Microsoft

Team Development with Visual Studio® .NET and Visual SourceSafe™

patterns & practices

Version 1.0

Contents

Introduction

This guide provides guidance and recommendations to enable you to set up a team development environment and work successfully within it.

If you are beginning a .NET team development project, you first need to understand how to establish development processes that work in a team environment. You need to know how to set up and work with the team development features supported by the Microsoft® Visual Studio® .NET integrated development environment (IDE), and you also need to be aware of the development techniques (such as how to set assembly references in the correct way) that must be followed by your development team members to ensure successful team working.

The guide is divided into the following chapters:

- Chapter 1, "Introducing the Team Environment." This chapter presents an overview of the team environment and introduces the key building blocks and processes. Read this chapter to gauge the scope of the guide and to understand the team development model upon which the document is based.
- Chapter 2, "ASP.NET Web Application Development Models." This chapter describes the approach you should adopt to build Web applications within a team development environment.
- Chapter 3, "Structuring Solutions and Projects." This chapter explains how you should organize and structure Visual Studio .NET solutions and projects and presents the tradeoffs associated with single-solution and multi-solution development models. It also recommends folder structures that you can use to store projects locally and within Microsoft Visual SourceSafe™ (VSS).
- Chapter 4, "Managing Dependencies." This chapter explains how you should handle assembly references, Web references, database references, and COM object references.
- Chapter 5, "The Build Process." This chapter covers the build process and the role of the build server and automated build script that is used to generate system builds.
- Chapter 6, "Working with Visual SourceSafe™." This chapter provides a series of step by step procedures that walk you through common development tasks such as how to add solutions and projects to Visual SourceSafe, how to retrieve solutions from VSS, and how to check files in and out on a daily basis. This chapter gets you up to speed quickly on the essential tasks.
- Chapter 7, "Setting Up and Maintaining the Team Environment." This chapter describes the infrastructure and the hardware and software requirements for all of the workstations and servers within the team environment. It also provides guidance on how to create and maintain a VSS database.

To get the most from this guide, you are encouraged to read all chapters in order.

Who Should Read This Guide

This guide provides team-development guidelines for lead developers, developers, test team members, and system administrators. Read this guide if you plan to, or currently work on, a team-based .NET development project.

What You Must Know

To use this guide to establish a team development environment and a development process suited to .NET, you must have some development experience with Visual Studio .NET. This guide assumes you have either created or are familiar with .NET assemblies and Web services.

You should also be aware of the general issues and challenges traditionally associated with team-based software development projects. Ideally, you will have some experience with a source control system, preferably VSS.

Note: This guide focuses on the use of VSS version 6.0c (the version that ships with Visual Studio .NET) as the source control system. However, much of the guidance and many of the discussed processes apply equally well to other change management systems, many of which can be directly integrated into the Visual Studio .NET IDE.

Terminology

Terms such as "system," "solution," and "project," which are used extensively in this guide, tend to be heavily overloaded. The following sections describe the context in which this guide uses these terms.

System

The term "system" refers to the overall application that you are developing. Your system ultimately consists of the combined set of assemblies that you release into a production environment.

Inner and Outer System Boundaries

This guide also introduces the idea of an inner and outer system boundary. This division becomes important when you start to consider which components of your system are built in-house by your central build process and which components fall outside of the domain of the build process and are simply referenced as external dependencies. The following describes the system boundaries:

- Inner system assemblies are built as part of your system build process.
- Outer system assemblies are all other assemblies, including third-party components and .NET Framework assemblies.

Figure 1 illustrates the idea of the inner and outer system boundaries.

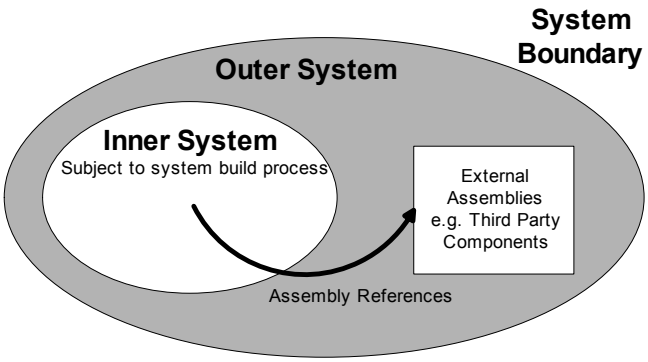


Figure 1
Inner and Outer System Boundaries

Solution

If you are new to Visual Studio .NET, the term "solution" will also be new. A solution essentially represents everything you are currently working on. Visual Studio .NET uses solutions as containers for individual projects—these generate your system components (.NET assemblies). Solution files maintain project dependency information and are primarily used to control the build process. Solutions are discussed further in Visual Studio .NET Solutions in Chapter 3, "Structuring Solutions and Projects."

Project

In the context of this guide, there are three types of projects:

- **General development projects.** The term "project" in its loosest sense refers to your team's current development effort.
- **Visual Studio .NET projects.** Project files are used by Visual Studio .NET as containers for configuration settings that relate to the generation of individual assemblies. These are discussed further in Visual Studio .NET Projects in Chapter 3, "Structuring Solutions and Projects."
- **Visual SourceSafe Projects.** A project in a VSS database is simply a collection of (usually logically related) files. A VSS project is similar to an operating system folder, with added version control support.

1

Introducing the Team Environment

There are many elements, processes, and roles that combine to enable successful team-based software development projects. This document concentrates on two of the core processes:

- The development process
- The build process

Although these are separate processes, they share much in common and as a result, it's essential to develop working practices and project structures that work well for both scenarios.

The team development environment is illustrated in Figure 1.1. The shaded region of the diagram illustrates the areas that this document addresses. Study the diagram carefully because it defines the working model for the remaining chapters of the document.

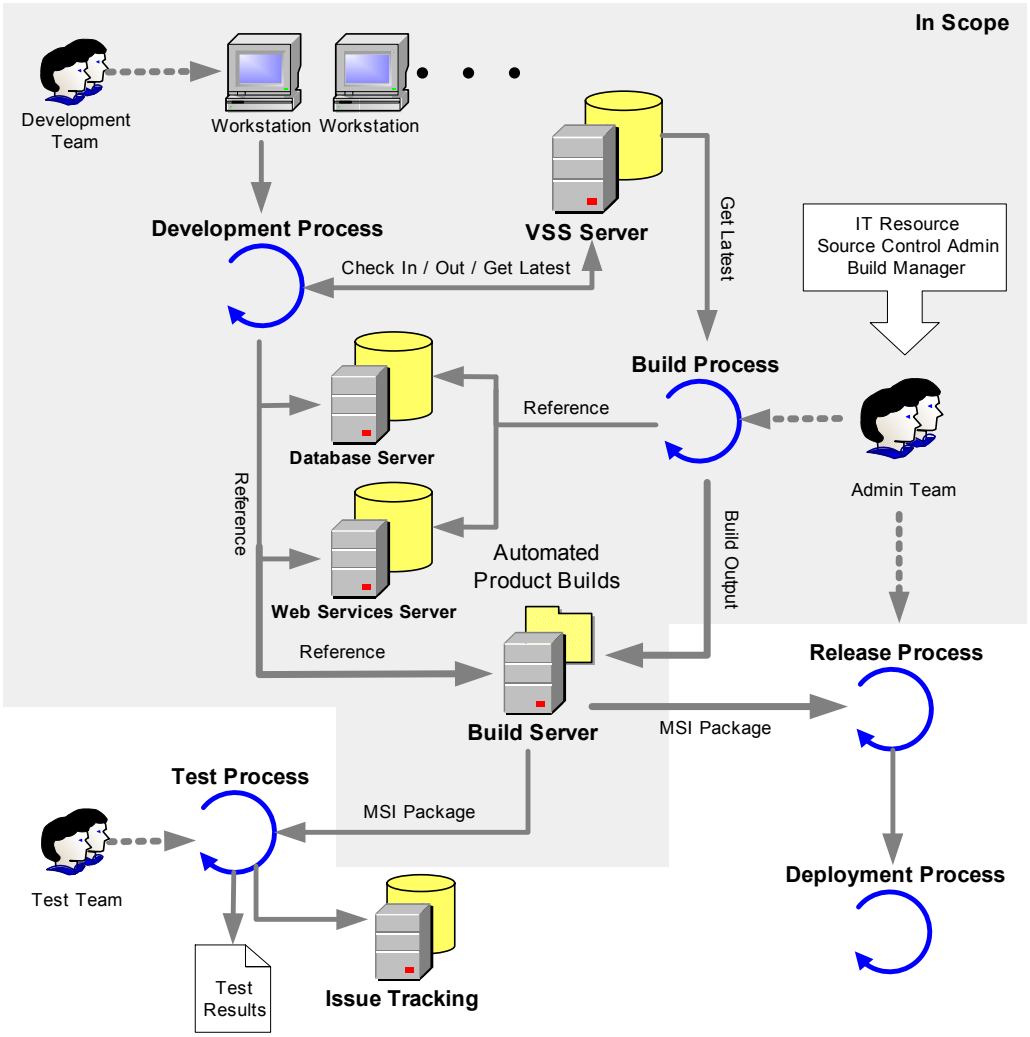


Figure 1.1
The Team Development Environment

Team Development Servers and Workstations

The roles and responsibilities of the key servers and workstations illustrated in Figure 1.1 are defined in the following sections. Other essential servers, such as the backup server, are omitted from the diagram for the sake of clarity. For detailed information about the infrastructure within the team environment that includes hardware and software requirements, see Chapter 7, "Setting Up and Maintaining the Team Environment."

The VSS Server

This is a central server that hosts one or more Microsoft® Visual SourceSafe™ (VSS) databases used to provide versioned controlled access to project source files. As a developer, you interact with it on a daily basis as you check project files in and out through the Microsoft Visual Studio® .NET integrated development environment (IDE). It is also accessed by the build script to obtain the latest source code required for the current system build.

More Information

For information about how VSS projects should be structured, see Use a Consistent Folder Structure for Solutions and Projects in Chapter 3, "Structuring Solutions and Projects."

For more information about how to configure the VSS server, see Installing and Administering VSS in Chapter 7, "Setting up and Maintaining the Team Environment."

The Build Server

An automated build script that is used to compile and build your entire system runs on this server. The build script is a critical element for all software development projects. It allows you to generate successive versions of your system in an automated and consistent, repeatable fashion.

The output assemblies generated by the build process are maintained in folders on this server.

More Information

For more information about referencing external assemblies, see Referencing Assemblies in Chapter 4, "Managing Dependencies."

For more information about the build process, see Chapter 5, "The Build Process."

Development Workstations

All workstations should be configured in a similar way. This includes the installation and configuration of the Visual Studio .NET IDE. Enterprise Templates can help with this.

More Information

For more information about the benefits of Enterprise Templates, see Visual Studio Enterprise Templates in Chapter 7, "Setting Up and Maintaining the Team Environment."

Database Servers

These servers host instances of Microsoft SQL Server™ and provide a central location to which developers can connect to databases whose schemas match the current system database design. In some scenarios, you also need local SQL Server databases on development workstations to perform isolated unit testing. For example, local servers allow you to govern the current set of test data, and when you manipulate this data, you do not impact other team members.

More Information

For more information about working with databases in the team environment, see Database Development in Chapter 4, "Managing Dependencies."

For more information about how to most flexibly manage connection strings in a team environment, see Referencing Databases in Chapter 4, "Managing Dependencies."

Web Services Server

The primary function of the Web services server in the team development environment is to host Extensible Markup Language (XML) Web services that are currently under development. While the development teams responsible for Web services develop them on their local workstations using local instances of Microsoft Internet Information Server (IIS), the central Web server allows the services to be published for other developers or development teams to reference from client projects.

More Information

For more information about working with Web services, see Referencing Web Services in Chapter 4, "Managing Dependencies."

2

ASP.NET Web Application Development Models

This chapter describes how you should approach Web application development in a team environment. It recommends an isolated model for Web development but contrasts this with alternate approaches.

There are three main models for developing Web applications:

- Isolated (recommended)
- Semi-Isolated
- Non-Isolated

Isolated

With this model you develop (edit, debug and run) in complete isolation on your own development workstation using your local Web server (http://localhost). Access to the master source files is controlled via a Microsoft® Visual SourceSafe™ (VSS) database located on a network file share. You may or may not choose to allow developers to check out the same file simultaneously. For more information, see Multiple Checkout in Chapter 6, "Working with Visual SourceSafe."

Semi-Isolated

With this model, you use a common Web server (http://remoteserver) for application development and debugging. You check files in and out via a VSS database located on a network file share. Your working copy of the project is located on the common Web server in a specified project folder, which is also a Microsoft Internet Information Server (IIS) virtual root. Each developer has a unique project folder on the common Web server.

Note: When you obtain a Web project from VSS for the first time, Microsoft Visual Studio® .NET does not allow you to place working files in a folder that already contains another Web project.

Developers can check out and edit the same file simultaneously if the VSS multiple checkout feature is enabled, but only one developer can debug the application on the Web server at any one time. This is because when you debug an application, IIS is blocked. This prevents other Web requests for any other application being serviced.

Non-Isolated

With this model, you again use a common Web server (http://remoteserver) for application development and debugging. However, you do not have your own working copy of files and all developers use a single folder and virtual root on the server; for example, http://remoteserver/projectname.

When you save changes to a file, the in-memory version of the file on your workstation is transferred to the server using Hypertext Transfer Protocol (HTTP). This overwrites the existing copy on the server. When you subsequently check in your changes by using the Visual Studio .NET integrated source control services, Microsoft FrontPage® Extensions are used to update the master copy of the file within a VSS database.

The three models are illustrated in Figure 2.1.

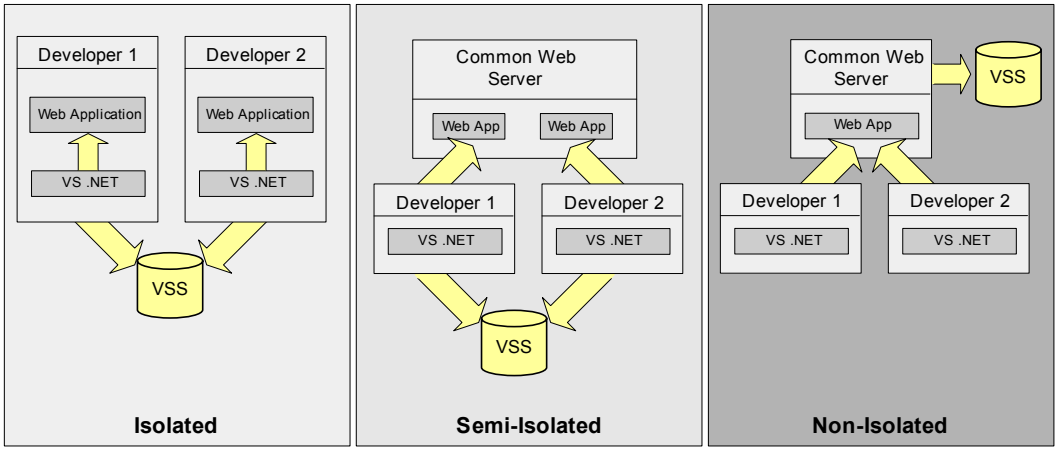


Figure 2.1
Web Development Models

Use an Isolated Development Model

You are strongly advised to adopt the isolated development model for team developments because it offers a number of significant advantages.

Advantages of Isolated Development

Adopting an isolated development model provides the following advantages:

- You and your fellow team members develop independently of one another using separate (local) instances of the Web application.
- You can both develop and debug the application without inadvertently interfering with one another.
- It provides superior support for source-code control (compared to the non-isolated model that uses FrontPage Extensions).
- It is slightly faster in a local area network (LAN) environment (compared to FrontPage Extensions).

Avoid Semi-Isolated and Non-Isolated Development Models

It is difficult to use semi-isolated and non-isolated models in a team development environment. These should be avoided whenever possible.

