cursors. They are, however, very different in form.

The following example represents a simple 16 × 16 "X" figure in X Windows.

```
#define simple_width 16
#define simple_height 16
static unsigned char simple_bits[] = {
      0x01, 0x80, 0x02, 0x04, 0x20, 0x08, 0x10, 0x10, 0x08, 0x20, 0x04,
      0x40, 0x02, 0x80, 0x01, 0x80, 0x01, 0x02, 0x20, 0x04, 0x10, 0x08,
      0x08, 0x10, 0x04, 0x20, 0x02, 0x40, 0x01, 0x80
};
```

The following example represents a simple 16 × 16 "X" figure in Windows.

000000		42	4D	7E	00	00	00	00	00	00	00	3E	00	00	00	28	00
000010		00	00	10	00	00	00	10	00	00	00	01	00	01	00	00	00
000020		00	00	40	00	00	00	CA	0E	00	00	C4	0E	00	00	00	00
000030		00	00	00	00	00	00	00	00	00	00	FF	FF	FF	00	7F	FE
000040		00	00	BF	FD	00	00	DF	FB	00	00	EF	F7	00	00	F7	EF
000050		00	00	FB	DF	00	00	FD	BF	00	00	FE	7F	00	00	FE	7F
000060		00	00	FD	BF	00	00	FB	DF	00	00	F7	EF	00	00	EF	F7
000070		00	00	DF	FB	00	00	BF	FD	00	00	7F	FE	00	00		

Window Manager vs. Windows Server 2003 and Windows XP

A special kind of X Windows client called the Window Manager provides a consistent working environment in the root window.

In a Windows environment, the operating system itself is the window manager and provides the desktop window. When a user logs on, the system creates three desktops within the WinSta0 windows station.

Note For additional information, search for "WinSta0" on the MSDN Web site at http://msdn.microsoft.com/.

Widgets are usually represented as controls in Windows Forms–based applications. Like the X Windows environment, Windows Forms offers many widgets. For example, X Windows dialog boxes are widgets. In Windows Forms, however, dialog boxes are not considered controls, although objects such as dialog boxes, buttons, and scroll bars are all windows.

X Library [Xlib] [X11] vs. Drawing Namespace

The X Windows library [Xlib][X11] is the lowest level library. Like the **System.Drawing** namespace, it provides all the basic drawing functions.

X Toolkit [Intrinsics] [Xt] vs. Windows Forms

The X Toolkit (Xt) is a library that accesses the lower-level graphics functionality of Xlib (X Windows) and provides UI elements such as menus, buttons, and scroll bars. It is similar to Windows Forms, which provides all the UI elements in the .NET environment.

Mapping X Windows Tools to Microsoft Windows

The primary tools for .NET development are the .NET Framework SDK and Visual Studio .NET. If Visual Studio .NET is installed in the default location, tools are found in *DriveLetter*:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools. Help provides information on these tools.

Note In some cases, the X Windows tool and the Windows tool have the same name but do not perform the same function. The **bitmap** tool is an example.

Bitmap vs. Mspaint.exe

In Visual Studio .NET, you can use the forms designer to create and edit bitmaps and icons. You can also use **Mspaint.exe** to edit bitmaps. This tool is found in the /System32 directory.

Manual Pages vs. Help

UNIX provides online documentation, which explains commands and procedures, in the form of manual pages. To access a particular manual page, type **man command_name** at the shell prompt.

Windows systems use the commands **help** and **help** *CommandName*. The Windows Help provides a similar look and feel to **man** on UNIX systems. However, most of Windows Help is found on the **Start** menu under Help. Additionally, most, if not all, of the components of the Microsoft development environment (MSDN, compilers, Visual Studio .NET, and Word) provide topical help and context-sensitive help.

At a command prompt, type **help** and press ENTER to see a list of available commands. Typing **help** followed by the name of the command (for example, typing **help setlocal**) provides information about the specified command. Windows Advanced Server installations provide ntbooks.exe in the /System32 directory. This is an excellent Help resource for all Windows server commands.

xcalc vs. Calc.exe

The **Calc.exe** tool is the Windows calculator program. It is located in the /System32 directory and provides number base conversion between decimal, hexadecimal, and binary.

xclipboard vs. Clipbrd.exe

The **Clipbrd.exe** tool is found in the /System32 directory and provides the Windows Clipboard viewing, sharing, and saving functions.

xedit vs. Notepad.exe

The **Notepad.exe** tool is a simple text editor such as **xedit**. Notepad is located in the system root directory.

xev vs. Spyxx.exe

The **Spyxx.exe** tool provides functionality similar to that of **xev**. This tool allows selection of a window and filtering of desired events and messages. The **Spyxx.exe** tool is provided in the .NET Framework SDK and can be located at *DriveLetter:*\Program Files\Microsoft Visual Studio .NET\Common7\Tools.

xfd vs. Fontview.exe

The **Fontview.exe** command-line tool enables viewing of fonts. For example, the following command displays the Modern fonts.

```
fontview modern.fon
```

However, the **Charmap.exe** tool, a GUI tool located in the /System32 directory, is a much better choice for viewing and manipulating fonts in a graphical manner.

xkill vs. Kill.exe

The Win32 **Kill.exe** tool provides the same functionality as the X Windows **xkill** command. The **Kill.exe** tool is located in the /System32 directory.

When a user presses CTRL+ALT+DEL on a Windows-based system, a dialog box appears. Click the **Task Manager** button to display the **Task Manager** dialog box. To display the process ID (PID) of the currently running tasks, click the **Processes** tab in the **Task Manager** dialog box. Locate the errant process in the list and use that PID in the **kill** command, or click the process and then click **End Process**.

xlsclients vs. Pview.exe

Like **xlsclients**, the **Pview.exe** tool lists the current running applications. **Pview** is a GUI application that can be used to select the name of a computer to view.

xlsfonts vs. Fonts Control Panel Item

The Windows Control Panel Fonts item provides all font management functionality. For more information, refer to Fonts Help in the /System/Help Fonts.chm file.

xmag vs. Magnify.exe or Zoomin.exe

The Windows **Magnify.exe** tool is equivalent to X Windows **xmag**. Magnify is located in the /System32 directory on Windows 2000 and Windows XP.

xon vs. Start.exe or Remote.exe

Like the X Windows **xon** command, the Windows **Start.exe** tool starts a new command window to run a specified program or command.

xset client vs. Control Panel Items

Windows provides a GUI interface for managing the keyboard, the mouse, and the screen. Control Panel includes an item for managing each of these devices.

The **Mode**, **Color**, and **Graftabl** commands can be used to perform some device management. To see a list of features for these three commands, type **help mode**, **help color**, or **help graftabl** at the command prompt.

If a particular application requires advanced device control, that application must provide code to perform this required functionality. Use the **SystemParametersInfo()** function to set or retrieve system-wide parameters.

User Interface Coding Examples

The following examples show how to port an X Windows–based application to Windows:

- X Windows "Hello World" example (including xHello.mak)
- .NET "Hello World" example

X Windows "Hello World" Example

The following example demonstrates the "Hello World" code for X Windows.

```
/*
** xHello.c
**
** One possible “Hello World” according to X11
**
*/
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Intrinsic.h>

char helloString[] = "Hello World";

int main( int argc, char **argv )
{
  int           iScreen;
  unsigned long ulForeground;
  unsigned long ulBackground;

  Display      *pDisplay;
  Window        exampleWindow;
  GC            gc;
  XSizeHints    sizeHints;
  XWMHints      wmHints;
  XTextProperty textProperty;
  XEvent        xEvent;

  pDisplay         = XOpenDisplay( NULL );
  iScreen          = DefaultScreen( pDisplay );

  ulBackground     = WhitePixel( pDisplay, iScreen );
  ulForeground     = BlackPixel( pDisplay, iScreen );

  sizeHints.x      = 0;
  sizeHints.y      = 0;
  sizeHints.width  = 250;
  sizeHints.height = 30;
  sizeHints.flags  = ( PPosition | PSize );

  wmHints.flags    = InputHint;
  wmHints.input    = True;

  exampleWindow    = XCreateSimpleWindow( pDisplay,
                                          DefaultRootWindow( pDisplay
),
                                          sizeHints.x,
                                          sizeHints.y,
                                          sizeHints.width,
                                          sizeHints.height,
                                          2,
                                          ulForeground,
                                          ulBackground );


  XStringListToTextProperty( &argv[0],1,&textProperty );

  XSetWMName( pDisplay, exampleWindow ,&textProperty );

  XSetWMProperties( pDisplay,
                    exampleWindow,
                    &textProperty,
                    NULL,
                    NULL,
                    0,
                    &sizeHints,
                    &wmHints,
                    NULL );

  gc = XCreateGC( pDisplay, exampleWindow, 0,0 );

  XSetBackground( pDisplay, gc, ulBackground );
  XSetForeground( pDisplay, gc, ulForeground );

  XSelectInput( pDisplay, exampleWindow, ( KeyPressMask | ExposureMask
) );

  XMapWindow( pDisplay, exampleWindow );
  do {

    XNextEvent( pDisplay, &xEvent );

    if ( xEvent.type == Expose ) {

      if ( xEvent.xexpose.count == 0 ) {

        XClearWindow( pDisplay, exampleWindow );

        XDrawImageString( pDisplay,
                          exampleWindow,
                          gc,
                          (sizeHints.width/10),
                          (sizeHints.height/2),
                          helloString,
                          (strlen( helloString )));

      }

    }

  } while (1);

  exit( 0 ) ;

}
```

(Source File: U_UIExample-UAMV4C7.01.c)

The xHello.mak File

The following example shows the X Windows xHello.mak file used with the previous "Hello World" code.

```
CC = cc
INSTALL = ./
INCLUDES = -I/usr/X11R6/include
LIBS = -L/usr/X11R6/lib -lX11 -lXaw -lXt -lXext
OBJS = xHello.o
xHello : ${OBJS}

      ${CC} -o xHello ${OBJS} ${INCLUDES} ${LIBS}

clean:
      rm -fr *.o xHello
```

.NET "Hello World" Example

The following example demonstrates the corresponding "Hello World" code in C#.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace WindowAppTest
{
            public class HelloWindowsForms : System.Windows.Forms.Form
            {
            //Label 1 displays the text message "Hello Windows Forms!"
                  private System.Windows.Forms.Label label1;

            //The constructor is where all initialization happens.
//Forms created with the designer have an //InitializeComponent()
//method.
                  public HelloWindowsForms()
                  {
//Initializes the label that will display the Hello World text
                        this.label1 = new System.Windows.Forms.Label();
//Sets the background color to black and the foreground color to white
                        this.BackColor = Color.Black;
                        this.ForeColor = Color.White;