Disadvantages of Semi-Isolated and Non-Isolated Models

Semi-isolated and non-isolated models suffer from the following disadvantages:

- It is very easy to inadvertently affect another developer. For example, when you debug an application, the debug process locks the common Web server and impacts other team members.
- In a non-isolated model, developers can also impact each other as there is only a single code behind dynamic-link library (DLL) per Web application.
- FrontPage Extensions (with no VSS integration) offer only limited source control capabilities. With the non-isolated development model, all developers work with a master copy located on the Web server. The source control functionality of FrontPage Extensions offers a "last check-in wins" development model. If users A and B both check out the same file in close succession, user A makes changes and then saves them, and then user B saves changes, user A's changes are lost.

The one occasion where you might be forced to adopt a semi-isolated or non-isolated model is when specific resources required by your Web application are available only on the common Web server. You may encounter this situation if you develop with Microsoft .NET Passport.

If you are forced to use FrontPage Extensions, you can configure Visual Studio .NET to use this mode of operation for all new Web projects and you can change the mode of an existing Web project.

► **To configure Visual Studio .NET to use FrontPage Extensions**

1. On the **Tools** menu, click **Options**.
2. Click the Projects folder.
3. In the Projects folder, click **Web Settings**.
4. In the right pane, select the **FrontPage Extensions** option.
5. Click **OK** to accept the changes.

► **To change the access mode of an existing Web project**

1. Right-click the project within Solution Explorer, and then click **Properties**.
2. Expand the Common Properties folder, and then click **Web Settings**.
3. Change the **Web Access Mode** setting.
4. Click **OK** to accept the changes.

More Information

For more information about developing source controlled Web projects in Visual Studio.NET, see Web Projects and Source Control Integration in Visual Studio.NET at *http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vetchWebProjectsSourceControlIntegrationInVisualStudioNET.asp*.

3

Structuring Solutions and Projects

To ensure that your development and build processes work effectively in a team environment, it's essential to start with a correct project structure that is consistent across all of your development workstations and build server.

This chapter presents guidelines on:

- Partitioning Microsoft® Visual Studio® .NET solutions and projects.
- Managing the local file system and Microsoft Visual SourceSafe™ (VSS) folder structure.
- Applying naming conventions for projects, assemblies and namespaces.

Visual Studio .NET Solutions and Projects

Before discussing how to organize Visual Studio .NET solutions and projects, it's very important that you understand their basic mechanics and how they are managed locally and by your source control provider—typically VSS.

If you are already familiar with Visual Studio .NET solutions and projects and you understand the various file types comprising a project, you can skip this section and jump to Always Use Visual Studio .NET for Source Control Operations.

Visual Studio .NET Projects

Visual Studio .NET uses project files as containers for all of the build and configuration settings required to generate a .NET assembly. Project files have either a .csproj or .vbproj file extension, depending upon the project's language. There are many different project types [and associated Rapid Application Development (RAD) templates] although these can be divided broadly into two categories. The two categories of project types are:

- **Web projects.** A Web project is one that is created with a Hypertext Transfer Protocol (HTTP) location (for example, http://localhost/MyWebProject). Web projects include ASP.NET Web Applications used to deliver content to Web browsers and ASP.NET Web Services used primarily for data integration over the Internet.
- **Non-Web or local projects.** Non-Web or local projects are created with a file system location (for example, C:\Projects\MySystem\MySolution\MyWinProject).

  The most common local project types are Windows applications and class libraries, although there are many others, including services, console applications, database projects, and so on.

Visual Studio .NET Solutions

Solution files (with the .sln file extension) are used to group related projects together and are primarily used to control the build process. You can use solutions to control build dependency issues and control the precise order in which contained projects are built.

Important: A project can be part of one or more solutions but solutions can't be included within other solutions.

Figure 3.1 illustrates the relationship between projects and solutions and indicates the file types used by VSS to maintain solution and project level settings:

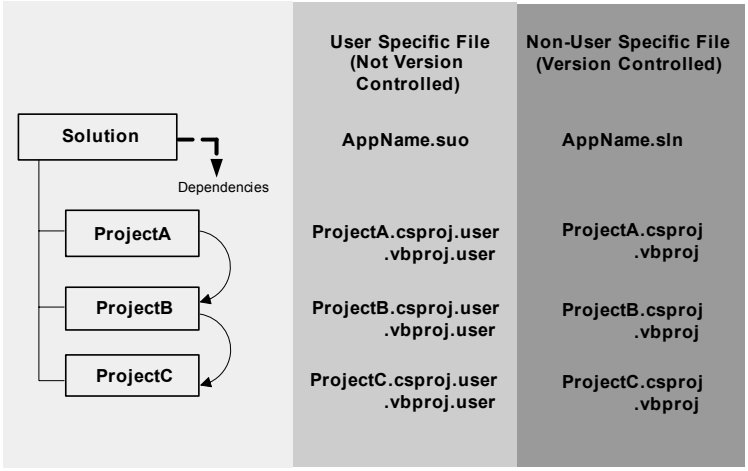


Figure 3.1
Visual Studio .NET Projects and Solutions

Solutions and Build Dependencies

The solution file also contains project dependency information used by the build process. For example, in the preceding diagram, the dependency information indicates that Project A depends upon Project B and Project B depends upon Project C. As a result, the build order must be Project C, then Project B, and then Project A. When project references are used within a single solution, Visual Studio .NET ensures the correct build order.

Important: There are two basic types of references—project references and file references. You set both types by using the **Add References** dialog box in Visual Studio .NET. Because project references also establish build order dependencies, you should use them whenever possible. For more information, see Referencing Assemblies in Chapter 4, "Managing Dependencies."

Files Subject to Source Control

The following list identifies the key file types that are automatically added to VSS when a solution is added to source control by using the Visual Studio .NET integrated development environment (IDE):

- **Solution files (*.sln).** The key items maintained within these files include a list of constituent projects, dependency information, build configuration details, and source control provider details.
- **Project files (*.csproj or *.vbproj).** The key items maintained within these files include assembly build settings, referenced assemblies (by name and path), and a file inventory.
- **Application configuration files.** These are configuration files based on Extensible Markup Language (XML) used to control various aspects of your project's run time behavior.

  Note: For Web applications, the source controlled configuration file is called Web.config. For non-Web applications, the source controlled file is called app.config and is contained within the project folder. At run time, the Visual Studio .NET build system copies App.config to the Bin folder and renames it to Yourappname.exe.config.

  For non-Web applications, a configuration file is not automatically added to a new project. If you require one, add it manually. Make sure you call it App.config and locate it within the project folder.

- **Source files (*.cs, *.vb, *.aspx, *.asax, *.resx, *.vsdisco, *.css, and so on).** All project source files are subject to source control.

Files Not Subject to Source Control

The following files are not added to source control because they are developer specific:

- **Solution user option files (*.suo).** These contain personalized customizations made to the IDE by an individual developer.
- **Project user option files (*.csproj.user or *.vbproj.user).** These contain developer specific project options and an optional reference path that is used by the IDE to locate referenced assemblies. The section Referencing Assemblies in Chapter 4, "Managing Dependencies," explains how assembly references should be managed in a team environment.
- **WebInfo files (*.csproj.webinfo or *.vbproj.webinfo).** This file keeps track of a project's virtual root location. This is not added to source control to allow individual developers to specify different virtual roots for their own working copy of the project. While this capability exists, you and all team members are recommended to use a consistent (local) virtual root location when you develop Web applications. The recommended structure for Web and non-Web applications is discussed in Use a Consistent Folder Structure for Solutions and Projects.
- Build outputs that include assembly dynamic-link libraries (DLLs), Interop assembly DLLs and executable files. However, note that you are advised to add outer-system assemblies that are not built as part of your system build process (such as third-party controls and libraries) to VSS within the projects that reference them. For details, see Include Outer System Assemblies within Projects in Chapter 4, "Managing Dependencies."

Always Use Visual Studio .NET for Source Control Operations

All project creation and manipulation within VSS should be performed by using the integrated VSS support menus within Visual Studio .NET—don't use VSS Explorer for this. The Visual Studio .NET functionality guarantees that:

- Only the appropriate files are added to source control.
- Your Visual Studio .NET project and solution files are updated with appropriate VSS specific details. For example, the VSS functionality within Visual Studio .NET updates solution (.sln) files with (among other items):
  1. A count of projects in the solution that are under source-control (this count includes the solution itself)
  2. The VSS server for each project
  3. The location on the server for each project
  4. The name of each project's source-control provider
  5. Each project's location relative to the solution file

Other files including the solution user file (.suo) and project file (.csproj or .vbproj) are also updated.

Important: Always interact with VSS through the Visual Studio .NET interface instead of through the VSS Explorer. The tight integration of the products ensures that files are managed correctly in a team environment.

Partitioning Solutions and Projects

The way you partition solutions and projects has a big impact on the way your development efforts and build process function in a team environment.

There are three main models to consider for partitioning solutions and projects. In order of preference, these are:

1. Single solution
2. Partitioned single solution
3. Multi-solution

Important: Unless you have very good reasons to use a multi-solution model, you should avoid this and adopt either a single solution model, or in larger systems, a partitioned single solution model. These are simpler to work with and offer a number of significant advantages over the multi-solution model, which are discussed in the following sections.

Use a Single Solution Model Whenever Possible

With the single solution model, you create a single Visual Studio .NET solution and use it as a container for all of the projects defined by your application. Note the following when you use a single solution model:

- If one project needs to reference an assembly generated by another, use project references.
- File references should be used only to reference outer-system assemblies (such as .NET Framework assemblies and third-party assemblies) that are not built with the rest of your system.

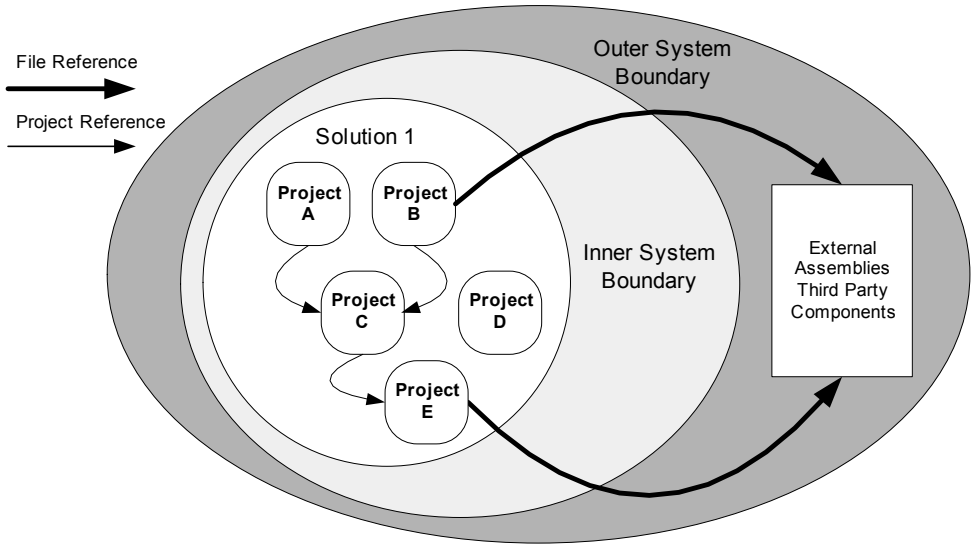


Figure 3.2
The Single Solution Model

Use a single solution model whenever possible because it offers a number of significant advantages.

Advantages

The single solution model offers the following advantages:

- When you need to reference another assembly generated by a separate project, you can use a project reference. Project references are the preferred way to establish references to other assemblies and they are guaranteed to work on all development workstations in a team environment. The many advantages of project references and guidance on when to use file references are discussed in Referencing Assemblies in Chapter 4, "Managing Dependencies."
- Assembly versioning issues are avoided, because Visual Studio .NET detects when a client of a referenced assembly needs to be rebuilt.
- Project references are sensitive to changes in the configuration of the referenced project. This means that you can automatically switch from Debug and Release builds across projects without having to reset references.
- The system build process and build script is much simpler.

Disadvantages

You are advised to adopt the single solution model whenever possible. However:

- The model scales only so far. If you want to work on a single project within the solution, you are forced to acquire all of the source code for all projects within the solution.
- Even minor (nonbreaking) changes to a single source file within a single project can result in a rebuild of many projects within the solution, due to project dependencies. If an assembly's interface changes within a referenced project, you want the client project to be rebuilt. However, unnecessary rebuilds can be very time consuming, especially for solutions containing many projects.

Consider a Partitioned Single Solution Model for Larger Systems

For larger systems where you want to reduce the number of projects and source files required on each development workstation, consider grouping related sets of projects together within separate subsolution files.

This allows you and fellow developers to work on separate and smaller subsystems within the inner-system boundary.

Note: A single project file can be included within one or more solution files. Solutions cannot be included within other solutions.

Figure 3.3 illustrates the partitioned single solution model. Notice how separate solution files are used to allow you to work on smaller subsystems contained within the inner system boundary. Also note how this results in projects being contained within more than one solution file. For example, Projects D and H are in a total of three solution files including the master solution.

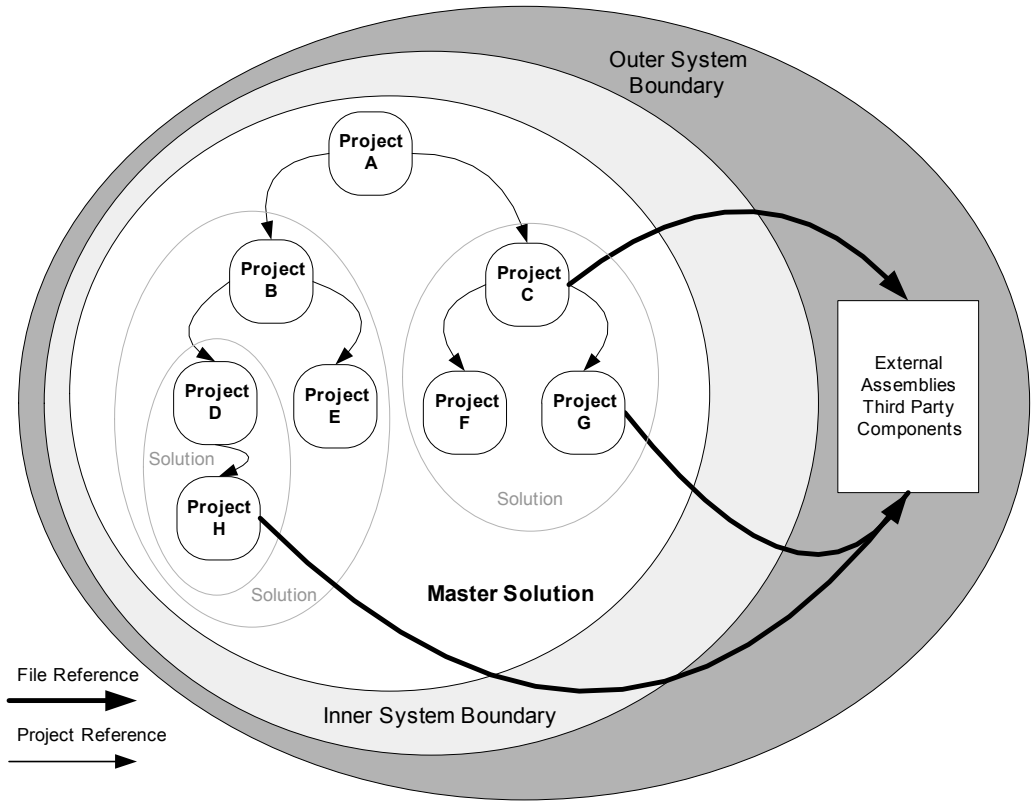


Figure 3.3
The Partitioned Single Solution Model

In the partitioned single solution model:

- All projects are contained within a master solution file. This is used by the system build process to rebuild the entire system. If you need to work on the top level project file, you also work with the master solution.
- Project references are used between individual projects.
- Separate solution files are introduced for selected project files. If you want, you can introduce a solution file for each project within your system. Each solution file contains the main project file, together with any downstream project it depends on and any further projects it depends on, and so on down the dependency chain.
- Separate solution files allow you to work on smaller subsystems within your overall system but retain the key benefits of project references. Within each subsolution file, project references are used between constituent projects.

Note: You should not separate two projects that reference one another into separate solutions, because this would necessitate the use of a file reference which should be avoided wherever possible. For more information, see Referencing Assemblies in Chapter 4, "Managing Dependencies."