//Specifies the location of the label in the form
```

```
                        label1.Location = new System.Drawing.Point(8,
8);
                        //Specifies the text of the label
                        label1.Text = "Hello Windows Forms!";
                        //Specifies the width and height of the label
                        label1.Size = new System.Drawing.Size(408, 48);
//Specifies the font type and style that needs to be
//used to display the text in the label
label1.Font = new System.Drawing.Font("Microsoft Sans Serif", 24f);
                        //Specifies the tab index of the label
                        label1.TabIndex = 0;
                        //Specifies the alignment of text in the label
                        label1.TextAlign =
ContentAlignment.MiddleCenter;
                        //Specifies the Form title
                        this.Text = "Hello World";
                        //Disables the maximize property of the form
                        this.MaximizeBox = false;
//Specifies the base size used for autoscaling of the form
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
                        this.FormBorderStyle =
FormBorderStyle.FixedDialog;
                        this.MinimizeBox = false;
                        this.ClientSize = new System.Drawing.Size(426,
55);
                        //Adds the label to the form
                        this.Controls.Add(label1);
                  }

            // This main function instantiates a new form and runs it.
                  public static void Main(string[] args)
                  {
                        Application.Run(new HelloWindowsForms());
                  }
            }

      } // end of namespace
```

(Source File: N_UIExample-UAMV4C7.01.cs)

Chapter 8: Developing Phase: Additional Features in .NET

This chapter describes some of the additional features of the Microsoft® .NET Framework that you can use in migrating applications from UNIX. These features include:

- Securing applications in .NET.
- Isolated storage.
- Serialization.
- .NET Remoting.
- XML Web services in .NET.
- Enterprise Services in .NET.
- Enterprise Templates.

Securing Applications in .NET

The .NET Framework and the common language runtime (CLR) provide new features that facilitate development of more secure applications. They provide many useful classes and services that enable developers to easily write more secure code. These classes and services also enable system administrators to customize code access to protected resources. In addition, the runtime and the .NET Framework provide useful classes and services that facilitate the use of cryptography and role-based security. The major security mechanisms available in the .NET Framework are described in the following sections.

Code Access Security

Code access security allows code to be trusted to varying degrees, depending on where the code originates and on other aspects of the code's identity. All managed code that targets the common language runtime receives the benefits of code access security. The following code access security concepts should always be considered in .NET:

- Writing type-safe code
- Imperative and declarative syntax
- Requesting permissions for your code
- Using secure class libraries

Role-based Security

Business applications often provide access to data or resources based on credentials supplied by the user. Typically, such applications check the role of a user and provide access to resources based on that role. .NET Framework role-based security supports authorization by making information about the *principal* available to the current thread. A principal represents the identity and role of a user and acts on the user's behalf. A principal is constructed from an associated identity; the identity can be either based on a Windows account or be a custom identity unrelated to a Windows account. .NET Framework applications can make authorization decisions based on the principal's identity or role membership, or both. Role-based security in the .NET Framework supports three kinds of principals:

- **Generic principals**. Represent users and roles that exist independent of Microsoft Windows operating system users and roles.
- **Windows principals**. Represent Windows users and their roles.
- **Custom principals**. Define an application's principals.

Note More information on role-based security is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconRole-BasedSecurity.asp.

Cryptographic Services

.NET provides various classes in the cryptography namespace to manage many details of cryptography, such as unmanaged Microsoft CryptoAPI and other managed implementations. Cryptography is used to achieve the following goals:

- **Confidentiality.** To help protect a user's identity or data from being read.
- **Data integrity.** To help protect data from being altered.
- **Authentication.** To ensure that data originates from a trusted data source.

To achieve these goals, a combination of algorithms and practices can be used as part of .NET cryptography as described as follows:

- Secret-key or symmetric encryption
- Public-key or asymmetric encryption
- Digital signatures
- Hash values
- Random number generation

Note More information about the preceding mechanisms is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconCryptographyOverview.asp.

.NET Application Security is also discussed in detail in Chapter 1, “Introduction to .NET” in this volume. More information on how to develop secured applications in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxoriSecuringApplications.asp

and

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuringyourapplication.asp.

Isolated Storage

Isolated storage is a data storage mechanism that provides isolation and safety by defining standardized methods for associating code with saved data. With isolated storage, data is always isolated by user and by assembly. Credentials such as the origin or the strong name of the assembly determine assembly identity. Data can also be isolated by the application domain, using similar credentials. When using isolated storage, applications save data to a unique data compartment that is associated with some aspect of the code's identity, such as its Web site, publisher, or signature. The data compartment is an abstraction, not a specific storage location. It consists of one or more isolated storage files, called stores, which contain the actual directory locations where data is stored.

Administrators can limit how much isolated storage an application or a user has available, based on an appropriate trust level. In addition, administrators can remove all a user's persisted data. To create or access isolated storage, code must be granted the appropriate **IsolatedStorageFilePermission**.

To access isolated storage, the code must have all necessary native platform operating system rights. Microsoft .NET Framework applications already have operating system rights to access isolated storage unless they perform (platform-specific) impersonation. In this case, the application is responsible for ensuring that the impersonated user identity has the proper operating system rights to access isolated storage. The three main classes provided to help you perform tasks that involve isolated storage are:

- **IsolatedStorageFile.** It derives from **IsolatedStorage** and provides basic management of stored assembly and application files. An instance of the **IsolatedStorageFile** class represents a single store located in the file system.
- **IsolatedStorageFileStream**. It derives from **System.IO.FileStream** and provides access to the files in a store.
- **IsolatedStorageScope**. It is an enumeration that enables you to create and select a store with the appropriate isolation type.

The isolated storage classes enable you to create, enumerate, and delete isolated storage. The methods for performing these tasks are available through the **IsolatedStorageFile** object. Some operations require you to have the **IsolatedStorageFilePermission** that represents the right to administer isolated storage. The Isolated Storage tool, **Storeadm.exe**, can also be used for simple store management such as listing or deleting all the stores for the current user.

Note More information on isolated storage is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconisolatedstorage.asp.

Serialization

Serialization is the process of converting an object or a graph of objects into a linear sequence of bytes either for storage or for transmission to another location. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications. When the object is serialized to a stream, it carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory. Deserialization is the process of taking in stored information and recreating objects from it.

The .NET Framework provides built-in mechanisms for implementing serialization and deserialization of objects such as run-time serialization, binary serialization, and XML serialization. These mechanisms are described in detail in the following section.

Run-Time Serialization

The namespace associated with run-time serialization in the .NET Framework is **System.Runtime.Serialization**. It contains classes that can be used for serializing and deserializing objects. It also implements the **ISerializable** interface, which provides a way for classes to control their own serialization behavior. Classes in the **System.Runtime.Serialization.Formatters** namespace control the actual formatting of various data types encapsulated in the serialized objects.

Two kinds of serialization can be used to serialize any object at run time. They are:

- **Binary serialization**. Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. The **System.Runtime.Serialization.Formatters.Binary** namespace contains the **BinaryFormatter** class, which can be used to serialize and deserialize objects in binary format.
- **XML serialization.** XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. The **System.Xml.Serialization** namespace contains the classes necessary for serializing and deserializing XML.

Note More information on the run-time serialization namespace is available at http://msdn2.microsoft.com/en-us/library/system.runtime.serialization.aspx.

.NET Remoting

.NET Remoting provides a rich and extensible framework for objects residing in different application domains, processes, and computers to communicate with each other seamlessly. The framework provides a number of services, including activation and lifetime support, as well as communication channels for transporting messages to and from remote applications. Formatters are used for encoding and decoding the messages that are transported by the channel. Remoting was designed with security in mind, and a number of hooks are provided that allow channel sinks to gain access to the messages and serialized stream before the stream is transported over the channel.

.NET Remoting also provides an abstract approach to interprocess communication that separates the remotable object from a specific client or server application domain and from a specific mechanism of communication. As a result, it is flexible and easily customizable. You can replace one communication protocol with another or one serialization format with another without recompiling the client or the server. In addition, the remoting system assumes no particular application model. You can communicate from a Web application, a console application, a Windows service—from almost anything you want to use.

To use .NET Remoting to build an application in which two components communicate directly across an application domain boundary, you need to build only the following:

- A remotable object.
- A host application domain to listen for requests for that object.
- A client application domain that makes requests for that object.

Note More information on building a .NET Remoting application is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbuildingbasicnetremotingapplication.asp.

XML Web Services in .NET

XML Web services are the fundamental building blocks of distributed computing on the Internet. The focus on communication and collaboration among people and applications and open standards have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where the applications reside or how the applications were implemented.

There are as many definitions of XML Web services as there are companies building them. However, almost all definitions have the following points in common:

- XML Web services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is Simple Object Access Protocol (SOAP).
- XML Web services describe their interfaces in sufficient detail to allow a user to build a client application to talk to the services. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document; it mainly consists of the metadata necessary for the client applications to consume the Web service.
- XML Web services are registered so that users can find the services easily. This is done with Universal Discovery Description and Integration (UDDI).

Note More information on XML Web services is available at http://msdn.microsoft.com/webservices/.

Enterprise Services in .NET

Component-based applications can be built using COM. However, the plumbing work required to write COM components is significant and repetitive. COM+ is not just a new version of COM; rather, COM+ provides a services infrastructure for components. The components are built and then installed in COM+ applications to build scalable server applications that achieve high throughput and are easy to deploy. (If a component does not need to use any services, then it should not be placed in a COM+ application). Scalability and throughput is achieved by designing applications that make use of services such as transactions, object pooling, and activity semantics.

The .NET Framework provides a way of writing component-based applications that has the following advantages over the COM programming model:

- Better tool support.
- Common language runtime (CLR).
- Easier coding syntax.

The .NET Framework provides a managed class into COM+ called Enterprise Services (ES) within the **System.EnterpriseServices** namespace. This simplifies the programmatic usage of the settings in COM+ catalog by reducing the COM+ component configuration and administration time.

Figure 8.1 depicts usage of COM+ and .NET.

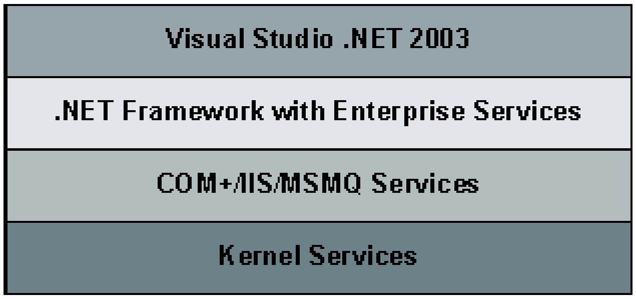


Figure 8.1. Usage of COM+ services in .NET Framework

The COM+ services infrastructure can be accessed from managed as well as unmanaged code. Services in unmanaged code are known as COM+ services. In .NET, these services are referred to as Enterprise Services. Deriving a class from the **ServicedComponent** class indicates that services will be required for a component. (If a component does not need to use any services, then it should not derive from **ServicedComponent**). The important features provided by COM+ services or the .NET Enterprise Services are:

- Automatic transaction processing
- Just In Time activation
- Object pooling
- Queued components
- Role-based security

Note More Information about the COM+ services and .NET Enterprise Services is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsummaryofservices.asp.

Enterprise Templates

Visual Studio .NET 2003 Enterprise edition provides a set of Enterprise Templates to use as a basis for developing more complex applications. They are based on the task the application is trying to accomplish instead of focusing on the programming language being used. Enterprise Templates perform three fundamental services that improve development effectiveness and reduce the overall cost of distributed applications:

- **Defining the initial structure of a distributed application**. Enterprise Templates provide architectural guidance in the form of the initial application structure and the policy that defines where various components and technologies can be used. Visual Studio .NET 2003 provides several templates targeted at generic distributed applications. Typically, this generic structure divides an application into a set of components, such as Web/Windows User Services, Business Facade/Logic, and Data Services/Storage, and populates each tier of the solution with language projects to provide the application's components, interfaces, and services.
- **Reducing complexity for developers**. An Enterprise Template makes it easier for less-experienced developers to make appropriate choices by presenting those choices that are recommended by the organization's architectural experts. Enterprise Templates implement preferred development policies, controlling available options and properties in the Visual Studio integrated development environment (IDE), and specifying custom help to guide project completion. These development policies apply to the entire application, projects, classes, and other solution items.

- **Providing architectural and technological guidance**. Architectural guidance is about using proven, reusable building blocks for an application's structure. Each Enterprise Template provides such architectural guidance by defining a basic application structure, providing the beginning project files, controlling associated development policy, and displaying appropriate developer guidance information through dynamic Help and Task List items. The policy and guidance information drives developer completion of the application's final components, interfaces, and services.

Creating an Enterprise Template in Visual Studio .NET is a two-step process. First, you must create an "Enterprise Template Project" project in Visual Studio .NET, which allows you to define the application skeleton developers receive when they create a project with your template. Next, you must expose the template to developers so that it can be accessed by the environment's **New Project** dialog box. The latter step is accomplished by converting the template into a format understood by Visual Studio .NET.

Note More information on Enterprise Templates is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsent_enterprisetemplatesbk.asp.

Along with the features described in this chapter, .NET also supports various new features in the latest versions of .NET Framework 2.0 and Visual Studio .NET 2005.

Note More information on .NET Framework 2.0 and its new features is available at

http://msdn2.microsoft.com/en-US/library/t357fb32.aspx.

More information on Visual Studio .NET 2005 and its new features is available at

http://msdn.microsoft.com/vstudio/products/newfeatures/default.aspx.

Chapter 9: Developing Phase: Deployment Considerations and Testing Activities

This chapter discusses the key development considerations for deploying applications migrated to the Microsoft® .NET Framework. It also discusses the testing activities involved in the Developing Phase. This chapter will help you identify the activities and milestones required to complete the Developing Phase.

Deployment Considerations

To ensure a smooth deployment in the upcoming Deploying Phase, you need to address the following topics in the Developing Phase:

- Process environment
- Building the application in .NET
- .NET deployment considerations
- Instrumentation

The process for deploying the migrated application is discussed in detail in Volume 5, *Deploy and Operate* of this guide.

You can use the information provided in this section to identify implementation requirements, such as environment variables and system message logging, which need to be considered when creating the migrated environment in .NET.

Process Environment

This section describes the key elements of the process environment and highlights the notable differences in these elements between UNIX and .NET. The process environment contains the following key elements:

- Environment variables
- Computer information
- Logging system messages

This section also provides the necessary information for setting up or retrieving various environment-specific details in the .NET environment.

Environment Variables

Every process has an environment block associated with it. An environment block is a block of memory allocated within the address space of the process. Each block contains a set of environment variables and their values. Both UNIX and Microsoft Windows® operating systems support the process environment blocks, although differences may exist depending on the supplier and the version of UNIX.

The .NET Framework provides an **Environment** class in the **System** namespace that provides information about, and the means to manipulate, the current environment and platform. You can use the **Environment** class to retrieve information such as command-line arguments, the exit code, environment variable settings, contents of the call stack, time since last system boot, and the version of the common language runtime (CLR).

The following C# code example accesses the environment block in .NET.

.NET example: Accessing the environment block

```
using System;
using System.Collections;
class Sample
{
      public static void Main()
      {
            String str;
            String nl = Environment.NewLine;
            Console.WriteLine("-- Environment members --");
            Console.WriteLine("ExitCode: {0}", Environment.ExitCode);
            Console.WriteLine("MachineName: {0}",
Environment.MachineName);
Console.WriteLine("OSVersion: {0}", Environment.OSVersion.ToString());
Console.WriteLine("SystemDirectory: {0}", Environment.SystemDirectory);
Console.WriteLine("UserDomainName: {0}", Environment.UserDomainName);
            Console.WriteLine("UserName: {0}", Environment.UserName);
Console.WriteLine("Version of the CLR: {0}",
Environment.Version.ToString());
String query = "My system drive is %SystemDrive% and my system root is
%SystemRoot%";
            str = Environment.ExpandEnvironmentVariables(query);
            Console.WriteLine("ExpandEnvironmentVariables: {0}  {1}",
nl, str);
Console.WriteLine("GetEnvironmentVariable: {0}  My temporary directory
is {1}.", nl,
            Environment.GetEnvironmentVariable("TEMP"));
            Console.WriteLine("GetEnvironmentVariables: ");
IDictionary    environmentVariables =
Environment.GetEnvironmentVariables();
            foreach (DictionaryEntry de in environmentVariables)
            {
                  Console.WriteLine("  {0} = {1}", de.Key, de.Value);
            }
      }
}
```

(Source File: N_ProcessEnv-UAMV4C9.01.cs)

Computer Information

At times, it is necessary to obtain information about a computer. This is particularly important when an application is designed to support multiple users or different types of hardware and operating systems. Some information that an application may require are:

- The host name.
- The operating system name.
- The network name of the computer.
- The release level of the operating system.
- The version number of the operating system.

In UNIX, the **gethostname** and **uname** functions are used to obtain this information. In.NET Framework, there are two classes—the **System.Environment** class and the **System.Windows.Forms.SystemInformation** class—that provide the system information. The following code example in UNIX prints out the system information, such as the computer host name, node name, operating system release, and version number, to the console.

UNIX example: Printing the system information

```
#include <unistd.h>
#include <stdio.h>
#include <sys/utsname.h>
int main()
{
    char computer[256];
    struct utsname uts;

    if(gethostname(computer, 255) != 0 || uname(&uts) < 0) {
        fprintf(stderr, "Could not get host information\n");
        exit(1);
    }

    printf("Computer host name is %s\n", computer);
    printf("Nodename is %s\n", uts.nodename);
    printf("Version is %s, %s\n", uts.release, uts.version);
    exit(0);
}
```

(Source File: U_SysInfo-UAMV4C9.01.c)

The following example in C# displays some of the system information, such as the computer host name, domain name, user name of the currently logged on user, the operating system version number, and the current platform to the console using the **System.Environment** class. If you are developing a Windows Forms application, then consider using the **System.Windows.Forms.SystemInformation** class.

.NET example: Printing the system information

```
using System;
public class MySysInfo
{
      static void Main()
      {
      //Creates an object of OperatingSystem class
      OperatingSystem obOS = System.Environment.OSVersion;
      //Displays the Host name of the Computer
      Console.WriteLine("Computer 
Name:{0}",System.Environment.MachineName);
      //Displays the Domain name of the computer
      Console.WriteLine("Domain 
Name:{0}",System.Environment.UserDomainName);
      //Displays the log-in name of the user currently logged in
      Console.WriteLine("Logged In 
User:{0}",System.Environment.UserName);
      //Displays the OS Version No(Major,Minor and Revision No 
combined)
      Console.WriteLine("OS Version:{0}",obOS.Version.ToString());
      //Displayes the current platform
      Console.WriteLine("Platform:{0}",obOS.Platform.ToString());

      }
}
```

(Source File: N_SysInfo-UAMV4C9.01.cs)

Note More information on the **Environment** class and its members is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemEnvironmentClassTopic.asp.

More information on the **SystemInformation** class and its members is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemwindowsformssysteminformationmemberstopic.asp.

Logging System Messages

In UNIX, diagnostic messages are logged by writing the formatted output to the system logger. The diagnostic messages are written to the system log files, such as USERS, or forwarded to the appropriate computer. If a log daemon process is not running, the log information may be written to a standard log file, such as /var/adm/log/logger.

Table 9.1 lists the numerous levels of logged information contained in the daemon syslogd in UNIX.

Table 9.1. UNIX Logging System Messages

Priority	Description
LOG_EMERG	A panic condition.
LOG_ALERT	A condition that should be corrected immediately.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warnings.
LOG_NOTICE	Non-error–related conditions.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages intended for debug purposes.

The .NET Framework provides an **EventLog** class that interacts with the Windows event logs. The **EventLog** class enables you to access or customize Windows event logs, which record information about important software or hardware events. Using **EventLog**, you can read from the existing logs, write entries to logs, create or delete event sources, delete logs, and respond to log entries. The **CreateEventSource** method is used to create an event source for writing entries to a log. While creating an event source, you can specify the name of the log such as application, security, system, or custom event log. The **WriteEntry** method can be used to write any customized message into the event source. The event type of the event log can also be specified in this method. This type can be one of the following examples listed in Table 9.2.

Table 9.2. .NET EventLog Class Event Types

Name of Event	Description
Error	This indicates a significant problem the user should know about, usually a loss of functionality or data.
FailureAudit	This indicates a security event that occurs when an audited access attempt fails.
Information	This indicates a significant, successful operation.
SuccessAudit	This indicates a security event that occurs when an audited access attempt is successful.
Warning	This indicates a problem that is not immediately significant but that may signify conditions that could cause future problems.

UNIX example: System logging