Advantages

The partitioned single solution model offers the following advantages:

- You can work on small subsystems. You do not require the source code and project files for the entire system on every development workstation. As a result, the Solution Explorer within Visual Studio .NET remains less cluttered and easier to work with.
- You can use project references within each solution.
- The master solution allows you to easily rebuild the entire system. The master solution file is used by the build process on the build server.

Disadvantages

The partitioned single solution model suffers from the following disadvantages:

- When new projects are added, you must potentially add them and update any project references in multiple solution files; for example, the master solution file and one or more others.
- You are limited in the way you can partition your system. This is driven by project dependency relationships. As a result, if you work on the top level project, for example, an ASP.NET Web application within your presentation tier, you are forced to copy all dependent projects to your development workstation. This is likely to include projects from your business and data tiers. On the other hand, if you work on the development of a class library or data access component with few if any further dependencies, you require only those individual projects.

Use a Multi-Solution Model Only if Absolutely Necessary

The multi-solution model is similar to the partitioned single solution model except:

- There is no master solution file.
- File references are used between projects in separate solutions (although project references are still used between projects within an individual solution).
- The system build script that runs on the build server builds each solution in turn, based on known dependency relationships. The build script places output assemblies in a fixed location on the build server.

The multi-solution model is illustrated in Figure 3.4.

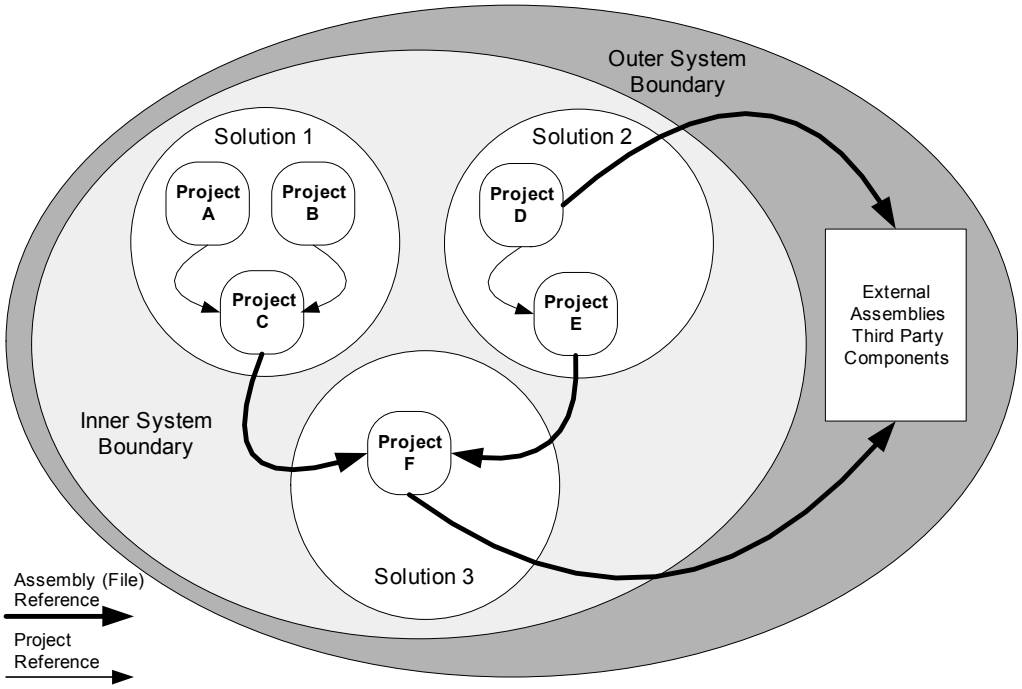


Figure 3.4
The Multi-Solution Model

Advantages

The multi-solution model offers the following advantages over the partitioned single solution model:

- Each project is contained only within a single solution. This means that adding and removing projects to and from your system is easier.
- You are able to subdivide your system into multiple solutions based on logical boundaries and you are not driven by project dependencies. For example, the division that you choose may be based on areas of business functionality.

Disadvantages

The multi-solution model suffers from the following disadvantages:

- You are forced to use file references when you need to reference an assembly generated by a project in a separate solution. These (unlike project references) do not automatically set up build dependencies. This means that you must address the issue of solution build order within the system build script. While this can be managed, it adds extra complexity to the build process.

- You are also forced to reference a specific configuration build of a DLL (for example, the Release or Debug version). Project references automatically manage this and reference the currently active configuration in Visual Studio .NET.
- When you work with single solutions, you can get the latest code (perhaps in other projects) developed by other team members to perform local integration testing. You can confirm that nothing breaks before you check your code back into VSS ready for the next system build. In a multi-solution system this is much harder to do, because you can test your solution against other solutions only by using the results of the previous system build.

You should easily be able to work with a single solution that contains 10, 20, or even 30, projects. The maximum number of workable projects within a solution is difficult to precisely define, because it depends on the specification of your build server and development workstations and the size and number of source files associated with individual projects.

Considerations for Grouping Projects into Solutions

The best way to approach the problem of grouping projects into solutions is to consider the overall architecture of your application. For example:

- Start by identifying the constituent assemblies (or components) within your system; this defines the individual projects that your system requires. Remember that each assembly is generated by a separate Visual Studio .NET project.
- Aim for a single solution model. If you want to break up your projects to provide a greater level of isolation and control, you can use the partitioned single solution model. Think about which groups of projects you want to work on in isolation, for example a set of middle-tier business components, and create separate solutions accordingly.

If you break your projects into multiple solutions and cannot do so using the partitioned single solution model, carefully consider your cross solution dependencies, and the nature of the interfaces that separate the two dependent assemblies. When you organize projects into separate solutions you should:

- Identify the external interfaces that separate the constituent parts of your system. Try to identify those that are least likely to change. If an interface that is exposed by one project changes frequently, any dependent projects should ideally be placed in the same solution.
- If you are using Web services to connect some components of your system together, the Web service interface provides a good dividing line. For example, the projects on the client side of the Web service interface and the projects on the server side are good candidates for separate solutions.

Use a Consistent Folder Structure for Solutions and Projects

Life in a team development environment is a whole lot easier if you use a common structure for storing Visual Studio .NET solutions and projects. To keep things symmetrical (and as a result, simpler), set up a folder structure within VSS that matches your local file system structure.

Define a Common Root Folder

Define a common root folder—for example, C:\Projects on your file system and $/Projects within VSS. This acts as a container for all of your development systems.

Beneath the common root folder, create a system root folder for each of your systems—for example, C:\Projects\MyCompanysInsuranceApp and $/Projects/MyCompanysInsuranceApp respectively.

Adopt a Parent-Child Folder Structure for Solutions and Projects

You should adopt a parent-child folder structure for solutions and projects. To do this:

- Create a subfolder beneath the system root folder for your Visual Studio .NET solution.
- Create additional child subfolders beneath the solution folder for each constituent project.
- Adopt this common structure for Web and non-Web applications
- Do not add the Bin folder or its contents to VSS.

Note: If you use the partitioned single solution model, place project folders beneath the folder that contains the main solution file. For any subsolutions that you create, include project files directly from this location.

Figure 3.5 demonstrates the recommended structure.

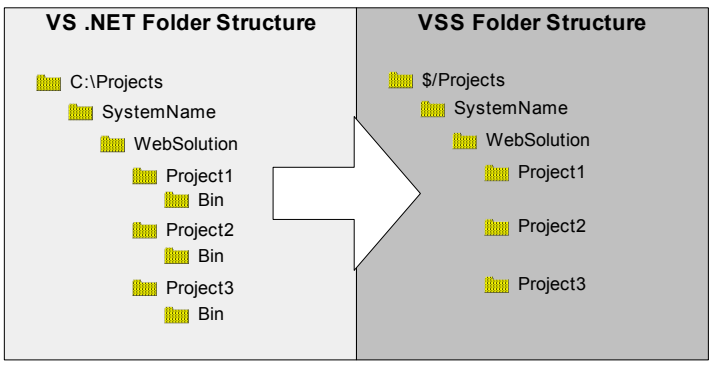


Figure 3.5
Visual Studio .NET and VSS Folder Structure

The following subsections describe how to use the Visual Studio .NET IDE to create the appropriate structures for Web and non-Web applications.

How to Create a New ASP.NET Web Project

By default, when you create a new ASP.NET Web application, the project file is located in a nominated virtual root beneath your default Web site (usually \inetpub\wwwroot) and the associated solution file is located beneath \My Documents\Visual Studio Projects. This default arrangement is not ideal in a team development environment because it breaks the symmetrical structure between your VSS projects and local files.

The following steps guide you through the creation of a new ASP.NET Web application in accordance with the recommended solution and project folder structure. The steps assume a solution called **MyWebAppSolution** and a project call **MyWebApp**.

If you include the word **Solution** (or **Soln**) in the solution name, this helps to clearly differentiate it from the project file name.

The desired structure is shown in Figure 3.6. Note that the project folder and Microsoft Internet Information Server (IIS) virtual root are one and the same.

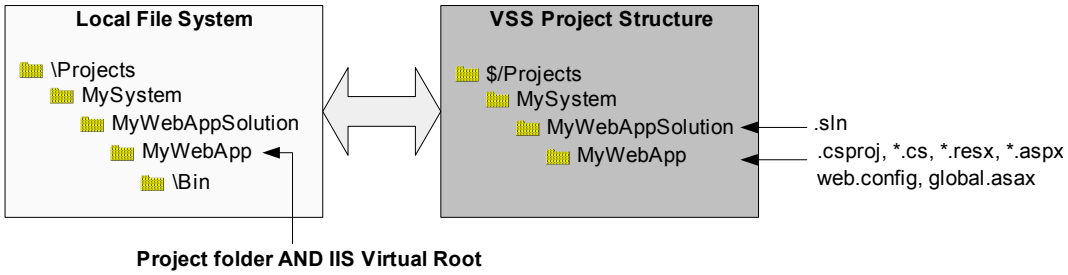


Figure 3.6
Recommended Web Application Structure

► To create a new Web application with this structure

1. Open Visual Studio .NET, and on the **File** menu, point to **New,** and then click **Blank Solution**.
2. Enter **MyWebAppSolution** as the solution name, and then set its location to **\Projects\MySystem**.
3. Click **OK**. Visual Studio .NET creates the MyWebAppSolution folder beneath the specified root folder location.
4. Use Windows Explorer to create a new subfolder called MyWebApp beneath the \Projects\MySystem\MyWebAppSolution folder.

5. Use either Windows Explorer or Internet Services Manager to establish the MyWebApp folder as an IIS virtual root.

   Note: Visual Studio .NET requires that the virtual root name match the project folder name.

6. Return to Visual Studio .NET, and on the **File** menu, point to **Add Project,** and then click **New Project**.
7. Add an ASP.Net Web Application project.
8. In the **Location** field, enter **http://localhost/MyWebApp**, and then click **OK**. The project and Web source files will be created in the specified virtual root.

How to Split a Web Application into Multiple Projects

In some cases, you may want to divide your Web application into multiple projects within the same solution to facilitate team development. For example, if you have separate teams responsible for different aspects of the same Web application, it is useful to be able to divide the application into multiple projects. This capability is not supported natively by Visual Studio .NET, but it can be achieved with some manual manipulation of virtual roots.

Important: A single project approach is simpler, so split a Web application up into multiple projects only if absolutely necessary—typically for very large Web applications.

More Information

For more detailed information about creating sub Web projects, see article Q307467, "HOWTO: Create an ASP.NET Application from Multiple Projects for Team Development," in the Microsoft Knowledge Base at *http://search.support.microsoft.com/*.

Working with ASP.NET Code-Behind Files

You should be aware that when you check out a Web form (aspx) file or a code-behind file (aspx.cs or aspx.vb), Visual Studio .NET automatically checks out both files. This is by design because user interface changes generally involve an aspect of coding within the code-behind file.

For example, if you add a new control to a Web form, Visual Studio .NET automatically creates a member variable for the control in the code-behind file.

How to Create a New Non-Web Project

The following steps guide you through the creation of a new non-Web project type such as a Windows-based or console application, a class library or a service. The steps assume a solution called **MyWinAppSolution** and a project call **MyWinApp**. The desired structure is shown in Figure 3.7.

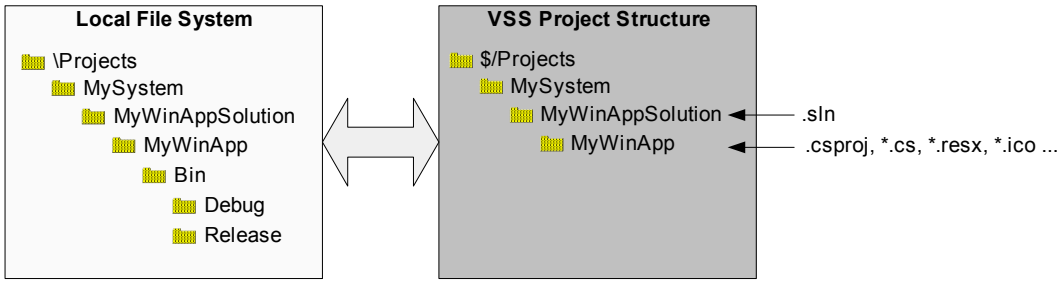


Figure 3.7
Recommended Project Structure for non-Web Projects

► **To create a new non-Web application with this structure**

1. Open Visual Studio .NET, and on the **File** menu, point to **New**, and then click **Project**.
2. Select the appropriate project type and template.
3. Enter the project name in the **Name** field (in this example **MyWinApp**).
4. Set the Location field to \Projects\MySystem.
5. Click the **More** button.
6. Select the **Create directory for solution** check box. This causes the project file to be created in a project sub folder beneath the solution folder.
7. Enter MyWinAppSolution into the New Solution Name field.
8. Click **OK** to complete the project and solution creation process.

For information about how to add the newly created project and solution to Visual SourceSafe, see How to Check In a New Solution to VSS in Chapter 6, "Working with Visual SourceSafe."

Carefully Consider Naming Conventions

Give careful and early consideration to the way you name projects, assemblies, folders, and namespaces. While it is possible to rename these items later on in the development cycle, you should avoid this if possible. For information about how to rename a project, see Renaming a Project in Chapter 6, "Working with Visual SourceSafe."

You should also aim for a consistent set of names because this can greatly simplify project organization.

Use Common Names for Projects and Assemblies

Your output assembly name should always match the project name from which it is generated. For example, you should be able to assume that an assembly called MyCompany.Utilities.Data.dll is generated by a project called **MyCompany.Utilities.Data**.

If you change the name of an output assembly, consider changing the project name to match, and vice-versa.

Use a Common Root Namespace Name

The root namespace into which you place your types (structures, classes, interfaces, and so on) should match the project and assembly name.

For example, use **MyCompany.Utilities.Data** as the root namespace within the MyCompany.Utilities.Data.dll assembly.

While .NET does not require this alignment, it makes sense to synchronize names because it then becomes easy to tell which types live in which assemblies.

Note: Microsoft Visual Basic® .NET projects expose the root namespace via project properties. By default, any type created within the Visual Basic project will be placed inside this namespace. If you use explicit **namespace** statements in your Visual Basic .NET project, delete the root namespace entry, otherwise the explicit namespace name is appended to the root namespace name.

C# projects expose a default namespace property via project properties. This is again used to determine the namespace into which new types added to the project are placed. However, unlike Visual Basic .NET projects, the root namespace is explicitly stated via **namespace** statements within your source files.

Use Common Names for VSS and Local Folders

As mentioned earlier, keep your local solution and project folder names in synchronization with their equivalent VSS folder names.

4

Managing Dependencies

The information in this chapter helps you:

- Manage dependencies and references between projects and solutions.
- Work with dependencies on .NET assemblies, Web services, databases, serviced components, and COM Interop libraries.

You need a consistent and maintainable approach to managing dependencies in a team environment. Dependencies inevitably change over time and as a result they impact the build process (and the build order) of your application.

For example, when a dependency changes, client assemblies must be rebuilt in order to stay in step with the latest version. Depending upon the type of dependency and the way it is referenced, the Microsoft® Visual Studio® .NET build system may or may not be able to automatically handle build ordering issues.