```
#include <syslog.h>
#include <stdio.h>

int main()
{
    FILE *fp;

    fp = fopen("Bad_File_Name","r");
    if(!fp)
        syslog(LOG_INFO|LOG_USER,"error - %m\n");
    fclose(fp);
    exit(0);
}
```

(Source File: U_SysLog-UAMV4C9.01.c)

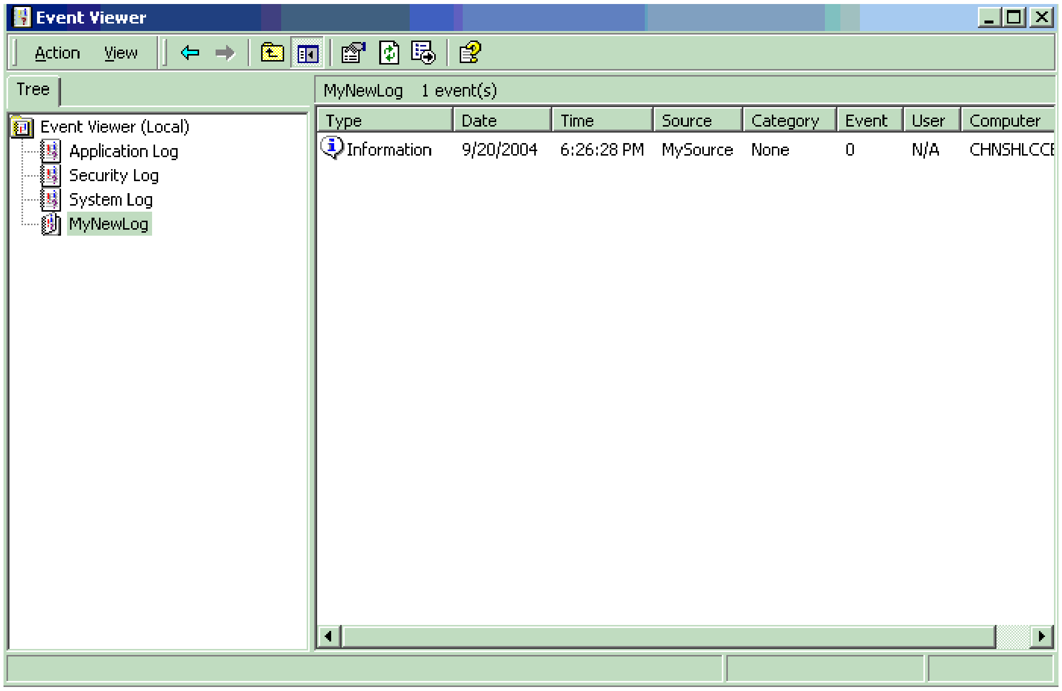
The message in this example would be logged to /var/log/messages on a typical Linux system, to /var/adm/messages on a Solaris system, and to /var/adm/log/messages on Interix (when syslogd in running). Refer to the /etc/syslog.conf file for more specific information—specifically, a *.info entry that specifies the file where the preceding message will be logged.

.NET example: System logging

```
using System;
using System.Diagnostics;
using System.Threading;

class MySample{

    public static void Main(){

        // Create the source, if it does not already exist.
        if(!EventLog.SourceExists("MySource")){
            EventLog.CreateEventSource("MySource", "MyNewLog");
            Console.WriteLine("CreatingEventSource");
        }

        // Create an EventLog instance and assign its source.
        EventLog myLog = new EventLog();
        myLog.Source = "MySource";

        // Write an informational entry to the event log.
        myLog.WriteEntry("Writing to event log.");

    }
}
```

(Source File: N_SysLog-UAMV4C9.01.cs)

If the log that you specify in a call to **CreateEventSource** does not exist on the computer, the system creates a custom log and registers the application as a source for that log. You can use the **EventLog** class to read and write entries to any event log for which you have the appropriate access. Figure 9.1 depicts a new log, MyNewLog, created using the **CreateEventSource** call. As the log did not initially exist on the computer, the log is created after running the Eventvwr.exe code.

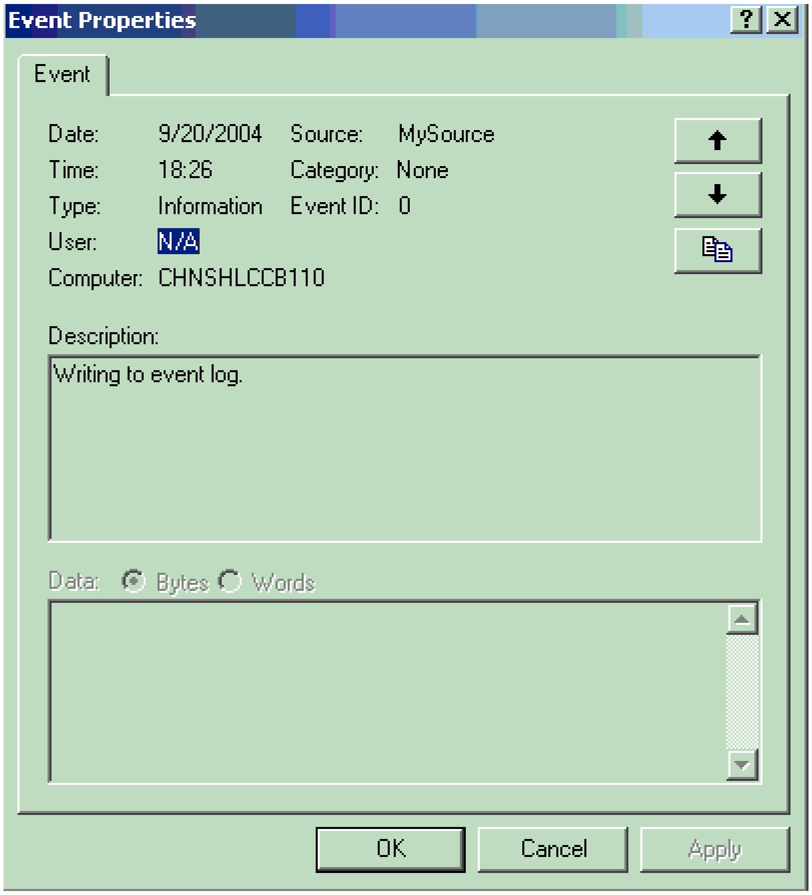


MyNewLog 1 event(s)

Type	Date	Time	Source	Category	Event	User	Computer
Information	9/20/2004	6:26:28 PM	MySource	None	0	N/A	CHNSHLCCE

Figure 9.1. Windows Event Viewer

Double-clicking the **Information** option in the **Type** column opens a detailed view of the event, as depicted in Figure 9.2.



Date: 9/20/2004 Source: MySource
Time: 18:26 Category: None
Type: Information Event ID: 0
User: N/A
Computer: CHNSHLCCB110

Description:
Writing to event log.

Data: ● Bytes ○ Words

Figure 9.2. Details of an event in Windows Event Viewer

This is a very simple example of generating log information and posting it to the Windows event log.

Note More details and complexities of event logging in Windows are available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconIntroductionToEventLogComponents.asp.

Building the Application in .NET

Microsoft Visual Studio® .NET 2003 offers a variety of methods for organizing the files that are to be included in a build of a solution or a project, setting the project properties in effect while building and arranging the order in which the projects is to be built.

Following are the common Visual Studio procedures for preparing and managing builds.

To build or rebuild an entire solution

9. In Solution Explorer, select or open the desired solution.
10. On the **Build** menu, click **Build Solution** or **Rebuild Solution** as per the following requirements:
    - Click **Build** <***ProjectName***> to compile only the necessary project files and components in the project named <*ProjectName*> of the solution.
    - Click **Build Solution** to compile only those project files and components in all the projects of the solution.
    - Click **Rebuild Solution** to clean the solution first, and then build all the project files and components.

      Note "Cleaning" a solution or project deletes all the intermediate and output files, leaving only the project and component files, from which new instances of the intermediate and output files are built.

Note More information on building the application in .NET using the Visual Studio .NET integrated development environment (IDE) is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxtskPrepareandManageBuilds.asp.

.NET applications can also be built from the command line. The syntax for building a .NET application from the command line is given as follows:

devenv /build ConfigName [/project ProjName] [/projectconfig ConfigName] SolutionName .

The summary information for builds, including errors, appears in the command prompt. It can also be redirected to any log file using the /out argument.

Note More information on the available command-line parameters for building the application from the command line is available at

http://msdn.microsoft.com/library/en-us/vsintro7/html/vxlrfBuild.asp.

.NET Deployment Activities

Deploying an application involves at least two different activities: packaging the code and distributing the packages to the various clients and servers on which the application is to be deployed. One of the primary goals of the .NET Framework is to simplify the deployment process, especially the distribution aspect, by making the zero-impact install and an XCopy deployment feasible. The following subsections describe the most important .NET deployment considerations: assemblies, configuration files, and packaging tools.

Assemblies

The .NET Framework introduces *assemblies*, which form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. Assemblies act as the smallest distribution unit for the component code in the .NET Framework. Assemblies are self-describing deployment units; they are self-describing through metadata called a manifest. The .NET Framework uses the metadata to describe the types as well as the assemblies that contain the types.

Assemblies consist of four elements: the assembly metadata (manifest), metadata describing the types, the intermediate language (IL) code that implements the types, and a set of resources. Not all of these elements are present in each assembly. The manifest is mandatory for all assemblies and either types or resources are needed to give the assembly any meaningful functionality. Versioning in .NET is done at the assembly level.

There are two types of assemblies in the .NET infrastructure: private and shared.

Private Assemblies

Private assemblies are not designed to be shared. They are designed to be used by one application and must reside in the directory or subdirectory of that application. By default, all assemblies are private. An XCopy deployment can only be done for private assemblies.

Shared Assemblies

For the components that must be distributed, .NET offers the shared assembly. The shared assembly concept is centered on two principles. The first, called side-by-side execution, allows the CLR to house multiple versions of the same component on a single computer. The second, termed binding, ensures that the clients obtain the version of the component that they expect.

Shared assemblies have the following two issues:

- Shared assemblies can be updated by anyone in the absence of a proper authorization.
- If an assembly is shared by more than one party and one or more components in the assembly are updated without the knowledge of the other sharing parties, the clients may receive the incorrect version of one or more requested components.

The first issue can be resolved by using a private key to sign an assembly, allowing only the developer signing the key to update the assembly. Using the Global Assembly Cache (GAC), a computer-wide code cache that exists on a computer where the CLR is installed, can resolve the second issue. GAC can house multiple copies of a shared assembly based on the signature and the version information used to build it. Before installing the assemblies to GAC, they should be strongly named using the Strong Name tool (**Sn.exe**), provided with Visual Studio .NET 2003. The Global Assembly Cache tool (**GACUtil.exe**) allows you to view and manipulate the contents of the global assembly cache. This information (signature and version) is stored in the manifest of all clients who want to access the assembly, allowing the CLR to load the appropriate version at run time.

Note More information about assemblies in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblies.asp.

More information on Global Assembly Cache (GAC) is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconglobalassemblycache.asp.

Configuration Files

Configuration files in .NET are XML files that can be changed as required. Developers can use the configuration files to change the settings without recompiling the applications. Administrators can use the configuration files to set policies that affect how the applications run on the computers. This section helps you understand the .NET configuration files and their role in .NET applications.

The .NET Framework provides the **System.Configuration** class to read the setting from the configuration files. There are three types of configuration files in .NET:

- Machine configuration files
- Application configuration files
- Security configuration files

Note More information on configuration files in .NET and their formats is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfigurationfiles.asp.

Machine Configuration Files

The machine configuration file, Machine.config, contains settings that apply to an entire computer. This file is located in the %runtime install path%\Config directory. Machine.config contains configuration settings for the machine-wide assembly binding, built-in remoting channels, authentication modes, and authorization modes for ASP.NET applications.

The configuration system first looks in the machine configuration file for the <appSettings> element and other configuration sections that a developer might define. After that, the configuration system looks in the application configuration file.

Application Configuration Files

It is possible to configure an application by using a configuration file in the same directory as the application. The name of the configuration file is the name of the application with a .config extension for Windows applications or Web.config for the Web applications. Therefore, a configuration file for the ListBoxAdd application is called ListBoxAdd.exe.config.

The application configuration file contains settings specific to an individual application. This file contains configuration settings that the CLR can read (such as assembly binding policy and remoting objects) and settings that the application can read (such as the location of the files and database connection string). The assembly binding policy that defines how assemblies should be probed is also configured in this file.

If probing fails to find an assembly in either the application directory or a subdirectory with the same name as the assembly, it next looks for probing instructions in the application configuration file (if one exists).

Here is an example of an application configuration file:

```
<configuration>
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
     <probing privatePath="AssemblyDir1;AssemblyDir2"/>
  </assemblyBinding>
</runtime>
</configuration>
```

This application configuration file uses a probing tag to set the private path to be probed to include two directories, called AssemblyDir1 and AssemblyDir2. These directories are considered as subdirectories that exist below the application directory.

Security Configuration Files

The security configuration file contains information about the code group hierarchy and the permission sets associated at a policy level. It is strongly recommended that you use the .NET Framework Configuration tool (**Mscorcfg.msc**) or Code Access Security Policy tool (**Caspol.exe**) to modify the security policy to ensure that the policy changes do not corrupt the security configuration files.

Note More information on the .NET Framework Configuration tool is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconNETFrameworkAdministrationToolMscorcfgmsc.asp.

More information on the Code Access Security Policy tool is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfCodeAccessSecurityPolicyUtilityCaspolexe.asp.

More information on configuration files in .NET is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfigurationfiles.asp.

Manifests

Manifests are XML files used to describe the side-by-side execution of assemblies or the isolated application through the assembly's <assemblyIdentity> element. Side-by-side execution is the process of maintaining multiple of versions of the same assembly on a single computer. Manifests contain information used for binding and activation. Side-by-side assemblies are not registered on the system but are available to applications and other assemblies on the system that specify dependencies in manifest files. Side-by-side assemblies can be installed as shared assemblies or as private assemblies.

The following types of manifests are used with side-by-side assemblies:

- **Assembly manifests describe side-by-side assemblies.** These are used to manage the names, versions, resources, and dependent assemblies of side-by-side assemblies. The manifests of shared assemblies are stored in the WinSxS folder of the system. Private assembly manifests are stored either as a resource in the DLL or in the application folder.
- **Application manifests describe isolated applications.** These are used to manage the names and versions of shared side-by-side assemblies that the application should bind to at run time. Application manifests are copied into the same folder as the application executable file or included as a resource in the application's executable file.
- **Application Configuration Files.** These are manifests used to override and redirect the versions of dependent assemblies used by side-by-side assemblies and applications.
- **Publisher Configuration Files.** These are manifests used to redirect the version of a side-by-side assembly to another compatible version. The version that the assembly is being redirected to should have the same major.minor values as the original version.

Packaging Tools

The .NET Framework comes with tools that allow developers to quickly build and deploy robust applications that take advantage of the new CLR environment to provide a fully managed and feature-rich application-execution environment. The .NET Framework also provides the following benefits:

- Improved isolation of application components.
- Simplified application deployment.
- Robust version numbering.

The .NET Framework SDK includes several useful tools for examining assemblies and working with the system assembly cache. Following is the list of the tools:

- **Assembly Linker (Al.exe)**.For creating assembly manifests, satellite assemblies, and working with the Global Assembly Cache (GAC).
- **Global Assembly Cache Tool (Gacutil.exe)**. Console tool that manages the Global Assembly (GAC) and download caches.
- **MSIL Disassembler (Ildasm.exe)**. Windows-based tool for examining the manifest (containing metadata) and MSIL code inside assemblies.
- **Assembly Binding Log Viewer (Fuslogvw.exe)**. Windows-based tool for examining assembly and resource bind requests.
- **Strong Name Tool (Sn.exe)**. Console tool to help generate strongly named assemblies.

Note More information on these tools is available at http://msdn.microsoft.com/library/en-us/cptutorials/html/Appendix_B___Packaging_and_Deployment_Tools.asp.

The following five packaging options are available in the .NET Framework:

- **As-built (DLLs and EXEs)**. An application can be deployed in the format produced by building the application using the development tool. In many scenarios, no special packaging is required.
- **CAB files**. CAB files can be used to compress an application for more efficient downloads.

  CAB files provide the following benefits:

  - Delivery of all application files in one file.
  - Prevention of partial installations.
  - Capability to install an application from multiple sources.

  You can use Microsoft Visual Studio .NET 2003 to generate a CAB file automatically, or you can create a custom CAB file. You can distribute the CAB files for an application from a variety of sources, including a Web site, a memory storage card, another device, or a server or desktop computer.

  Note More information on CAB file projects is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconcabfileprojects.asp.

- **Microsoft Windows Installer packages**. Visual Studio .NET 2003 and other installation tools allow developers to build Windows Installer packages (.msi files), which enable developers to take advantage of the application repair, on-demand install, and other Microsoft Windows application-management features.

  Windows Installer is based on a data-driven model that provides all installation data and instructions in a single package. In contrast, traditional scripted setup programs were based on a procedural model, providing scripted instructions for application installations. Scripted setup programs focused on how to install something, whereas Windows Installer focuses on what to install.

  With Windows Installer, each computer keeps a database of information about every application that it installs, including files, registry keys, and components. When an application is uninstalled, the database is checked to ensure that no other applications rely on a file, registry key, or component before removing it. This prevents breaking of other applications because of the removal of an application.

  Windows Installer also supports self-repair, the capability of an application to automatically reinstall missing files that may have inadvertently been deleted by the user. In addition, Windows Installer provides the capability to roll back an installation. For example, if an application relies on a specific database and the database is not found during the installation, installation can be aborted and the computer returned to its preinstallation state.

  The deployment tools in Visual Studio .NET 2003 build on the foundation of Windows Installer, providing you with rich capabilities for rapidly deploying and maintaining applications built with Visual Studio .NET 2003.

  Note More information of Windows Installer is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconwhatyouneedtoknowaboutmicrosoftwindowsinstaller.asp.

- **Merge module projects**. Merge module projects allow developers to create reusable setup components. A merge module (.msm file) is a single package that contains all the files, resources, registry entries, and setup logic necessary to install a component. Merge modules cannot be installed alone, but must be used within the context of a Windows Installer (.msi) file.

  Note More information on merge module projects is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconMergeModuleProjects.asp.

- **Setup projects**. Setup projects allow developers to create installers to distribute an application. The resulting Windows Installer (.msi) file contains the application, any dependent files, and information about the application, such as registry entries and instructions for installation. There are two types of setup projects in Visual Studio: Setup projects and Web Setup projects. The distinction between a Setup project and a Web Setup project is the location where the Windows Installer will be deployed. Setup projects will install files into the file system of a target computer; whereas Web Setup projects install files into a virtual directory of a Web server.

  Note More information on Setup projects is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconSetupProjects.asp.

- **ClickOnce**. ClickOnce is the feature supported by the latest versions of Visual Studio .NET 2005. ClickOnce provides a new deployment technology that gives the best of a rich Windows-based application user experience and the deployment and maintenance benefits of Web applications. It allows a user to download and execute a rich client application over the Web, off a network file share, or from local media. It offers a full Windows user interface running on a desktop, while allowing single server deployment of application files and updates. Client applications are automatically deployed and updated on the user's computer from the deployment server in a safe way and will not affect other applications or data that already exists on the computer.

  Note More information on ClickOnce is available at

  http://msdn.microsoft.com/msdnmag/issues/04/05/ClickOnce/.

The following are three third-party products that you can use for packaging the developed application:

- **InstallShield Developer**. More information on InstallShield Developer is available at http://www.installshield.com.
- **Wise for Windows Installer**. More information on Wise for Windows Installer is available at http://www.wise.com/.
- **Veritas WinINSTALL**. More information on Veritas WinINSTALL is available at http://www.veritas.com.

Note More information on deploying the applications in Visual Studio .NET 2003 is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vboriDeploymentInVisualStudio.asp.

Instrumentation

Windows Management Instrumentation (WMI) provides a rich set of system management services built into the Microsoft Windows operating system. The use of WMI-based management systems leads to a more robust computing environment and an increased level of system reliability, which allows the automation of administrative tasks in an enterprise environment.

In the .NET Framework, the **System.Management** namespace provides common classes to traverse the WMI schema. In addition to the .NET Framework, you must have WMI installed on the computer to make use of the management features in this namespace.

Note More information on instrumentation in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmanagingapplicationsusingwmi.asp.

Testing Activities

This section discusses the testing activities designed to identify and address potential solution issues prior to deployment. Testing starts when you begin developing the solution and ends when the testing team certifies that the solution components meet the schedule and quality goals established in the project plan.

Testing in migration projects involving infrastructure services is focused on finding discrepancies between the behavior of the original application, as seen by its clients, and the behavior of the newly migrated application. All discrepancies must be investigated and fixed.

In the Developing Phase, the testing team executes the test plans for acceptance tests on the application submitted for a formal round of testing on the test environment. The testing team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components address the project goals.

The inputs required for the Developing Phase include:

- Functional specifications document.
- A feature-complete application, which has been unit tested.

The documents that are used during the Developing Phase include:

- **Test plan**. The test plan is prepared during the Planning Phase. It should describe in detail everything that the test team, the program management team, and the development team must know about the testing to be done.
- **Test specification**. The test specification conveys the entire scope of testing required for a set of functionality and defines individual test cases sufficiently for the testers. It also specifies the deliverables and the readiness criteria.
- **Test environment**. The test environment is an exact replica of the production environment; it is used to test the application under realistic environments. It also describes the software, hardware, and tools required for testing purposes.
- **Test data**. The test data is a set of data for testing the application. Test data is usually a diverse set of data that helps test the application under different conditions.
- **Test report**. The test report is an error report of the tests performed. It includes a description of the errors that occurred, steps to reproduce the errors, severity of the errors, and names of the developers who are responsible for fixing them.

  The test report is updated during the Stabilizing Phase and is also one of the outputs of this phase, along with the tested and stabilized application.

The key deliverables of the Developing Phase include:

- Application ready to be deployed in the production environment.
- Application source code.
- Project documentation and user manual.
- Test plan, test specification, and test reports.
- Release notes.
- Other project-related documents.

Testing begins with a code review of the application and unit testing. In the Developing Phase, the application is subjected to various tests. The test plan organizes the testing process into the following elements:

- Code component testing
- Integration testing
- Database testing
- Security testing
- Management testing

You can test the migrated application in all the scenarios by using a defined testing strategy. Although each test has a different purpose, together they verify that all system elements are properly integrated and perform their allocated functions.

Visual Studio .NET 2005 provides the integrated test environment along with the integrated development environment (IDE).

Note More information on Testing features of Visual Studio .NET 2005 is available at http://msdn.microsoft.com/vstudio/products/newfeatures/209/default.aspx.

Code Component Testing

A component may be a class or a group of related classes performing a similar task. Component testing is the next step after unit testing. Component testing is the process of verifying a software component with respect to its design and functional specifications.

Component testing in a migration project is the process of finding the discrepancies between the functionality and output of components in the Windows application and the original UNIX application. Basic smoke testing, boundary conditions, and error test cases are written based on the functional specification of the component.