Referencing Assemblies

When you need to use a type (such as a class or structure) contained in another assembly, you must set a reference to that assembly. This creates an assembly reference within the client assembly's manifest that identifies the name and version of the dependency. Visual Studio .NET supports two types of references: project references and file references.

Use Project References

The Projects page within the Visual Studio .NET **Add Reference** dialog box lists all of the other projects in the current solution. This allows you to create a project reference to another project in the same solution. Project references are the recommended way to set references because they offer many advantages.

Note: Project references are the main reason you should adopt a single solution or partitioned single solution model wherever possible.

Advantages of Project References

The advantages of using project references are:

- They work on all development workstations where the solution and project set are loaded. This is because a project Globally Unique Identifier (GUID) is placed in the project file, which uniquely identifies the referenced project in the context of the current solution.
- They enable the Visual Studio .NET build system to track project dependencies and determine the correct project build orders.
- They avoid the potential for referenced assemblies to be missing on a particular computer.
- They automatically track project configuration changes. For example, when you build using a debug configuration, any project references refer to debug assemblies generated by the referenced projects, while they refer to release assemblies in a release configuration. This means that you can automatically switch from debug to release builds across projects without having to reset references.
- They enable Visual Studio .NET to detect and prevent circular dependencies.

Use File References Only Where Necessary

If you can't use a project reference because you need to reference an assembly outside of your current solution's project set, you must set a file reference. The following are the two ways to set a file reference:

- To reference a .NET Framework assembly, you select the assembly from the list displayed on the .**NET** tab of the **Add References** dialog box.
- You can use the **Browse** button in the **Add Reference** dialog box.

If you set a file reference, the path to the assembly is stored in the source controlled project file. A relative path is stored for local assemblies, while the full network path is stored for server-based assemblies, as demonstrated by the following project file snippet. Notice the **HintPath** attributes.

```
<References>
  <Reference
     Name = "System.XML"
     AssemblyName = "System.Xml"
     HintPath =
"..\..\..\..\WINDOWS\Microsoft.NET\Framework\v1.0.3423\System.XML.dll"
  />
  <Reference
    Name = "Lib1"
    AssemblyName = "Lib1"
    HintPath = "\\BuildServer\Latest\Release\SharedComponent\SomeControl.dll"
  />
</References>
```

Note: Assemblies such as System.XML.dll are located in the Global Assembly Cache (GAC). However, you never directly refer to an assembly within the GAC. Instead, when you select an assembly on the **.NET** tab of the **Add References** dialog box, you actually reference a copy of the assembly, located within the %windir%\Microsoft.NET\Framework\<version>\ folder.

Use Copy Local = True for Project and File References

Every reference has an associated **copy local** attribute. Visual Studio .NET determines the initial setting of this attribute (**true** or **false**) when the reference is initially added. It is set to **false** if the referenced assembly is found to be in the GAC; otherwise, it is set to **true**.

You should not change this default setting. With **copy local** set to **true**, the Visual Studio .NET build system copies any referenced assembly (and any dependent downstream assemblies) to the client project's output folder when the reference is set.

For example, if your client project references an assembly called Lib1, and Lib1 depends on Lib2 and Lib3, then Lib1, Lib2, and Lib3 are copied to your project's local output folder automatically by Visual Studio .NET at build time.

Automated Dependency Tracking

Each time you build your local project, the build system compares the date and time of the referenced assembly file with the working copy on your development workstation. If the referenced assembly is more recent, the new version is copied to the local folder. One of the benefits of this approach is that a project reference established by a developer does not lock the assembly dynamic-link library (DLL) on the server and does not interfere in any way with the build process.

Using File References in Single and Partitioned Single Solution Systems

Use project references wherever possible and aim to minimize the use of file references.

If you adopt either the single solution or partitioned single solution model, you need to use file references only to reference outer-system assemblies (that is, those not built by your system build process). Specifically, these include:

1. .NET Framework assemblies that are referenced on the **.NET** tab of the **Add References** dialog box. This does not present an issue within the team environment because .NET Framework assemblies are located in a common location on all development workstations.
2. Outer system assemblies referenced by using the **Browse** button on the **Add References** dialog box. These include third-party components and assemblies built elsewhere within your company outside of your current system. For information about how to manage this type of outer system assembly, see Include Outer System Assemblies within Projects.

Using File References in Multi-Solution Systems

If you adopt a multi-solution model, this forces you to use file references on one further occasion—when you reference an assembly that is generated by a project within a separate solution.

Consider Your Reference Locations

When you are forced to reference a cross-solution assembly, you have the following two choices:

- You can reference an assembly on the build server by using either a virtual drive letter or Universal Naming Convention (UNC) path.
- You can copy the set of assemblies that are generated by the build process on the build server to your local development workstation and then establish a local reference with a virtual drive letter. For more information about using a virtual drive letter, please see Use a Virtual Drive for Greater Flexibility.

Advantages of Referencing Assemblies from a Build Server

The advantages of this approach are:

- You guarantee to reference the latest version of a particular assembly as the build process updates these assemblies at regular (for example, daily) intervals.
- The paths within your project files that contain virtual drive letters or UNC paths work across all development workstations and the build server.

Disadvantages of Referencing Assemblies from a Build Server

This approach does suffer from a couple of drawbacks which may be significant in some development environments (particularly the larger ones). The following are some disadvantages of this approach:

- You are not in direct control of when an assembly that you reference is updated. As a result, it's possible that a new version of an assembly on the server could break your local build process at an inconvenient time when you are in the middle of developing and debugging a different area of the system. Although the central build process usually runs overnight, on occasion, interim builds need to be generated during the day. These interim builds have the potential to cause problems.
- This approach does not support disconnected development. You are required to have a direct connection to the build server whenever you build a local project that references an assembly on the build server.

Consider an Isolated Development Approach

If you require a high level of isolation during your development work, you can adopt an isolated development approach. With this approach, you:

1. Copy the build output from the build server to a common location on your development workstation. This can be performed manually or with the help of a script.
2. Establish a common virtual drive (for example, drive R) so that all developers reference assemblies with the same path.
3. Set references to the local assemblies by using the virtual drive letter (drive R).
4. Periodically check for new build output on the build server and manually copy it locally at your convenience.

Use a Virtual Drive Letter for Greater Flexibility

If you reference a local assembly that has been copied from the build server, you should do so with a virtual drive letter established with the **subst** command.

Important: All developers within the team must adopt the same drive letter because it is maintained within the source controlled project file as illustrated in the following code.

```
<References>
  <Reference
    Name = "Lib1"
    AssemblyName = "Lib1"
    HintPath = "R:\Latest\Release\SharedComponent\SomeControl.dll"
  />
</References>
```

An additional advantage of using the virtual drive letter is that it enables you to easily remap it to different locations. For example, you may want to map it to the build server for the majority of your development time, but when you need a period of isolated development you can remap it locally

Always Reference Release Builds with File References

The build script updates the build server on a regular (typically daily) basis with current assemblies. The build script typically generates debug and release versions of your assemblies. Developers use the debug DLLs and testers use the release DLLs, as follows:

- As a developer, you install the debug DLLs to perform your development and unit testing work.
- Members of the test team install a release version of the system and always test with release DLLs. Do not delay the generation of a release build until your product release date is imminent because the release build could potentially exhibit problems not present within the debug build.

To accommodate debug and release builds, the build script copies assemblies to Release and Debug folders on the build server. For detailed information, see Chapter 5, "The Build Process."

As a developer, you should always reference assemblies from the Release folder for the following reasons:

- These ultimately represent the versions of dependent assemblies that are deployed into production.
- You never need to change file references from the Debug folder to the Release folder in order to generate a release build. File references do not dynamically change to track configuration changes in the same way that project references do.

Use the Reference Path to Assist Isolated Development and Debugging

The one issue with referencing release assemblies is that you are unable to debug and step into the assembly. If you need to debug a referenced assembly contained within a separate solution:

1. Copy the solution to your development workstation
2. Rebuild a debug version of the assembly.
3. Set the reference path within the client project to point to the debug output folder of the referenced assembly.
4. Rebuild and run the client project. This results in the local debug version of the referenced assembly being copied to the client's project output folder. You can then debug and step into the referenced assembly.
5. When you complete debugging and want to revert back to referencing the assembly from its usual location, remove the reference path entry.

What is the Reference Path?

The reference path is a per-computer, per-developer setting that is maintained as an Extensible Markup Language (XML) element in the project user options file (*.csproj.user or *.vbproj.user). It is used by Visual Studio .NET to help locate assembly references at build time.

Important: If you follow the guidelines presented earlier and use either project references or file references (with virtual drives) to reference assemblies, all references are directly resolved by Visual Studio .NET on any computer and the reference path will not usually be required. However, the reference path can be useful on occasion because it allows you to override the path maintained by the <**HintPath**> element in the source controlled project file.

Resolving Assembly References at Build Time

At build time, Visual Studio .NET resolves assembly references by searching the following locations in the following order:

1. Look for the assembly in one of the project folders. This assumes that you have added the assembly to the project by using the **Add Existing Item** menu option. Project folders include any folder displayed by Solution Explorer (except when **Show All Files** is in effect).
2. Look in the folders listed in the **ReferencePath** attribute of the <**Settings**> element within the project user options file. This attribute can contain a comma delimited list of folders.
3. Use the <**HintPath**> element in the project file.
4. Look in a set of folders identified by registry settings. These are the ones that contain assemblies displayed on the .**NET** tab of the **Add references** dialog box. For more details, see Using the .NET Tab of the Add Reference Dialog Box.
5. Look for COM Interop assemblies in the **obj** sub folder beneath the project folder. For more details, see Referencing COM Objects.

Notice that the reference path in the project user options file takes precedence over the hint path established when you set a file reference.

Performing Isolated Development and Unit Testing

The reference path can also help you perform isolated development and unit testing within multi-solution systems. Consider solution SA and solution SB where a project within SA named PA depends upon a library project within SB called PB. If you want to perform some isolated development and unit testing on these two solutions in advance of the next system build, you can change PA's reference path to point to the output folder of PB. This allows you to make changes to PB, rebuild it locally, and then test against it in SA.

You can do this without checking in SB again and waiting for the next build process to regenerate its assemblies.

Because the reference path is a per-developer setting in the project user options file (which is not added to source control), when you next check the solutions back into VSS, you will not impact the build process or fellow developers.

How to Set the Reference Path for a Specific Project

Use the following steps to set the reference path property for a specific project.

► **To alter the reference path for a specific project**

1. Right-click the project within Solution Explorer, and then click **Properties**.
2. Expand the **Common Properties** folder, and then click **Reference Path**.
3. For C# projects, click the **New Line** icon and enter the new reference path.

   –or–

   For Visual Basic .NET project, enter the reference path into the **Folder** field, and then click **Add Folder**.
4. Click **OK** to close the **Properties** dialog box.

Include Outer System Assemblies within Projects

The best way to handle outer system assemblies such as third-party Web controls or components that are not rebuilt by your build process is to include them directly into those projects that need to reference them. Conceptually, think of outer system assemblies the same way as .bmp or .gif files.

► **To include and then reference an outer-system assembly**

1. In Solution Explorer, right-click the project that needs to reference the assembly,,and then click **Add Existing Item**.
2. Browse to the assembly, and then click **OK**. The assembly is then copied into the project folder and automatically added to VSS (assuming the project is already under source control).
3. Use the **Browse** button in the **Add Reference** dialog box to set a file reference to assembly in the project folder.

Advantages

There are a couple of advantages to this approach:

- Outer system assemblies remain source controlled alongside the project files. When a new version of the assembly is available, the file can be added to VSS as a new file version complete with its own file history.
- Most importantly, your entire system is contained within VSS; this includes all outer-system assemblies, such as third party controls. You can retrieve an earlier version of the system from VSS, including all source code and external dependencies. This allows you to have a complete snapshot of the earlier system version.

Consider Sharing Outer System Assemblies in VSS

If a particular outer system assembly is referenced by multiple projects, it is maintained within every project. This makes updating the assembly to a later version a more difficult task.

You can address this issue by sharing the outer system assembly in VSS between the projects that use it. This allows the file to be updated within one project. You can refresh the copies maintained within other projects by right-clicking the project within Solution Explorer, and then clicking **Get Latest Version (Recursive)**.

Using the .NET Tab of the Add Reference Dialog Box

The **.NET** tab of the **Add Reference** dialog box displays system assemblies and Primary Interop Assemblies that are supplied with the .NET Framework and optionally other assemblies. These are usually (but not necessarily) those installed in the GAC.

You can enable your own assemblies to appear in this list, but this requires a registry modification to designate the folder or folders that contain your assemblies.

For example, you could apply registry updates to point to the folders that contain assemblies built by your build script (either locally or on the build server). This allows developers to reference these assemblies from the .**NET** tab and avoids the use of the **Browse** button.

► To add your own assemblies to the .NET tab

1. Create a new registry key (for example, one called **InnerSystemAssemblies**) beneath either of the following registry keys:

   ```
   HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AssemblyFolders
   HKEY_CURRENT_USER\Software\Microsoft\.NETFramework\AssemblyFolders
   ```

2. Set the new key's default value to point to the folder that contains your assemblies.
3. If you have Visual Studio .NET open, you must close it and then launch it again for the changes to take effect.

Referencing Web Services

In single solution systems, all developers end up with local working copies of all Web services because they are defined by projects within the single solution. When you open a solution from VSS for the first time, all projects (including any Web services) are installed locally. Similarly, if a Web service is added to the solution by another developer, you install the Web service the next time you refresh your solution from VSS. In the single-solution world, there is no need to publish Web services on a central Web server within the team environment.

For Web services developed as part of a multi-solution system, not all developers need to locally install the Web service.

Within a multi-solution system, the developer of the Web service should publish the service on the central development Web server to allow other developers to access it from their client projects.

Versioning Web Services in Development

The Web services that you currently have in development will evolve over time. You should aim to keep the Web server up to date with the latest versions.

In many respects, the versioning issues associated with Web services are similar to those associated with databases. When a database schema is updated, the change must be planned and coordinated so that dependent client applications can be updated in synchronization with the schema change.

You should handle updates to Web services in a similar way. When you change the interface of a Web service, the team responsible for the service must publicize the change so that other dependent teams can update their client-side references.

Note: Implementation changes require less coordination than interface changes. However, in both cases, the team responsible for the Web service should publish change details to other developers and development teams.

Always Use Dynamic URLs

If you want to call a Web service, you must first add a Web reference to your project. This generates a proxy class through which you interact with the Web service. The proxy code initially contains a static Uniform Resource Locator (URL) for the Web service, for example http://localhost or http://SomeWebServer.

Important: For Web services in your current solution that execute on your computer, always use http://localhost rather than http://MyComputerName to ensure the reference remains valid on all computers.

The static URL that is embedded within the proxy is usually not the URL that you require in either the production or test environments. Typically, the required URL varies as your application moves from development to test to production. You have two options to address this issue:

- You can programmatically set the Web service URL when you create an instance of the proxy class.

- A more flexible approach that avoids a hard coded URL in the proxy, is to set the **URL Behavior** property of the Web service reference to **dynamic**. This is the preferred approach. When you set the property to **dynamic**, code is added to the proxy class to retrieve the Web service URL from the **<appSettings>** section of the application configuration file, Web.config for a Web application or SomeApp.exe.config for a Windows application.

  The dynamic URL approach also lets you provide a user configuration file, which can override the main application configuration file. This allows separate developers (and members of the test team) to temporarily redirect a Web service reference to an alternate location.

How to Use Dynamic URLs and a User Configuration File

Set the **URL Behavior** property of your Web service references to **dynamic** to gain maximum configuration flexibility both within the development and production environments. Set the production URL of the Web service in the application configuration file and provide a user configuration file for development and test purposes. The absence of the user configuration file results in the application configuration file being used.

► To specify a Web Service URL in a user configuration file

1. Add a **file="user.config"** attribute to the **appSettings** element of your main application configuration file. This silently redirects the runtime to the named user configuration file when it accesses information from the **appSettings** section. If the user configuration file is missing, the settings from the main application configuration file are used instead and no runtime error is generated.

```
<configuration>
 <appSettings file="user.config">
   <add key="ClientApplication.SomeServer.SomeService"
        value="http://ProdWeb/myXmlWebService/Service1.asmx"/>
 </appSettings>
</configuration>
```

   In the preceding example, **ClientApplication.SomeServer.SomeService** is the qualified name of the Web service proxy class. **ClientApplication.SomeServer** is the namespace and **SomeService** is the proxy class name.