Code component testing tests the components for the following:

- Functionality
- Input and output, interactions within and with other components
- Stress testing
- Performance

The test cases for component testing cover, either directly or indirectly, constraints on their inputs and outputs (pre-conditions and post-conditions), the state of the object, interactions between methods, attributes of the object, and other components.

The code component testing requires the following inputs:

- **Test plan and specification**. It provides the test cases.
- **System requirements**. These are used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of a component must be tested for vulnerabilities.
- **Specifications of the component**. The specifications are used to build the functional test cases. Information on the component inputs, outputs, and interactions with other components can be derived from here.
- **Design document**. The actual implementation of the design provides the information necessary to construct the structural and interaction test cases.

Components must also be stress tested. Stress testing is the process of loading the component to the defined and undefined limits. Each component must be stressed under a load to ensure that it performs well within a reasonable performance limit.

System CPU and memory usage per component can also be measured and monitored to determine the performance of individual components. For this, you can use such tools as the Windows Performance Monitor. For more information, refer to the "Testing and Optimization Tools" section of Chapter 10, “Stabilizing Phase” of this volume.

Integration Testing

Integration testing involves testing the application as a whole, with all the components of the application put together. Component testing is done during the testing performed in the Developing Phase. Integration testing is the process of verifying the application with respect to the behavior of components in the integrated application, interaction with other components, and the functional specifications of the application as a whole. Integration testing in a migration project is the process of finding discrepancies in the interaction between components and the behavior of components in the Windows application and the original UNIX application.

Integration testing tests the components for:

- Functionality: behavior of the application as a whole and the individual components after integration.
- Input and output: interactions within and with other components.
- Response to various types of stresses.
- Performance.

Test cases for integration testing directly or indirectly include functionality of the components, constraints on their inputs and outputs (pre-conditions and post-conditions), the state of the object, interactions between components, attributes of the object, and other components. The application must also be stress tested. Inputs required for integration testing include:

- **Test plan**. It provides the details of testing the application.
- **Test specification**. It is used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of the application must be tested for vulnerabilities.

Stress testing must also be performed. Stress testing is the process of loading the application to the defined and undefined limits to ensure that it performs well within a reasonable performance limit.

System testing is also performed after completion of integration testing. System testing is the process of ensuring that the integrated application is compatible with all platforms and to test against its requirements. The system CPU and memory usage for the application can also be measured and monitored to determine their performance. For this, you can use such tools as the Windows Performance Monitor.

Note For more information, refer to the "Testing and Optimization Tools" section of Chapter 10, "Stabilizing Phase."

Database Testing

The database component is a critical piece of any data-enabled application. In a migration project, the database may be the same or may have been replaced by another database. In both cases, data must be migrated to the respective database on Windows. Testing of a migrated database includes testing of:

- Migrated procedural code.
- Data integration with heterogeneous data sources (if applicable).
- Customized data transformations and extraction.

Database testing also involves testing at the data access layer, which is the point at which your application communicates with the database. Database testing in a migration project involves:

- Testing the data and the structure and design of the migrated database objects.
- Testing the procedures and functions related to database access.
- Security testing, which tests the database to guarantee proper authentication and authorization so that only users with the appropriate authority access the database. The database administrator must establish different security settings for each user in the test environment.

i. Testing of data access layer.
- Performance testing of data access layer.
- Manageability testing of the database.

An application maintains the following three databases, which are replicas of each other:

- **Development database**. This is where most of the testing is carried out.
- **Deployment database (or integration database)**. This is where the tests are run prior to deployment to ensure that the local database changes are applied.
- **Live database**. This has the live data; it cannot be used for testing.

Database testing is done on the development database during development, and the integrated application is tested using the deployment database.

Security Testing

Security is about controlling access to a variety of resources, such as application components, data, and hardware. Security testing is performed on the application to ensure that only users with the appropriate authority are able to use the applicable features of the application. Security testing also involves testing the application from the point of view of providing the same security features and measures that were provide by the original application.

To ensure that the application is as secure as possible, most security measures are based on the following four concepts:

- **Authentication**. This is the process of confirming the identity of the users, which is one layer of security control. Before an application can authorize access to a resource, it must confirm the identity of the requestor.
- **Authorization**. This is the process of verifying that an authenticated party has the permission to access a particular resource, which is the layer of security control following the authentication layer.
- **Data protection**. This is the process of providing data confidentiality, integrity, and nonrepudiability. Encrypting the data provides data confidentiality. Data integrity is achieved through the use of hash algorithms, digital signatures, and message authentication codes. Message authentication codes (MACs) are used by technologies such as SSL/TLS to verify that data has not been altered while in transit.
- **Auditing**. This is the process of logging and monitoring events that occur in a system and are of interest to security.

  Note For more information, refer to "Event Logging" on the TechNet Web site at http://technet2.microsoft.com/WindowsServer/en/Library/0473658c-693d-4a06-b95b-ebe8a76648a91033.mspx.

The systems engineer establishes different security settings for each user in the test environment. Network security testing is performed to help secure the network from unauthorized users. To minimize the risks associated with unchecked errors on the system, you should know the user context in which system processes run, keeping to a minimum the privileges that these accounts have, and log their access to these accounts. Active monitoring can be accomplished using the Windows Performance Monitor for real-time feedback.

All security settings and security features of the application must be documented properly.

Notes

More information about security testing is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingforsecurability.asp.

More information on how to make your code more secure is available at http://msdn.microsoft.com/security/securecode/.

More information on "Secure Coding Guidelines for the .NET Framework" is available at http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp.

Management Testing

Testing for manageability involves testing the deployment, maintenance, and monitoring technologies that you have incorporated into your migrated application.

Following are some important testing recommendations to verify that you have developed a manageable application:

- **Test Windows Management Instrumentation (WMI)**. WMI can provide important information about your application and the resources it uses. During the design of your application, you made certain decisions about the types of WMI information that must be provided. These might include server and network configurations, event log error messages, CPU consumption, available disk space, network traffic, application settings, and many other application messages. You must test every source of information and be certain you can monitor each one.
- **Test Network Load Balancing (NLB) and cluster configuration.**You can use Application Center 2000 clustering to add a front- or back-end server while the application is still running. After installing new server hardware on the network, use your monitoring console to replicate the application image and start the server. The new server should automatically begin sharing some of the workload. You can set up the Application Center 2000 Performance Monitor (PerfMon) to track multiple front-end Web servers. After setting up PerfMon, make some requests to generate traffic. PerfMon will show you that there is an increase in traffic in the back-end servers and that the workload is evenly spread across the front-end computers.

  Note Additional information about Application Center 2000 is available at

  http://www.microsoft.com/applicationcenter/.
- **Test change control procedures**. An important part of application management is the handling of both scheduled and emergency maintenance changes. Test and validate all of the change control procedures including the automated and manual processes. It is especially important to test all people-based procedures to ensure that the necessary communication, authority, and skills are available to support an error-free change control process.

  Note Additional information on testing for manageability is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingformanageability.asp.

Interim Milestone: Internal Release *n*

The project needs interim milestones to help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward the completion of the solution featuresets and achievement of the associated quality level. Depending on the complexity of the solution, a number of internal releases may be required. Each internal release represents a fully functional addition to the solution’s core feature set, indicating that it is potentially ready to move on to the Stabilizing Phase.

Closing the Developing Phase

Closing the Developing Phase requires completing a milestone approval process. The team documents the results of different tasks that it has performed in this phase and obtains a sign-off on the completion of development from the stakeholders (including the customer).

Key Milestone: Scope Complete

The Developing Phase culminates in the Scope Complete Milestone. At this milestone, all features are complete and the solution is ready for external testing and stabilization. This milestone provides the opportunity for customers and users, operations and support personnel, and key project stakeholders to evaluate the solution and identify any remaining issues that must be addressed before beginning the transition to stabilization and, ultimately, to release.

Key stakeholders, typically representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the milestone is complete. The sign-off document becomes a project deliverable and is archived for future reference.

Now the team must shift its focus to verify that the quality of the solution meets the acceptance criteria for release readiness. The next phase, the Stabilizing Phase, describes the activities—for example, user acceptance testing (UAT), regression testing, and conducting the pilot—required to achieve these objectives.

Chapter 10: Stabilizing Phase

This chapter describes the suggested strategy for stabilizing an application that has been migrated from UNIX to the Microsoft® Windows® operating system. The Stabilizing Phase involves testing the application for the expected functionality and improving the quality of the application to meet the acceptance criteria set for the project.

This chapter describes the objectives of testing in the Stabilizing Phase. It introduces testing processes, methodologies, and tools that can be used to test applications with different architectures.

Goals for the Stabilizing Phase

The primary goal of the Stabilizing Phase is to improve the quality of the solution so that it meets the acceptance criteria and can be released to the production environment. During this phase, the team tests the feature-complete migrated application by subjecting it to various tests, such as User Acceptance Testing (UAT), regression testing, and bug tracking based on the application requirements. The build must demonstrate that it reaches the defined quality and performance levels and is ready for full production deployment.

Testing during the Stabilizing Phase is an extension of the testing that was conducted during the development of the application in the Developing Phase. Testing in the Stabilizing Phase tests the usage and operation of the application under realistic conditions. Test plans include testing the functionality in the migrated application and making a comparison of the migrated application’s functionality with that provided by the original application. Test plans also must include test cases for testing the new features added to the application.

After a build is stabilized, the solution is deployed. This phase ends with the Release Readiness Approved Milestone, indicating that the team and customer agree that all the outstanding issues have been addressed.

Major Tasks and Deliverables

Table 10.1 describes the tasks that must be completed during the Stabilizing Phase and lists the owners responsible for achieving them.

Table 10.1. Major Stabilizing Phase Tasks and Owners

Major Tasks	Owners
Testing the solution The team executes the test cases that were created during the Planning Phase and enhanced and tested during the Developing Phase. Testing includes comparing the test results of the parent application with the migrated application as well as testing the application from different perspectives.	Test
Resolving defects The team triages the defects identified and resolves them. New tests are developed to reproduce issues reported from other sources. The new test cases are integrated into the test suite.	Development, Test
Conducting the solution pilot This task involves setting up the deployment environment and testing the migrated application on the staging area before it is deployed. The team moves a solution pilot from the development area to a staging area in order to test the solution with the actual users and real scenarios. It also includes testing the solution in a live environment. The solution pilot is conducted before starting the Deploying Phase.	Release Management
Closing the Stabilizing Phase The team documents the results of the tasks performed in this phase and solicits management approval at the Release Readiness Approved Milestone meeting.	Project

Table 10.2 lists the tasks described in Table 10.1 and considers the tasks from the perspective of the team roles. The primary team roles driving the Stabilizing Phase are Test and Release Management.

Table 10.2. Role Cluster Focuses and Responsibilities in Stabilizing Phase

Role Cluster	Focus and Responsibility
Product Management	Execute communications plan and launch test phase.
Program Management	Track project and bug triage.
Release Management	Preparation for deployment of the application and setting up the production environment.
Development	Bug triage and resolution, code optimization, and hardware or service reconfiguration.
User Experience	Stabilization of user documentation and training materials.
Test	Generate build and triage plan. Track test schedule. Review bugs entered in the bug-tracking tool and monitor their status during triage meeting. Generate weekly status reports. Escalate issues that are blocking progress, review impact analysis, and generate change management document. Ensure that the appropriate level of testing is achieved for a particular release. Lead the actual Build Acceptance Test (BAT) execution. Execute test cases and generate test report.

Testing the Solution

This section describes the testing activities that are performed in the Stabilizing Phase. In the Stabilizing Phase, because all features and functions of the solution are now complete and all solution elements have been built, testing is performed on the solution as a whole, not just on individual components. The testing that began during the Developing Phase according to the test plan created during the Planning Phase continues with further testing, tracking, documentation, and reporting activities during the Stabilizing Phase. This mainly involves user acceptance testing (UAT) and regression testing as explained in the next subsections in detail.

User Acceptance Testing

The emphasis on user acceptance testing (UAT) during the Stabilizing Phase is to ensure that the migrated solution meets the business needs. UAT is performed on a collection of business functions in a production environment after the completion of functional testing. This is the final stage in the testing process before the system is accepted for operational use. It involves testing the system with data supplied by the actual user or customer instead of the simulated data developed as part of the testing process. UAT helps to validate the solution for the overall user requirements and also determines the release readiness status of the system. Running a pilot for a select set of users helps to identify areas where users have trouble understanding, learning, and using the solution.

For migration projects, UAT involves testing the migrated application and identifying its defects. These defects are addressed and regression testing is conducted for each fixed defect to ensure that the fix doesn't break any other functionality of the migrated application. The UAT Summary confirms that the solution meets the customer's acceptance criteria, thereby assisting in customer acceptance of the solution.

Regression Testing

Regression testing refers to retesting previously tested components and functionality of the system to ensure that they function properly even after a change has been made to parts of the system. For migration projects, this is the most important class of tests. As defects are discovered in a component, modifications should be made to correct them. This may require retesting of other components or the entire solution.

Regression testing helps in the following areas:

- To ensure that no new problems are introduced and that the operational performance has not been degraded because of modifications.
- To ensure that the effects of the changes are transparent to other areas of the application and other components that interact with the application.
- To modify the original test data and test cases from other testing activities.

Resolving the Solution Defects

In order to resolve defects, they must be reproduced and tested in the test environment. Each reproduced defect in the test environment should be tracked with its status and severity. An important aspect of such tests involves test tracking and test reporting. Test tracking and reporting occurs at frequent intervals during the Developing and Stabilizing Phases. During the Stabilizing Phase, this reporting is driven by the bug count. Regular communication of the test status to the team and other key stakeholders ensures that the project runs smoothly. After fixing the defects, test cases and test data should be updated and integrated with the test suite.

Bug Convergence

Bug convergence is the point at which the team makes visible progress against the active bug count. At bug convergence, the rate of bugs resolved exceeds the rate of bugs found, thus the actual number of active bugs decreases. After bug convergence, the number of bugs should continue to decrease until the zero bug bounce task, as explained in the next sections.

Interim Milestone: Bug Convergence

Bug convergence tells the team that most of the bugs are addressed and the rate of bugs resolved is higher than the rate of new bugs found. This can be considered as the interim milestone and the migrated application can be considered for zero bug bounce verification.

Zero Bug Bounce

Zero bug bounce is the point in the project when development finally catches up to testing and no active bugs currently exist. After zero bug bounce, the number of bugs should continue to decrease until the product is sufficiently stable for the team to build the first release candidate.

Interim Milestone: Zero Bug Bounce

Achieving zero bug bounce is a clear sign that the solution is near to being considered a stable release candidate.

Release Candidates

After the first achievement of zero bug bounce, a series of release candidates is prepared for release to the pilot group. Each release is marked as an interim milestone.

Guidelines for declaring a build as a release candidate include the following:

- Each release candidate has all the required elements to qualify for release to production.
- The test period that follows determines whether a release candidate is ready to release to production or if the team must generate a new release candidate with appropriate fixes.
- Testing the release candidates, performed internally by the team, requires highly focused, intensive efforts and concentrates heavily on discovering critical bugs.

Interim Milestone: Release Candidate

As each new release candidate is built, there should be fewer bugs reported, classified, and resolved. Each release candidate marks significant progress in the team's approach toward deployment. With each new candidate, the team must focus on maintaining tight control on quality.

Interim Milestone: Preproduction Test Complete

Eventually, a release candidate is prepared that contains no defects. When this has occurred, no defects should be found within the isolated staging environment. At this stage, all testing that can be done before putting the migrated component into production has been completed.

Conducting the Solution Pilot

This section describes the best practices to adopt for conducting a pilot of the migrated application. This section provides you with information regarding various points to be considered while conducting a pilot and deciding the next steps after the pilot.

A pilot release is a deployment into a subset of the live production environment or user group. During the pilot, the team tests as much of the entire solution as possible in a true production environment. Depending on the context of the project, the pilot can take various forms:

- In an enterprise, a pilot can be a group of users or a set of servers in a data center.
- For migration projects, the pilot might involve testing the most demanding application or database that is being migrated with a sophisticated group of users who can provide helpful feedback.

The common element in all piloting scenarios is testing under live conditions. The pilot is not complete until the team ensures that the solution is viable in the production environment and that the solution is ready for deployment.

Some of the best practices that should be followed while conducting a pilot are:

- Before beginning a pilot, the team and the pilot participants must clearly identify and agree upon the success criteria for the pilot. These should map back to the success criteria for the development effort.
- Any issues identified during a pilot must be resolved either by further development, by documenting resolutions and workarounds for the installation team and production support staff, or by incorporating them as supplemental material in training or Help documentation.
- Before the pilot is started, a support structure and an issue-resolution process must be in place. This may require that the support staff receive training in the application area that is being piloted.
- In order to determine any issues and confirm that the deployment process will work, it is necessary to implement a trial run or a rehearsal of all the elements of the deployment prior to the actual deployment.

After you collect and evaluate the pilot data, a corresponding strategy should be selected based on the findings from the analysis of pilot data. The next strategy could be one of the following:

- **Stagger forward.** Deploy a new release to the pilot group.
- **Roll back.** Execute the rollback plan and revert the pilot group to the stable state they had before the pilot started.
- **Suspend.** Suspend the entire pilot.
- **Fix and continue.** If you find an issue during the pilot, fix the issue and continue with the next steps.
- **Proceed.** Advance to the Deploying Phase.

After the pilot has been completed, the pilot team must prepare a report detailing each lesson learned and how new information was incorporated and issues were resolved.

Interim Milestone: Pilot Complete

This milestone signifies that the pilot has been successfully completed and that the team is ready to proceed to the Deploying Phase.

Closing the Stabilizing Phase: Release Readiness Approved

The Stabilizing Phase culminates with the Release Readiness Approved Milestone. The team builds a release candidate (with all the major defects fixed) that satisfies the necessary quality policy of the organization. All rounds of testing must be completed, meaning that all test plans have been executed and test cases satisfied before the migrated component can be moved into the production environment. Then the release is approved with a formal sign-off marking that the Release Readiness Approved Milestone has been reached.

Key stakeholders, typically representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the solution is complete and approved for release. The sign-off document becomes a project deliverable and is archived for future reference.

The performance of the application following deployment in the production environment is a key criterion in indicating a successful application migration. The following sections will help you to optimize the performance of the application and the tools following deployment.

Tuning

This section discusses tuning of the solution in detail, including how to performance-tune the migrated application, and scaling up and scaling out of the application. In addition, the section discusses multiprocessor considerations for applications and network utilizations. You can use this information to identify the parameters that affect application performances and steps to consider in the scalability of applications.

Performance Tuning

Performance management starts with the gathering of a data baseline that indicates what system performance should look like. After establishing a baseline, it is used to evaluate the performance of the application. Performance problems typically do not become apparent until the application is placed under an increased load.

Measuring the performance of an application when placed under ever increasing loads determines the scalability of that application. When the performance begins to fall below the stated minimum performance requirements, you have reached the limit of scalability of the application. For more information about scaling, refer to the "Scaling Up and Scaling Out" section later in this chapter.

Performance tuning can be done in the following ways:

- Tuning the computer hardware by adding more memory, updating CPUs, adding disk controllers, or upgrading network controllers. This is the most efficient way and helps performance-tune the application as well.
- Application rearchitecture to remove bottlenecks such as poor threading and looping and checking for other loops that use too much CPU time. This step also helps considerably in performance tuning.
- Operating system parameter tuning, which involves adjusting the amount of page store and tweaking network stack parameters.
- Tuning the configurations on a database server, application server, or Web server.

In UNIX, performance is monitored using a type of kernel-level instrumentation, along with rudimentary tools for monitoring the CPU, disk, and memory usage. Windows Server™ 2003 is designed such that it exposes a great deal of performance data. Tools like Windows Performance Monitor (PerfMon) can be used to export detailed information about the processor, memory, disk, and network usage. Performance Monitor support is integrated throughout Windows. Administrators can gather a variety of performance data from many computers simultaneously.

UNIX kernels tend to have many configurable parameters that can be fine-tuned for specific applications. By contrast, the Windows kernel is largely self-tuned. The virtual memory, thread scheduling, and I/O subsystems all dynamically adjust their resource usage and priority to maximize throughput. The difference between these two approaches is that the UNIX approach is to tweak kernel parameters for maximum advantage in the benchmark, even if those tweaks affect the real-world performance, whereas the Windows approach is to let the kernel tune itself for whatever load is placed on it.

Notes

More information on improving performance is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/fastmanagedcode.asp.

More information on writing high-performance managed applications is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/highperfmanagedapps.asp.