2. Create a User.config file (located in the same folder as the application configuration file), and add an **appSettings** entry that contains a key-value pair that identifies the Web service URL, as illustrated in the following code. Notice that in this example, the URL references the local Web server. Also notice that the **<configuration>** element is missing from the User.config file.

```
<appSettings>
   <add key="ClientApplication.SomeServer.SomeService"
        value="http://localhost/myXmlWebService/Service1.asmx"/>
</appSettings>
```

3. Don't check in the User.config file to VSS. In this way, each developer (and the test team) can explicitly bind to specific URLs through their own User.config file entry. The main application configuration file should maintain the production Web service address. This is used in the absence of a User.config file.

Tip: By default, the user configuration file is automatically added to VSS. To prevent this, right-click the file within Solution Explorer, and then click **Exclude From Project**. To subsequently view the file within Solution Explorer, click the **Show All Files** icon at the top of the Solution Explorer window.

Important: For Web applications that employ a user configuration file, any changes made to the file do not result in the Web application being automatically recycled. This only happens for the Web.config file. As a result any changes to the user configuration file are not immediately seen by the application. You must manually stop and restart the Web application. This is another reason why you should use the Web.config file for production settings and use the User.config file for only development and test settings.

Updating a Web Service Reference

To update an existing Web Service reference, in Visual Studio .NET Solution Explorer, right-click the file, and then click **Update**. The next time you rebuild your project, new service information in the form of a Web Services Description Language (WSDL) document is downloaded and a new local Web service proxy class is created.

Referencing Databases

Database references in the form of connection strings can also be managed by using the User.config file. The advantage of this is that each developer can easily specify his own connection string is his own private User.config file. Any changes made by one developer, such as redirecting the connection to a local database for unit testing purposes, does not affect other developers.

User configuration files can also be used to control environmental-specific settings, such as those required by a test environment. The test environment can also use a User.config file which references the test database.

The procedure is similar to the preceding Web references example, except that in that example the Web service proxy contains the code to retrieve the Web service URL from the configuration file. For database connection strings, you must provide the code to read the connection string.

How to Use User Configuration Files for Database Connection Strings

The following procedure explains how to store and then reference a database connection string within a user configuration file.

► **To use a user configuration file to store database connection strings**

1. Add a **file="user.config"** attribute to the **appSettings** element of your main application configuration file and include a default connection string in the main configuration file. This is used in the absence of any user-specific override.

```
<configuration>
 <appSettings file="user.config">
  <add key="DBConnStr"
     value="server=PRODDB;Integrated Security=SSPI;database=Accounts"/>
 </appSettings>
</configuration>
```

2. To override the main application configuration file, create a User.config file (located in the same folder as the application configuration file), and then add a similar **appSettings** entry to the file. Notice that the following connection string references a local database.

```
 <appSettings>
  <add key="DBConnStr"
     value="server=(local);Integrated Security=SSPI;database=Accounts"/>
 </appSettings>
```

3. Within your project, use the following code to obtain the connection string from the user configuration file (if it is present) or the application configuration file if it is not. This code uses the static **AppSettings** property of the **System.Configuration.ConfigurationSettings** class.

```
using System.Configuration;
private string GetDBaseConnectionString()
{
  return ConfigurationSettings.AppSettings["DBConnStr"];
}
```

Database Development

There are two basic approaches for database development in the team environment:

- Central database server(s)
- Central database servers(s) and local databases on developer workstations

Central Database Servers

The central database server or servers in the team environment mirror the production schema—or if the system is not yet in production, they reflect the most up-to-date schema. The following guidelines govern the use of database servers within the team environment:

- Do not give developers administrative access to the database server and do not allow changes to the schema to be made by individual developers. You need to have careful control over the environment because changes made by one developer can easily impact others.
- Grant developers specific permissions; for example, grant developers the permission to write stored procedures, functions, and (possibly) views.
- Manage schema changes and other objects such as stored procedures with source controlled database scripts.
- Use database scripts to ease the creation and recreation of databases.
- Provide separate database instances for separate development efforts because configuration settings may be different for each development project.

Local Databases

The main problem with working solely against one or more central development database servers is that it is difficult to maintain a consistent set of test data. The unit testing efforts of one developer can easily impact another because test data is likely to change at frequent intervals.

As a result, it is common to install Microsoft SQL Server™ Developer Edition on each development workstation to provide an isolated test environment for each developer.

Use Database Scripts for Managing Change

You should manage all changes to the database on either the local server or on the central server through source controlled database scripts. You should provide scripts for any changes you might make manually through Enterprise Manager. You should use database scripts (that contain as much auditing and error logging as possible) to:

- Enable a database (including stored procedures and so on) to be installed from scratch.
- Enable a fresh set of test data to be loaded.
- Apply updates to database schema and database objects.

Consider Visual Studio .NET Database Projects

Database scripts should be source controlled and maintained within VSS. You have two options for handling scripts:

- Handle scripts outside of the Visual Studio .NET integrated development environment (IDE) and manually create a suitable folder structure within VSS. For example, you could create a subproject folder beneath **$/Projects/SystemName** called **Database Objects** to maintain database specific code and objects and then use separate subfolders such as **Tables and Views**, **Diagrams and Documentation**, and **Stored Procedures and Functions** as containers for the various object types.
- Use Visual Studio .NET Database Projects. These are simple file based projects that allow you to store and execute database scripts, and store other information associated with your databases, such as database documentation or build files. Visual Studio .NET provides an integrated editor for Transact-SQL code, a visual Query Builder and T-SQL debugging support. The advantage of this approach is that it provides tight and automatic integration with VSS, in common with other project types.

For more information about database projects, refer to the section, "Managing Data with Database Projects within the Developing with Visual Studio .NET," of the Microsoft MSDN® Library. Also search for "Large Database Projects" within MSDN.

Referencing COM Objects

When your code calls a COM object, an Interop assembly is used to handle the type conversions between the .NET world and the COM world. When the COM object is called from managed code, a proxy within the Interop assembly is actually called. The proxy initially performs the necessary type conversion and then invokes the COM object to perform the required work.

Always Generate Compatible Interop Assemblies

In a team development environment, if you and another developer reference the same COM DLL from two different projects, the result is two Interop assemblies—one for each project.

In most circumstances, the type of the two assemblies that are created are the same, and as a result you can safely pass a reference to the COM object (via the Interop assembly) from one project to another.

To guarantee that the type of any generated Interop assembly is the same, you must ensure that the following conditions are met. Failure to do so results in a type mismatch exception at runtime when the reference is passed from one project to another even if the Interop Assembly was generated from the same COM DLL. The conditions that must be met are:

1. All Interop assemblies must be generated from exactly the same version of the COM type library.
2. The identity of all Interop assemblies must be the same. The assembly identity includes:
   a. The file name without extension.
   b. The public key, which may be null.
   c. The version.
   d. The culture (usually neutral for code).

The preceding conditions are generally met when you generate the Interop assembly from the Visual Studio .NET project system, by selecting the COM type library from the **Add References** dialog. The only exception is that it is possible (for C# projects) for the Visual Studio .NET project system to assign a strong name to Interop assemblies (the project properties dialog supports a Wrapper Assembly Key File property). In this case it is possible that two different developers could generate two Interop assemblies that are incompatible with one another.

In a team development environment, to guarantee that only one Interop assembly exists for any given version of a COM type library, you should adopt one of two approaches:

1. Always use Primary Interop Assemblies (PIAs).
2. If you cannot obtain a PIA, manually generate a single Interop assembly (by using tlbimp.exe) and then:
   a. Optionally assign the assembly a strong name
   b. Include it directly into those projects that need to reference it.

The following section describes both of these approaches.

Use Primary Interop Assemblies Whenever Possible

A Primary Interop Assembly (PIA) is an Interop assembly that is officially signed by the provider of the COM object and is the Interop assembly to use in all cases. If you don't have a PIA, you should request one from the provider of the COM object.

If you have a PIA, treat it like an outer-system assembly and include it directly into the project that references it. This results in the assembly being copied to the project folder. Set a file reference to the assembly in the project folder. The process is the same as the one described earlier, in Include Outer-System Assemblies within Projects.

Note: The PIAs that are supplied with the .NET Framework are located in the \Program Files \Microsoft.NET\Primary Interop Assemblies folder. When you obtain new PIAs, do not place them in this folder; if you do, you need to update all of the development workstations and build server in the team development environment. Instead, include them directly into the specific projects that need to reference them.

Use TLBIMP if you don't have a Primary Interop Assembly

If you don't have a PIA, create a single Interop assembly by using the Tlbimp.exe tool and optionally assign it a strong name. Like a PIA (and outer system assemblies), include the manually generated Interop assembly in any project that needs to reference it. This results in the assembly being copied to the local project folder from where you can reference it with a file reference.

Register COM Classes Locally

If you add an Interop assembly to a project, it does not result in the associated COM DLL being copied locally (and registered) and so you must register the COM DLL on each developer workstation. This unfortunate consequence is a result of interacting with the unmanaged world of COM.

Calling Serviced Components

A serviced component is a .NET managed class that derives from the **ServicedComponent** class within the **System.EnterpriseServices** namespace. Such classes can be hosted by COM+ applications and can use COM+ services.

In a team development environment, adopt the following recommendations when using serviced components.

Use Delayed Signing

Serviced components must have a strong name. To assign a strong name, you need a public and private key-pair that can be generated by the Sn.exe tool. In many companies, the private key is a closely guarded secret and is not available to developers on a daily basis. As a result, you should use delayed or partial signing which can be performed with only access to the public key.

The partial signing process results in a place holder being created within the assembly portable executable (PE) file for the strong name signature. The actual signing is deferred until a later stage, between system testing and product release. The build coordinator is generally responsible for finally signing assemblies.

Use Dynamic Registration

You should add the relevant assembly attributes (for example, **ApplicationName**, **ApplicationID**, and **ApplicationActivation**) to support dynamic registration. This allows the COM types associated with your serviced components to be automatically installed into the COM+ catalog on all development workstations, as soon as an instance of the serviced component is created.

Control the CLSID Used Within the COM+ Catalog

By default, when you rebuild an assembly it is assigned a new version number. This is because Visual Studio .NET sets the **AssemblyVersion** attribute to "1.0.*" for new projects. As a result, a new Class Identifier (CLSID) is generated for serviced components each time the assembly is rebuilt.

Note: This behavior is slightly different between C# and Visual Basic .NET projects. For C# projects, the assembly version is incremented every time it is rebuilt. For Visual Basic .NET projects, the assembly version is incremented the first time the project is rebuilt after it is loaded into Visual Studio .NET. Subsequent rebuilds within the same instance of Visual Studio .NET do not result in the assembly version being incremented.

This does not represent a problem, because the assembly version is for information only in assemblies that do not have a strong name. For strong named assemblies, you should use static version numbers that are manually maintained. This and other versioning issues are discussed further in Controlling Assembly Version in Chapter 5, "The Build Process."

In order to control the CLSID that serviced components end up with in the COM+ catalog and avoid multiple versions appearing each time a developer rebuilds the serviced component, use either of the following approaches:

1. Explicitly control the CLSID by using the following **Guid** attribute:

```
[Guid("2136360E-FEBC-475a-95B4-3DDDD586E52A")]
public interface IFoo
{
}

[TransactionAttribute(TransactionOption.Required),
Guid("57F01F20-9C0C-4e63-9588-720D5D537E66")]
public class Foo: ServicedComponent, IFoo
{
}
```

2. Maintain a static assembly version number for the serviced component's assembly and don't use the Visual Studio .NET default "1.0.*" version numbering scheme. For more information about assembly versioning, see Controlling Assembly Version in Chapter 5, "The Build Process."

5

The Build Process

This chapter will help you:

- Manage versioning and dependency relationships.
- Automate builds using the appropriate scripts and tools.
- Organize and distribute build output.

The build process is a critical element for all software development projects. Do not be tempted to get by without one—particularly in a team development environment. You should configure a build server and create the necessary build scripts as early as possible in the development cycle—certainly well before you are ready to begin integration testing.

The main function of a build script is to provide an automated way to generate system builds in a repeatable and consistent manner. Under normal circumstances, you schedule the build script to run at night, to avoid placing unwanted stress on the build server, Microsoft® Visual SourceSafe™ (VSS) server and the network during development hours.

The build script accesses VSS by using the VSS automation model or by calling VSS from the command line. It labels and extracts the latest versions of source and project files and then it uses Microsoft Visual Studio® .NET (by executing devenv.exe from the command line) to build your solution or solutions. The build script typically builds both a debug and release version of your system.

Handling Dependency Relationships

In single solution or partitioned single solution systems, project references take care of dependencies and build order. The build script for these types of systems is simple because it can execute Devenv.exe once against a single solution.

The build script is more complex for multi-solution systems because it must build solutions in the correct order, based on known dependency relationships. This is to ensure that when an assembly is updated (and its version changes), all client assemblies that reference it are also updated and rebuilt against the latest version.

Controlling Assembly Version

An assembly's version is specified via the **AssemblyVersion** attribute (which is defined within the AssemblyInfo.cs or AssemblyInfo.vb file). The version number is physically represented as a four part number separated with periods:

```
<major version>.<minor version>.<build number>.<revision>
```

You do not have to set and update each part explicitly because you can use wild characters (*) to automatically generate the build and revision numbers. Visual Studio .NET generates an AssemblyInfo source file with the **AssemblyVersion** attribute defined as follows:

```
[assembly: AssemblyVersion("1.0.*")]
```

The result of this is a build number set to the number of days since a random, designated start date and the revision based on the number of seconds since midnight.

You can either use auto-increment version numbers or opt to manually control version numbers as part of the build process. Each approach has associated benefits and drawbacks.

Note: For a Microsoft Visual Basic® .NET project with an **AssemblyVersion** set to "1.0.*", the assembly version is only updated the first time the project is rebuilt within the Visual Studio .NET integrated development environment (IDE). The version number remains constant for subsequent rebuilds within the same instance of Visual Studio .NET. This does not represent a problem because the assembly version is for information only in assemblies that do not have a strong name. For strong named assemblies, you should avoid the use of wild characters in the **AssemblyVersion** attribute, as explained in the following section.

For C# projects with an **AssemblyVersion** set to "1.0.*", the assembly version is updated every time the project is rebuilt.

Using Auto-Increment Version Numbers

An auto-increment version number is established by adopting the default "1.0.*" pattern for the **AssemblyVersion** attribute.

Advantages

Using auto-increment version numbers has the following advantages:

- The build and revision numbers are handled automatically by Visual Studio .NET and do not have to be handled by the build script or build coordinator.
- You guarantee never to have different builds of an assembly with the same version number.

Disadvantages

Using auto-increment version numbers has the following disadvantages:

- The internal assembly build number does not match your system build number, which means there is no easy way to correlate a particular assembly with the build that generated it. This may be particularly problematic when you need to support your system in a production environment.
- Build and revision numbers are not increased by one but are based on the time an assembly is built.
- A new version of an assembly is generated each time it is built regardless of whether any changes have been made to the assembly. For strongly named assemblies, this means that all clients of that assembly must also be rebuilt to point to the correct version. However, if the build process rebuilds the whole system this should not be an issue.

Using Static Version Numbers

With this approach, you use a static version number, for example "1.0.1001.1", and update the major or minor numbers only when a new version is shipped with your next system release.

Advantages

Using static version numbers has the following advantages:

- You have complete control over the exact version number.
- Assembly build numbers can be synchronized with the system build number.

Disadvantages

Using static version numbers has the following disadvantages:

- The version numbers must be manually updated by the build coordinator or by the build script.
- If the version is not incremented with every build, you may end up with multiple builds of the same assembly with the same strong name. This is undesirable and can be problematic for assemblies that are installed in the Global Assembly Cache (GAC).

Important: If you do not change the version of a strongly named assembly and attempt to install it into the GAC using the Microsoft Windows® operating system Installer, the latest dynamic-link library (DLL) does not install if a previous version exists in the GAC with the same version number.

If you use Gacutil.exe instead of Windows Installer to install the assembly, the updated DLL is installed even if the assembly version number is the same.

Consider Centralizing Assembly Version Numbers

To update the assembly version information for multiple assemblies, the build script or build coordinator must check out and update multiple AssemblyInfo files. To centralize the version information and enable a single file checkout and update, consider the following approach:

1. Place the **AssemblyVersion** attribute in a single source file, for example AssemblyVersionInfo.cs or AssemblyVersionInfo.vb.
2. Share the file across any projects that need to share the same version number within VSS.

This approach assumes that you want the same assembly version assigned to all assemblies generated by a particular system build. This may or may not be appropriate for your particular system.

Build Server Folder Structure

You should set up the same basic folder structure on the build server as your development workstations. Therefore, the basic folder structure discussed within Use a Consistent Folder Structure for Solutions and Projects in Chapter 3, "Structuring Solutions and Projects," remains the recommended structure for the build server.

Consider Maintaining Previous Builds

You may also want to create a folder structure based on build numbers to maintain the output from previous system builds. The benefit of this is that it allows the test team to easily install and test specific versions of your system without being affected by ongoing development changes.

The build server folder structure is illustrated in Figure 5.1.

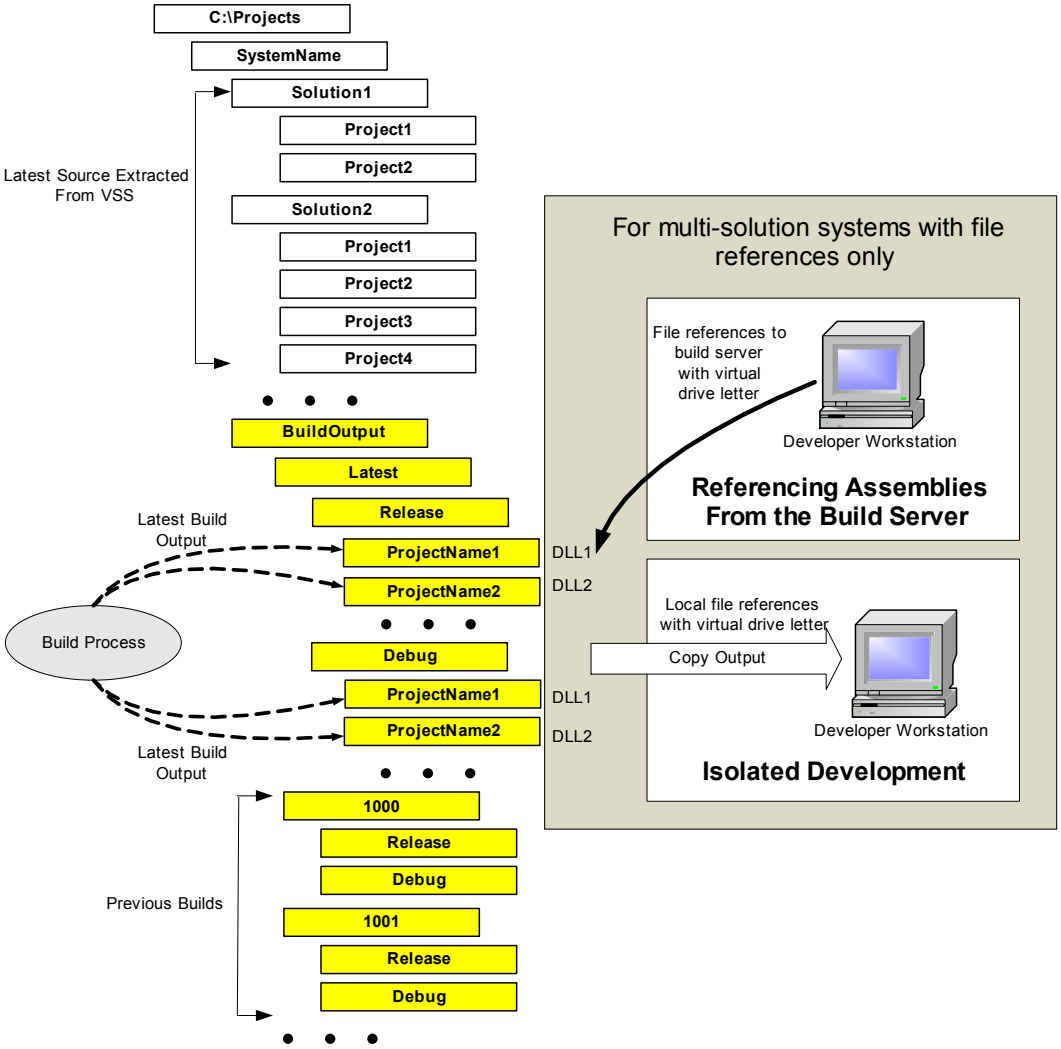


Figure 5.1
Build Server Folder Structure

Note the following points from Figure 5.1:

- Builds are organized by build number. A simple approach for generating build numbers is described later in this chapter.
- The Latest folder always contains the output from the latest build—this matches the binaries contained in the current highest build number folder.

- The contents of each project's output folder is copied by the build script to a subfolder beneath the Latest\Release folder, the name of which is based on the project name.
- In a multi-solution system, you can reference assemblies built in other solutions from folders beneath the Latest\Release folder. These folders are repopulated after each build; this guarantees that you always reference the most up-to-date assemblies with the latest version numbers.
- If isolated (and potentially disconnected) development is required within a multi-solution system, developers copy the build output to their local workstations and reference local assemblies by using a virtual drive letter.

Don't Alter the Build Output Path

You might be tempted to alter the output paths of your projects in order to build to a single folder and then establish file references to that folder. Do not do this for the following reasons:

- It causes the build process to fail with file lock errors when a referenced assembly exceeds 64 KB in size. This problem is likely to be fixed in a future version of Visual Studio .NET.
- You can encounter another problem when you reference an assembly from a folder that is designated as your project's output folder. In this event, Visual Studio .NET cannot take a local copy of the assembly because the source and destination folders for the copy operation are the same. If you subsequently remove the reference from your project, Visual Studio .NET deletes the "local" working copy. Because the local folder is the same as the assembly's original folder, the original assembly is deleted.

The Build Script

Figure 5.2 illustrates the individual steps that should be performed by your build script.

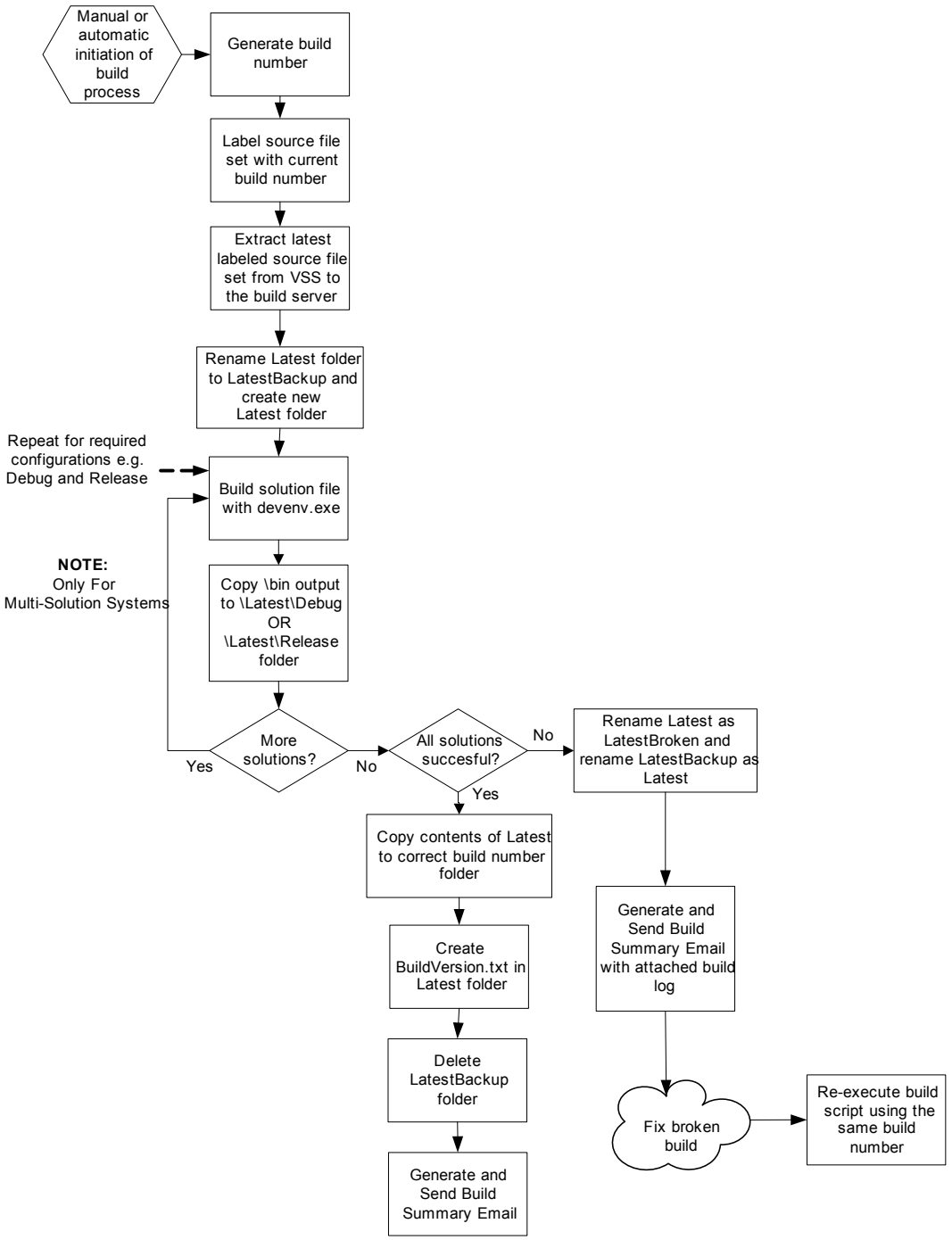


Figure 5.2
The Build Process

The following subsections describe the main steps performed by the build process.

Generating Build Version Numbers

Successive builds are identified by unique build numbers. Build numbers are used to name the output folder that contains the results of a particular build and to label source files within VSS.

Build numbers are generally incremental numbers that are unique across the system. There are many ways to generate build numbers. One simple approach is to maintain a text file in the Latest folder whose file name represents the latest build number; for example, 2021.txt. The build script can access this file to generate the next number.

Labeling Source Files

It's important to establish a relationship between a particular build and the set of project and source files that are used to generate that build. The simplest way to do this is to label the latest source file set with the current build number.

This allows you to recreate the exact build at any time in the future if the need arises. It also helps when you need to resolve a broken build, as described later in this chapter.

The following script fragment shows how to use the VSS command line to label a set of files within a given VSS project folder. To label all source files within your system, the top level VSS folder should be set to $/Projects/SystemName.

```
' Run VSS with the Label command with options to disable UI and answer Yes
' to any prompts. Specify output log file location, label to write and VSS project
to label. Label is recursive therefore only one command is needed against the
' top-level VSS project.
' Current version number is held within the Version variable
WSHShell.Run "ss Label -I-Y -O@..\SSafeOut_Label.txt -LVersion1." & Version &
SSafeProj, 1, true
```

Extracting the Latest Source File Set

The build script performs a get operation to extract the latest set of source files from VSS. It uses the assigned label to pull the correct set of source files. Source files are extracted into the common folder structure on the build server, as illustrated earlier in Figure 1.

The following script fragment shows how to perform the get operation by using the VSS command line.

```
' Current version number is maintained in the Version variable
WSHShell.Run "ss Get -I-Y -O@..\SSafeOut_Get.txt -VLVersion1." & Version &
SSafeProj, 1, true
```

Creating a New Latest Folder

The build process always copies generated assemblies into the Latest folder, to ensure that any file references used within multi-solution systems resolve to the latest assembly versions.

The build script initially renames the current Latest folder to LatestBackup and then creates a new (and empty) Latest folder. This allows the script to roll back to the output of the previous build if the current one fails.

Building Solutions with Devenv.exe

The script builds individual solutions by executing Devenv.exe, supplying at minimum the solution file name, the **/rebuild** switch (which ensures all output and temporary output files are deleted at the start) and the required solution configuration. The build script should execute Devenv.exe twice, once to generate a release build and once to generate a debug build. The following script fragment shows how to execute Devenv.exe.

```
' Call devenv with the rebuild option (clean up previous build leftovers),
' the specified build configuration, the out option to enable logging to a file
' and finally the solution file to build specified via the command line. A count
' is used when deriving the log file name to ensure a separate log is generated
' for each solution build in a multi-solution system
WSHShell.Run "devenv /rebuild " & Config & " /out buildoutput"&Count&".txt " &
SolutionName, 1, true
```

The build script parses the output build log generated by Devenv.exe to see whether or not the latest solution build was successful.

For multi-solution systems that include cross solution file references, the release build of a particular solution must always be generated before the debug build to ensure that any file references can be resolved. This assumes that you set file references to the Release version of assemblies as advocated within Always Reference Release Builds with File References in Chapter 4, “Managing Dependencies.”

Copying Output to the Latest Folder

Single solution or partitioned-single solution systems do not require a Latest folder because you use project references rather than file references to refer to inner-system assemblies. For these types of system, the build script can simply archive assembly output beneath the relevant build number folder.

For multi-solution systems however, after each solution is built, the output assemblies for all constituent projects must be copied to the Latest\Release or Latest\Debug folders in order to guarantee that projects that are built subsequently within other solutions reference the latest assembly versions.

Note: The build script can use the number of solutions passed via the command line to infer whether or not it is dealing with a multi-solution system. If the solution count is greater than one, it must copy assembly output to the Latest folder. Otherwise, it can omit this step.

Organize Assembly Output beneath the Latest Folder

Assemblies are actually copied to subfolders beneath Latest\Release and Latest\Debug, the names of which are based on the project names. The reason that this approach is preferred to placing all output assemblies directly into the Latest\Release folder is that it clearly organizes the output and it allows two versions of outer system assemblies (such as third-party assemblies) with the same name to be included in different projects.

Remember that outer system assemblies are not built as part of the build process and should be included in the projects that need to reference them. It is possible (although unlikely) that two projects reference different versions of the third-party assembly (with the same name). If you separate build output into distinct subfolders, you can ensure that these DLLs do not overwrite one another when the build script copies its output to the Latest folder.

Copying the Latest Folder to a Build Number Folder

If the build of a multi-solution system is successful, the contents of the Latest folder are copied to a folder named with the current build number. This allows successive builds to be archived.

Renaming the Latest Folder as LatestBroken

If the build fails, the build script renames the Latest folder as LatestBroken and reinstates the LatestBackup folder as the Latest folder. This allows any developers who are currently referencing the Latest folder on the build server to continue with their local development.

Resolving a Broken Build

The build coordinator must work to resolve a broken build as quickly as possible. Typically one or more source files are at fault and these either need to be updated within VSS or the latest changes need to be backed out all together.

If you update source files within VSS, you must manually label them with the current build number using VSS Explorer. This ensures that the updated files are associated with the correct build.

After you update the relevant files and you want to rerun the current build, the build script should be supplied with the current build number as a command line argument. The absence of a build number signifies a new build. If a build number is supplied, the number (which matches the source code labels) can be used to extract the relevant source files from VSS.

Rebuilding Multi-Solution Systems

While a build is running, you may experience problems compiling locally if you are working on a multi-solution system that uses file reference to reference assemblies on the build server because the build process starts by clearing the Latest folder.

To avoid this issue, you should switch to an isolated development model as described in Consider an Isolated Development Approach in Chapter 4, "Managing Dependencies," particularly when working on multi-solution systems. In this way, you are protected from builds that may run during the day.

Emailing Build Results

Build results should be sent via e-mail to the development team for successful and unsuccessful builds. The e-mail message should include the build log as an attachment to enable developers to help diagnose build failures or identify warnings generated from their code.

To avoid hard coding e-mail aliases in the build script, you should set up distribution lists based on project or solution names.