Scaling Up and Scaling Out

Scalability is a measure of how easy it is to modify the application infrastructure and architecture to meet variances in utilization. As with other application capabilities, the decisions you make during the design and early coding phases largely dictate the scalability of your application.

Application scalability requires a balanced partnership between two distinct domains: software and hardware. Because scalability is not a design concern of stand-alone applications, the applications discussed here are distributed applications.

Scaling up involves achieving scalability with the use of better, faster, and more expensive hardware to move the processing capacity limit from one part of the computer to another. Scaling up includes adding more memory, adding more or faster processors, or just migrating the application to a more powerful, single computer. Typically, this method allows for an increase in capacity without requiring changes to source code. However, adding CPUs does not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added.

Scaling out distributes the processing load across more than one server by dedicating several computers to a common task. In this, the fault tolerance of the application is increased. Scaling out also presents a greater management challenge because of the increased number of computers.

Developers and administrators use a variety of load-balancing techniques to scale out with the Windows platform. Load balancing allows an application site to scale out across a cluster of servers, making it easy to add capacity by adding replicated servers. It provides redundancy, giving the site failover capabilities so that it remains available to users even if one or more servers fail or are taken down.

Scaling out provides a method of scalability that is not hampered by hardware limitations. Each additional server provides a near linear increase in scalability.

The key to successfully scaling out an application is location transparency. If any of the application code depends on knowing which server is running the code, location transparency has not been achieved and scaling out will be difficult. This situation requires code changes to scale out an application from one server to many, which is seldom an economical option. If you design the application with location transparency in mind, scaling out becomes an easier task.

Notes

More information on scaling is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconmanageabilityoverview.asp.

Microsoft Application Center 2000 reduces the complexity and the cost of scaling out. More information on "Application Center 2000" is available at

http://www.microsoft.com/applicationcenter/default.mspx.

More information on scaling network-aware applications is available at http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/1000/Winsock/toc.asp.

Multiprocessor Considerations

Application performance improves by having multiple processors perform the same task. You can distribute the processing load across several processors.

Computationally intensive tasks are characterized by intensive processor usage with relatively few I/O operations. The ongoing challenge with these applications is to improve the performance. You can do this with a faster computer, a more efficient algorithm, and by improving the implementation or using more processors. You can improve the performance with the help of tuning techniques as well.

Using more processors can mean taking advantage of an SMP computer or by using distributed computing with multiple networked computers. However, adding CPUs does not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added. For computers with SMP configurations, each additional processor incurs system overhead. After you have upgraded each hardware component to its maximum capacity, you will eventually reach the real limit of the processing capacity of the computer. At that point, the next step is to move to another computer.

Multiprocessor optimization can be achieved by making use of threads.

Note More information on multiprocessor optimizations is available at

http://msdn.microsoft.com/msdnmag/issues/01/08/Concur/.

Network Utilizations

Network resources, such as available bandwidth and latency, must be predicted and managed on computers and devices throughout the network.

Optimal network utilization is achieved with cooperation among end nodes, switches, routers, and wide area network (WAN) links through which data must pass. Preferential treatment must be given for certain data as it traverses through the network in order to better service certain components during congestion. Tools are available that help analyze network traffic, provide network statistics and packet information, and thereby better use the network by analyzing areas of congestion.

Quality of Service (QoS), an industry-wide initiative, achieves a more efficient use of network resources by differentiating between data subsets. Windows 2000 implements QoS by including a number of components that can cooperate with one another.

Note More information on QOS on Windows is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/qos/qos/qos_start_page.asp.

Note Network Monitor captures network traffic for display and analysis. More information on Network Monitor is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netmon/netmon/network_monitor.asp.

Note Network Probe is another tool for traffic-level network monitoring and for analysis and visualization. More information on Network Probe is available at http://www.objectplanet.com/probe/.

Testing and Optimization Tools

This section lists some of the useful tools that can be used for testing and monitoring your applications.

Visual Studio .NET 2003 Tools

Microsoft Visual Studio® .NET 2003 includes tools for analyzing the performance of applications. These include:

- **Process Viewer (Pview)**. The PView process viewer uses dialog boxes to view and modify running processes and their threads. PView can monitor:
  - Memory usage of process, threads, and individual DLLs.
  - CPU time used by processes and threads.
  - How an application or the system runs with different system priorities.

  PView features provide powerful tools with which you can monitor processes of an application and threads at different priorities. More information about the Process Viewer is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample98/html/vcsmppviewer.asp.
- **Spy++.** Spy++ shows a graphical view of the processes of the system, threads, windows, and window messages. More information about Spy++ is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_overview.3a_.spy.2b2b.asp.
- **DDESpy.** DDESpy monitors dynamic data exchange (DDE) activity in the operating system. More information about DDESpy is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample98/html/vcsmppviewer.asp.
- **Visual Studio Team System (VSTS).** Microsoft Visual Studio .NET 2005 provides Visual Studio Team System, which supports a huge suite of integrated tools for code development, enforcement of good coding practices and conventions, static code analysis tools, integrated test tools, and code coverage tools. More information about VSTS is available at http://msdn.microsoft.com/vstudio/teamsystem/default.aspx.
- **FxCop**: FxCop is a tool that enables development teams to check code compliance with best practices. More information about FxCop is available at http://msdn.microsoft.com/netframework/programming/classlibraries/fxcop/.

Platform SDK Tools

Platform SDK tools includes debugging tools, file management tools, performance tools, and testing tools. These tools are available with the latest Platform SDK.

Debugging Tools

Platform SDK includes the following debugging tools:

- **Debug Monitor (DBMon)**. Debug Monitor runs in its own console window and displays messages sent by your application. More information about Debug Monitor is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/debug_monitor.asp.
- **Symbolic Debugger (NTSD)**. NTSD is a symbolic debugger that enables you to debug user-mode applications. You can display and execute program code, set breakpoints, and examine and change values in memory. More information about Symbolic Debugger is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/symbolic_debuggers.asp.
- **Windows Debugger (WinDbg)**. The WinDbg debugger is a powerful graphical tool that allows you to debug applications on Microsoft Windows. You can use the integrated text editor to edit your source code. WinDbg can also be used to debug service applications and kernel-mode drivers. More information about Windows Debugger is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/windbg_debugger.asp.

File Management Tools

- **WinDiff**. WinDiff is used to compare files and display the results graphically. More information about WinDiff is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/windiff.asp.

Performance Tools

Performance tools can be used to measure application performance and resolve some performance issues. Platform SDK includes the following performance tools:

- **Bind**. Bind minimizes application load time by binding your executable with all of your DLLs, plus the system DLLs. More information about Bind is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/bind.asp.
- **Extensible Performance Counter List (ExCtrLst)**. The extensible counter list tool is used to obtain information about the extensible performance counter dynamic-link libraries on a computer. More information about ExCtrLst is available at http://www.microsoft.com/downloads/details.aspx?FamilyID=7ff99683-b7ec-4da6-92ab-793193604ba4&DisplayLang=en.
- **Performance Meter (PerfMtr)**. PerfMtr can display a variety of system performance information. More information about PerfMtr is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/perfmtr.asp.
- **Performance Monitor (PerfMon)**. Windows Performance Monitor can simultaneously collect performance data from any number of network computers, then display it as a graph, format it as a tabular report, or log it for later analysis. Performance Monitor support is integrated throughout Windows. More information about PerfMon is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/perfmtr.asp.
- **PStat**. PStat lists statistics for each process. More information about PStat is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/pstat.asp.
- **Virtual Address Dump (VADump)**. Virtual Address Dump creates a listing that contains information about the memory usage of a specified process. More information about VADump is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/vadump.asp.

Testing Tools

- **Process Fault Monitor (PfMon)**. The Process Fault Monitor displays the faults that occur while executing a process. PFMon can start the application for you or attach to a running process. More information about PfMon is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/pfmon.asp.

Other Commonly Used Tools

This section lists other commonly used tools that are useful in testing and monitoring applications.

Monitoring Tools

- **Diskmon**. This tool captures all hard disk activity or acts as a software disk activity light in your system tray. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/diskmon.shtml.
- **Filemon**. This monitoring tool allows you to view all file system activity in real-time. This tool works on all versions of Windows NT, Windows 2000, Windows Server 2003, and Windows XP. It also works with the Windows XP 64-bit edition. This tool is available for download at http://www.sysinternals.com/ntw2k/source/filemon.shtml.
- **PMon**. This is a Windows NT GUI/device driver program that monitors process and thread creation and deletion, as well as context swaps if it is running on a multiprocessing or checked kernel. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/pmon.shtml.
- **Portmon**. You can monitor serial and parallel port activity with this advanced monitoring tool. It knows about all standard serial and parallel IOCTLs and even shows you a portion of the data being sent and received. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/portmon.shtml.
- **Regmon**. This monitoring tool allows you to view all registry activity in real-time. This tool is available for download at http://www.sysinternals.com/ntw2k/source/regmon.shtml.
- **TCPView**. You can view all the open TCP and UDP endpoints. TCPView even displays the name of the process that owns each endpoint. This tool is available for download at http://www.sysinternals.com/ntw2k/source/tcpview.shtml.
- **Task Manager**. Task Manager provides run-time information on processes. The Task Manager tool is available as part of Windows.

Testing Tools

- **WinRunner**. WinRunner helps in GUI capture and playback testing for Windows applications. More information on WinRunner is available at http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/.
- **Silktest**. Silktest is an object-oriented software testing tool for Windows applications. More information on Silktest is available at http://www.segue.com/products/functional-regressional-testing/silktest.asp.
- **LoadRunner**. LoadRunner is an automated client/server system testing tool that provides performance testing, load testing, and system tuning for multiuser applications. More information on LoadRunner is available at http://www.mercury.com/us/products/performance-center/loadrunner/.
- **Rational Robot Automated Test**. Rational Robot Automated Test provides automated functional, regression, and smoke tests for e-applications. More information on Rational Robot is available at http://www-306.ibm.com/software/rational.
- **Microsoft Application Center Test**. Designed to stress test Web servers and analyze performance and scalability problems with Web applications, including Active Server Pages (ASP) and the components they use. It simulates a large group of users by opening multiple connections to the server and rapidly sending HTTP requests. More information on Microsoft Application Center Test is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/act/htm/actml_main.asp.

Source Test Tools

- **Purify**. Purify is a run-time error and memory leak detector. More information on Purify is available at http://www-306.ibm.com/software/sw-bycategory.

Tools for win64:

- **VTune Performance Analyzer**. Intel VTune Performance Analyzer helps locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. More information about VTune Performance Analyzer is available at http://www.intel.com/software/products/vtune/.

Further Reading

- More information on testing software patterns is available at http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpag/html/tsp.asp.
- Tools are also available in .NET for creating components, which can be used to monitor system resources. More information on using event logs, performance counters, and services is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vborisystemmonitoringwalkthroughs.asp.

Index