Packaging the Build

To facilitate the installation of the latest build into the test (or development) environment, the build script can package the output within a Microsoft Installer (MSI) file.

For single solution (or partitioned single solution) systems, you can include a Visual Studio .NET Setup and Deployment project within the solution. This can be used to automatically generate an MSI file as part of the solution build.

Note: You can prevent the Setup and Deployment project from building each time you build the solution on your local workstation by excluding it through the Configuration Manager (on the **Build** menu) within Visual Studio .NET. If you exclude a project from a solution build using this method, you do not affect the source controlled solution file. Changes are maintained within the solution user option file, which is developer specific and not under source control.

Whenever new projects are added to your solution you must remember to update and configure the deployment project to ensure that the output of the new project is included within the MSI file and that any project specific installation steps are performed. This is generally the responsibility of the build coordinator, who should be informed by developers when new projects are added. All developers must be familiar with this process, so make sure it's part of your development process guidelines.

For multi-solution systems, things are once again more complex because a single Setup and Deployment project can be used only to package the output generated by projects in a single solution. In this scenario, the build script can generate an MSI by using third party software or the Windows Installer software development kit (SDK).

Creating Build Script Accounts

You must create a Windows account for the specific purposes of executing the build script on the build server. This account must have access to the shared folder or folders on the VSS server that host the VSS databases.

You must also create a build script user within each of the VSS databases that the build script requires access to. This is used by the script while accessing the VSS database.

More Information

For more information about VSS automation, see "Visual SourceSafe 6.0 Automation" at *http://msdn.microsoft.com/library/default.asp?URL=/library/techart/vssauto.htm*.

6

Working with Visual SourceSafe™

This chapter will help you work with the integrated source control features provided by Microsoft® Visual Studio® .NET in order to perform the following development tasks:

- Creating solutions and projects
- Working on existing solutions and projects
- Adding a new project to an existing solution
- Checking source files into Microsoft Visual SourceSafe™ (VSS)
- Renaming and deleting projects, files and folders

This chapter presents a series of mini-walk through procedures that guide you through a set of common tasks that you are likely to perform on a daily basis.

Note: This chapter describes procedures that apply to Visual SourceSafe version 6.0c.

Visual Studio .NET provides fully integrated VSS support. It's essential that you use Visual Studio .NET when checking projects and files in and out of VSS on a daily basis because Visual Studio .NET is aware of the various file types that constitute different types of .NET projects. It automatically places the appropriate files under source control, while leaving user-specific files untouched. The Visual Studio .NET source control functionality also adds its own specific information to both Visual Studio .NET solution and project files. If you use the VSS Explorer directly, this functionality is bypassed.

Figure 6.1 illustrates a high level overview of the development process and shows four common tasks that you will need to perform when interacting with VSS.

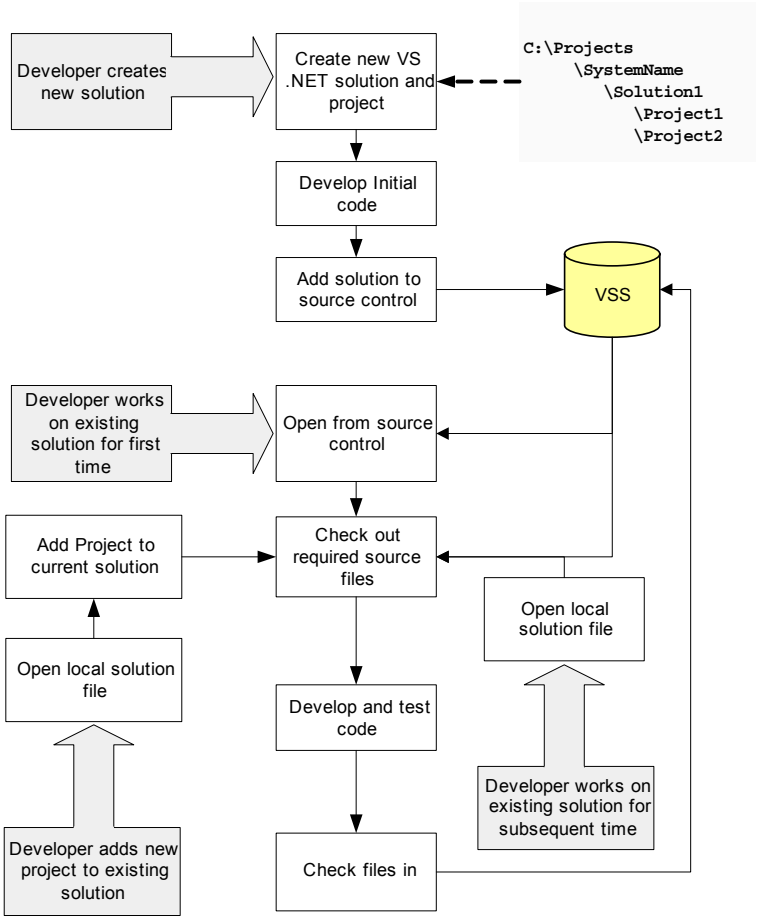


Figure 6.1
Interacting with VSS

As illustrated in Figure 6.1, there are four basic tasks that you regularly perform as a developer: You need to:

- Create a new solution and project and add these to VSS.
- Work on an existing solution for the first time.
- Work on an existing solution that you have worked on before.
- Add a new project to an existing solution.

This chapter explains how you must perform each of these tasks together with some other tasks that you may occasionally need to perform, such as renaming projects, files and folders within source controlled VSS projects and using VSS in a multiple checkout mode.

Creating a New Solution and Project

In this scenario, you want to create a brand new solution that initially contains a single project and then add the solution and project to VSS. The process of creating both Web and non-Web project types and adopting the recommended folder structure is described in Chapter 3, "Structuring Solutions and Projects." The following procedures assume you have created solutions and projects in accordance with the prescribed structure.

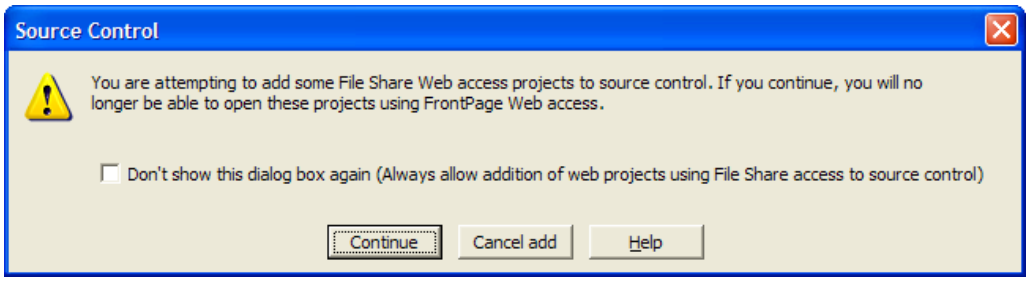
Note: Each project file contains a Globally Unique Identifier (GUID) to uniquely identify that project within a solution. Therefore, you should never copy and rename an existing project file to create a new project, as both will share the same GUID and cannot be distinguished from one another by Visual Studio.NET. Problems will ensue if you include both projects in the same solution.

How to Check In a New Solution to VSS

After you use Visual Studio .NET to create a new solution and project file, and after some initial development, you must check in the solution and project to VSS.

► **To add a new solution to source control**

1. On the **File** menu, point to **Source Control**, and then click **Add Solution to Source Control**. If you are adding an ASP.NET Web application to VSS, the following Source Control message box appears.



2. Select the **Don't show this dialog box again** check box, and then click **Continue** to close this dialog box.
3. You may then be prompted with the VSS login dialog box. This happens only if your VSS administrator has not configured VSS to use automatic network user login. Enter your VSS user name, and then click the **Browse** button to locate the VSS database on the VSS server. Select the Srcsafe.ini file that identifies your development database on your VSS server's file share.

Note: If automatic network user login is configured, you do not see the VSS login dialog box. In this event, if you want to change VSS database, run the VSS Explorer, and then click **Open SourceSafe Database** on the **File** menu. Then browse to the appropriate Srcsafe.ini file.

4. You can then enter a meaningful name for the database—one that represents the name of the system that you are currently working on. Click **Open**, and then click **OK** to connect to the database.
5. In the **Add to SourceSafe Project** dialog box, expand **Projects**, and then select your system name.
6. Click the **Create** button. This creates a VSS project folder specifically for your solution file. The VSS folder has the same name as your Visual Studio .NET solution.
7. Click **OK**.

Important: If you are adding an ASP.NET Web application to VSS, you are prompted to add each Visual Studio .NET project to VSS. For Web applications only (the following two steps are not required for non-Web applications):

a. Expand **Projects**, expand your system name, and then click your solution name. Click **OK**.

b. You are prompted to create a new VSS folder to contain the project. Click **Yes** to create the new folder. This process repeats for all Web projects in the solution.

Your solution, project, and source files are now placed under source control within VSS. Notice that a padlock symbol is displayed in the Solution Explorer window next to all of the files that have been added to VSS. This indicates that they are currently locked within VSS. Your local working files have been assigned the read-only attribute to prevent you from working with them until you check out the files from VSS.

Check Out on Edit

If you subsequently start to edit a locked file, Visual Studio .NET has an automatic check out on edit feature that prompts you to check out the file. If you decide you want to continue and edit the file, click the **Check Out** button in the **Check Out For Edit** dialog box. The file will be automatically checked out for you. You can subsequently right-click the file within Solution Explorer and click **Check In**, to check in the file to VSS.

Working on an Existing Solution for the First Time

In this scenario, another team member has created a solution and added it to VSS. You now need to obtain the solution for the first time in order to work on one of the projects contained within that solution.

► **To obtain an existing solution from VSS**

1. Start Visual Studio .NET.
2. On the **File** menu, point to **Source Control**, and then click **Open From Source Control**. You are prompted with the VSS login dialog box.

3. Enter your VSS user name, and then click the **Browse** button to locate the VSS database on the VSS server. Select the Srcsafe.ini file that identifies your development database on your VSS server's file share. If you have previously connected to this database and selected **Open this database the next time I run Visual SourceSafe**, the database will be pre-selected.
4. Click **OK** to connect to the database. The **Create local project from SourceSafe** dialog box displays.
5. Expand **Projects**, expand your system name folder, and then click the solution you want to obtain. In the **Create a new project in the folder** edit box, enter **\Projects\SystemName\SolutionName**, where **SolutionName** is the name of your particular solution. This ensures that you have the correct local file system structure.
6. Click **OK**. If the local solution folder doesn't exist, you will be prompted to create it. Click **Yes to All** to create the solution folder. For non-Web applications, this also creates the project subfolders.
7. For ASP.NET Web applications, you are prompted by the **Set Project Location** dialog box to enter a working location for the Web application. This allows you to specify a URL which identifies the path to the Web application's virtual root. By default, Visual Studio .NET assumes **http://localhost/<projectname>**. If you click **OK** to accept the default location, the new virtual root is created beneath your default Web site (typically **\Inetpub\wwwroot)**. This is not the recommended location when you are in a team environment, although you do want to create the application locally. To create a virtual root in a project subfolder beneath the solution folder, perform the following steps before you click **OK** to accept the **Set Project Location** dialog box:
   a. Use Microsoft Windows Explorer to create a project subfolder beneath the solution folder which has been created for you at **\Projects\SystemName\SolutionName**. Use the project name to name this subfolder.
   b. Use Windows Explorer or Microsoft Internet Information Services (IIS) to set this folder as an IIS virtual root.
   c. Return to Visual Studio .NET and enter **http://localhost/projectname/** as the working location.

      Important: You must overtype and change the existing Uniform Resource Locator (URL) or the dialog box assumes the address still maps to a folder beneath **\inetpub\wwwroot**).

   d. Click **OK** to close the **Set Project Location** dialog box. This places the project and Web application files in the virtual root, which maps to a subfolder one level beneath your solution folder.

8. The solution, project, and source files are now downloaded to your hard disk. However, note that they remain locked in VSS. The padlock symbol next to each file in Solution Explorer confirms this.
9. You can now either select one or more of the files within Solution Explorer, right-click, and click **Check Out,** or simply start editing the source files because the check out on edit feature of Visual Studio automatically prompts you when a file needs to be checked out.
10. After you complete your local development, you can either check in each file individually or use the **Pending Checkins** window within the integrated development environment (IDE). You may need to click **Pending Checkins** from the **View** menu to display this window.

Note: If you exit the solution without checking in the files, you are not prompted. The files remain checked out in your name.

Working on an Existing Solution for a Subsequent Time

In this scenario, you want to work on an existing source controlled solution that you have worked on at least once before. As a result, your local file system contains a solution and project folder with read-only solution and project files.

► To obtain an existing solution from VSS

1. Start Visual Studio .NET.
2. If the required solution is displayed on the start page, click the solution. Otherwise, open the solution by browsing to the solution (.sln) file on your local file system.
3. To ensure you have the most up-to-date set of project files, you can select the solution within Solution Explorer, right-click it, and then click **Get Latest Version (Recursive)**.

Important: When working on an existing solution that you have previously obtained from VSS, you should open the local solution file. Do not use the **Open From Source Control** menu item in this scenario. If you do so, Visual Studio .NET detects the presence of local solution and project files and you are prompted to overwrite these files. Also, if you have one or more of these files checked out, you are prompted to either replace the local files with versions from VSS, or to leave the local versions untouched. Opening the local solution file is a simpler and less error prone approach.

Adding a New Project to an Existing Solution

In this scenario, you want to add a new project to an existing solution. The following procedure assumes that you have obtained the solution from VSS as described earlier.

► **To add a new project and add it to VSS**

1. On the **File** menu, point to **Add Project**, and then click **New Project**.
2. Select the appropriate project type and template, and then enter a name for the project.
3. For ASP.NET Web projects, you must manually create a project subfolder beneath the solution folder and mark it as an IIS virtual root. For non-Web applications, the new project folder defaults to a location one level beneath the solution folder.
4. Click **OK**. Visual Studio .NET automatically prompts you to check out the solution file (if you haven't already got it checked out) because it needs to add the new project details to that file.
5. Click **Check Out**. The new project is added to the solution and appears in Solution Explorer. Notice that each file in the new project has a tick symbol displayed next to it. This indicates that the files are not locked and are available for edit.
6. After you complete your initial development work with the new project files, they must be checked in to VSS. To add the entire solution (with the new project) back into VSS, right-click the solution in Solution Explorer, and then click **Check In**.
7. The **Check In** dialog box displays all the files currently checked out. This includes the new project file and source files. To check in all the files, click **Check In**.

Checking In Source Files to VSS

After your solution and project are in VSS, you need to check-in only those files that you have updated. To do this you can right-click an individual file within Solution Explorer and click **Check In**. Alternatively, you can see a list of all files that need to be checked in by using the **Pending Checkins** page that is displayed at the bottom of the IDE. You can easily check in multiple files using this page.

Only Check In Files When They Are Ready to Build

Generally, you should check in projects and files to VSS only when you are confident that they are fully unit tested and are unlikely to cause integration problems when built alongside other project updates made by fellow team members.

Important: This may mean that some files are not checked in to VSS for several days. As a result, make sure you have adequate daily backup procedures in place for all of your development workstations.

Note: If you work with a change management system that natively supports the concept of promotion levels, you may prefer to check in files at regular intervals, regardless of their current state of completeness. Change management systems that support promotion levels usually provide an initial "Development" level specifically for this purpose. The build process never extracts source files from this level. Instead, it operates with source files associated with a higher promotion level such as "Integration." Developers or the build coordinator promote files to this level only when they are ready to be integrated with the overall system build.

Renaming and Deleting Files and Folders

On occasion, you may need to rename and/or delete individual files or folders that are within source controlled VSS projects. Rename or deletions that are performed with the Visual Studio .NET IDE are not automatically propagated to VSS and vice-versa. As a result you must use VSS Explorer in conjunction with Visual Studio .NET.

Renaming a File

The following procedure describes how to rename a file in a Visual Studio .NET project.

► **To rename an existing file contained within a Visual Studio .NET project**

1. Use VSS Explorer to ensure that the file you want to rename is not checked out.
2. Use VSS Explorer to rename the file.
3. Use Visual Studio .NET to open the solution that contains the project with the file to rename. The file that you have renamed within VSS appears checked out within Solution Explorer. If you already have the project loaded within Visual Studio .NET, select the project file within Solution Explorer, right-click it, and then click **Get Latest Version (Recursive)**.
4. Select the file within Solution Explorer, right-click it, and then click **Rename**. Rename the file to match the new name within VSS.
5. You will be prompted to check out the project file because it contains a file inventory. In the **Check Out For Edit** dialog box, click **Check Out**.
6. In the Source Control message box, click Continue with Change.
7. The file now appears locked within Solution Explorer. Check in the project file to VSS.

Renaming a Project

When you rename a project you should rename the following items to retain a consistent naming convention, as described in Carefully Consider Naming Conventions in Chapter 3, "Structuring Solutions and Projects":

- The Visual Studio .NET project file
- The local file system folder that contains the project file
- The VSS project folder
- The output assembly name
- The root namespace used for the project

► To rename a project

1. Ensure that the following files are not checked out: the project file, the solution file that contains the project file, or any file within the project.
2. Check out the solution file. Select the solution within the Visual Studio .NET Solution Explorer, right-click it, and then click **Check Out**. Make sure that only the solution file (and no project files) is selected in the **Check Out** dialog box, and then click **Check Out**.
3. You must now unbind the project from the VSS database. Within Visual Studio .NET, point to **Source Control** on the **File** menu, and then click **Change Source Control**.
4. Select the project you want to rename (this may also automatically select the solution) and then click the **Unbind** button. In the resulting **Source Control** message box, click **Unbind**.
5. Click **OK** to close the **Change Source Control** dialog box. The project file (and possibly the solution file) is now no longer bound to the VSS database.
6. Within Solution Explorer, right-click the project you want to rename, and then click **Rename**. Type the new project name.
7. On the **File** menu, click **Save All** to ensure the local solution file is updated.
8. You must now temporarily remove the project from the solution to allow you to rename the local project folder. Right-click the renamed project within Solution Explorer and select **Remove**. In the resulting **Microsoft Development Environment** message box, click **OK**.
9. Use Windows Explorer to rename the local file system project folder to match the renamed project file name.
10. If you are renaming a Web application, use Windows Explorer or IIS to establish the renamed folder as an IIS virtual root. Also, use Notepad to edit the .webinfo file within the project folder and set the **URLPath** attribute to point to the project's new virtual root folder.

11. Return to Visual Studio .NET. Right-click the solution file within Solution Explorer, point to **Add**, and then click **Existing Project**. Browse to the renamed local project folder, select the renamed project file, and then click **Open**.
12. Use VSS Explorer to rename the VSS project folder to match the renamed project.
13. You can now rebind the project to the VSS database. Return to Visual Studio .NET, point to **Source Control** on the **File** menu, and then click **Change Source Control**.
14. If the solution file is currently unbound, select it within the **Change Source Control** dialog box, and then click **Bind**. You may need to log into VSS at this point. Navigate to the solution folder within VSS, select it, and then click **OK**.
15. Select the renamed project within the **Change Source Control** dialog box, and then click **Bind**. Log into VSS if required.
16. Navigate to the renamed project folder within VSS, select it, and then click **OK**.
17. Click **OK** to close the **Change Source Control** dialog box. In the resulting **Source Control** message box, click **OK**.
18. Use the **Pending Checkins** window to check in the solution and renamed project file to VSS.
19. You should now check out the relevant source files and update the root namespace to match the project name.
20. You should also check out the project file and update project properties, including the Assembly Name which controls the name of the output dynamic-link library (DLL) or executable file. Also, for C# projects, update the Default Namespace and for Microsoft Visual Basic® .NET projects, the Root Namespace, because these govern the default namespaces into which new types are added.

► To tidy up the old project files

The old project file, the associated source control metadata file and for Web applications, the Web service dynamic discovery file remain in the renamed VSS project folder and also within the renamed project folder on your local file system. You should tidy up these items by deleting them:

1. Use VSS Explorer to delete the old project file (*.csproj.proj), the old Visual Studio Source Control Project Metadata file (*.csproj.vspcc) and for Web applications, the old Web service dynamic discovery file (*.vsdisco) from the renamed project folder. You may need to refresh the view within VSS Explorer to see the new project files.
2. Use Windows Explorer to delete the old Visual Studio Source Control Project Metadata file (*.csproj.vspcc) and for Web applications, the old Web service dynamic discovery file (*.vsdicso) from the locally renamed file system project folder. Note that these files are read-only on your local file system.
3. For Web applications, use IIS to remove the old virtual root that has now been renamed.

Deleting a File from VSS

This procedure assumes that you have opened a solution from VSS that contains at least one project and the file you want to delete is contained within a project.

► **To delete the file**

1. Ensure that the file you want to delete is not checked out.
2. In Solution Explorer, right-click the file you want to delete, and then click **Check Out**.
3. In the **Check Out** confirmation dialog box, click **Check Out**. If the file is already checked out by someone else, you must wait until the file is available exclusively to you.
4. In Solution Explorer, right-click the file, and then click **Delete**.
5. In the resulting **Microsoft Development Environment** message box that confirms you want the file to be deleted permanently, click **OK**.
6. You will be prompted to check out the project file because it contains a file inventory. In the **Check Out For Edit** dialog box, click **Check Out**.
7. Check in the project file to VSS.
8. Use VSS Explorer to delete the file (which appears checked out) from the VSS database. In the resulting **Microsoft Visual SourceSafe** message boxes, click **Yes** to confirm that the file should be deleted.

Deleting a Project from VSS

The following procedure describes how to delete a project from VSS.

► **To delete a project from VSS**

1. In Solution Explorer, right-click the project you want to delete, and then click **Remove**.
2. In the **Microsoft Development Environment** confirmation message box, click **OK**.
3. You will be prompted to check out the solution file because it contains a project list. In the **Check Out for Edit** dialog box, click **Check Out**.
4. Check in the solution to VSS.
5. Use VSS Explorer to delete the project from the VSS database.
6. Delete the project folder from your local hard disk and for a Web application, use IIS to remove the corresponding virtual root.

Deleting a Solution from VSS

The following procedure describes how to delete a solution from VSS.

► **To delete a solution (and all contained projects) from VSS**

1. In VSS Explorer, make sure none of the following files are checked out: the solution file, any subproject, or the project file.
2. Use VSS Explorer to delete the solution from VSS.
3. Delete the solution from your local hard disk.

Note: You should communicate the fact that you are about to delete a solution to other team members to allow them to unload the solution if they currently have it loaded within Visual Studio .NET. If any team member has the solution loaded (not checked out) when you delete it, they will receive VSS errors if they subsequently attempt to check out a file that is contained within the solution.

Multiple Checkout

Usually, only one user at a time is allowed to check out a file from VSS. However, you can use the VSS Administration tool to configure VSS to allow multiple users to check out the same file simultaneously.

This can be useful in a team environment because it can help to relieve contention on commonly accessed files such as project files. However, if you use this feature, it requires careful coordination between individual team members, and you must be careful to merge changes correctly when you check in the file to VSS.

► **To work with a file in multiple checkout mode**

1. Use Solution Explorer to check out the required file. Visual Studio .NET warns you that the file is checked out by another user. In the resulting **Microsoft Visual SourceSafe** message box, click **Yes**.
2. Make changes to your copy of the file and then compile and unit test your project.
3. Before you check in the file, you should merge any changes made by other developers and locally test those changes in conjunction with your own. To do this, right-click the file in Solution Explorer, and then click **Get Latest Version**. In the resulting **Microsoft Visual SourceSafe** message box, click **Merge**. VSS automatically merges the version from VSS with your local working copy. You can now compile and test the merged file.
4. When your tests are complete, use Visual Studio .NET to check in the file to VSS. If the file has been merged, VSS prompts you to check whether all conflicts have been properly resolved. Click **Yes** in response to the prompt and the file is checked back into VSS.

Checking Out Solution Files

Solution files are particularly difficult to check out simultaneously because they contain a project count which can make the merge process difficult and prone to error. You should generally avoid checking out solution files if another developer already has the file checked out.

Instead, aim to check out a solution file (for example, to add new projects) for short periods at a time to reduce contention on the file. Coordinate with other developers if contention arises and wait for the solution file to become exclusively available before checking it out.

More Information

For more information about multiple check out, see "Check Out Multiple Files" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/guides/html/vstskCheck_Out_Multiple_Files.asp*.

7

Setting Up and Maintaining the Team Environment

This chapter provides guidance to help you set up your team development environment. In summary, the environment building blocks consist of all of the following:

- Developer workstations
- A Microsoft® Visual SourceSafe™ (VSS) server to maintain the source code control database or databases
- A backup server to maintain VSS database backups
- Computers running Microsoft SQL Server™ for development and test databases
- Web servers for development and test Web services
- A build server on which an automated build script is used to generate daily builds

You will also typically require one or more separate test environments; for example, a physically isolated stress test environment that can be used to run scalability and performance tests. The test environments are out of the scope of this document and are not discussed further.

The infrastructure for a team development environment is illustrated in Figure 7.1.

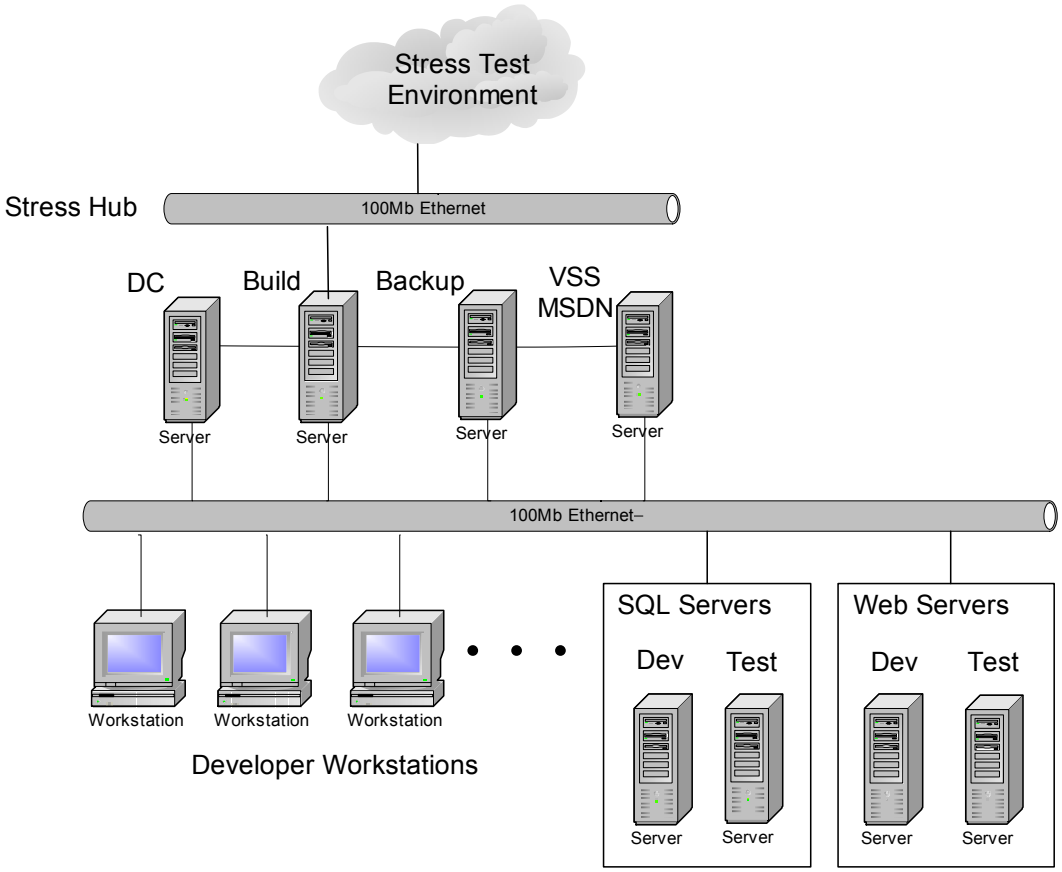


Figure 7.1
Team Development Environment—Infrastructure

Creating a Development Domain

Defining the development environment for your organization is not a simple task. You want to adhere closely to your existing corporate standards for user accounts and workstation builds, but at the same time you know that developers need more privilege on their workstations in order to:

- Install updates or operating system service packs.
- Manage Microsoft Internet Information Services (IIS) locally for application testing and fine-tuning.
- Debug ASP.NET Web applications.

The first step is to decide whether or not to put the development team in the corporate domain or in its own domain. There is no single right or wrong answer. To a large degree, the decision depends upon your corporate security policy. The desirable model is one that your organization is willing to accept and invest in and one that can be maintained properly by existing systems administrators.

Use the following scenarios as guidance to help you during your design process:

Standalone Domain with No Trust Relationship

The following are some of the features of using a standalone domain with no trust relationship:

- This offers the best security model because your corporate users are not able to access the development environment.
- A separate network means that you have flexibility and isolation when you need to perform load testing for capacity planning. You don't need to worry about impacting critical services.
- To use services in the corporate domain, for example e-mail and the intranet, developers can use a virtual private network (VPN) connection to connect to the corporate network.
- You can create a VPN server with Microsoft Windows® 2000 operating system and Routing and Remote Access Services, or you can deploy Microsoft Internet Security and Acceleration (ISA) Server, or third-party solutions.
- This model is more expensive because separate infrastructure servers—domain controllers, DNS, WINS, DHCP, and so on, are required (possibly with additional redundancy configurations).

Standalone Domain with Trust Relationships

The following are some of the features of using a standalone domain with trust relationships:

- This model is more convenient because no VPN solution is necessary.
- If you set up development user accounts within the development domain, the corporate domain has to trust the development domain in order to give access to services, which may be against your corporate security policy. Corporate users cannot access the development environment.
- If you set up development user accounts within the corporate domain, the development domain has to trust the corporate domain, which opens up the potential for hacking.
- With proper planning and pilot-testing, this option should work in most organizations.

Part of the Corporate Domain

The following are some of the features of using the corporate domain:

- This is the most convenient option because there is no domain separation.
- Developers can use all of the existing corporate infrastructure and services.
- Users might not be granted local administrator rights on their workstations under corporate security policy. This causes problems for developers because you must be a local administrator in order to be able to debug an ASP.NET Web application on your development workstation.
- Standard corporate network and security policies could hinder the development process. For example, developers may be not be allowed to install service packs unless they pass the corporate operations model.

The remainder of this section describes the roles of each of the computers in the development environment.

The VSS Server

This server is used to maintain one or more VSS databases that contain the source controlled solutions, project files, and source files for all of your team developments.

The VSS server is also a good candidate to host the Microsoft MSDN® Library for developer reference material. For more information, see article Q271776, "HOWTO: Create an MSDN Library Shared Install Point on the Network," in the Microsoft Knowledge Base at *http://search.support.microsoft.com/*.

Hardware Requirements

The minimum recommended system specification for a VSS server is documented in the VSS readme file. However, in anything but the very smallest team environments, you will want a more powerful system, closer to the recommended specification for a Visual Studio .NET developer workstation. This is documented in the Build Server section later in this chapter.

Remember that the amount of time taken to perform routine VSS database administration tasks is greatly affected by the processor speed and amount of available RAM. You should also aim for a hard disk capacity approximately twice the size of your VSS database.

Software Requirements

The following software is required on the VSS server:

- VSS version 6.0c. This ships with Visual Studio .NET Enterprise Architect and Enterprise Developer versions. It is also available via MSDN on CD/DVD or as a subscriber download.
- Appropriate backup software. The VSS database must be backed up on a regular basis, due to its critical role in the team development process. Use your current backup software and procedures to backup the VSS database.

Build Server

The build server hosts the build script that allows you to generate specific versions of your system. In addition to the build script, the build server also maintains a set of shared folders that contain the latest output from the most recent build operation, together with previous versions, organized by build number. This allows developers to reference the latest inner-system assembly versions.

For more information about how developers must reference assemblies in a team development environment, see Referencing Assemblies in Chapter 4, "Managing Dependencies."

A recommended folder structure for organizing build output is discussed in Chapter 5, "The Build Process." The local folder structure used to maintain Visual Studio .NET solutions, projects, and source files should be consistent across the build server and developer workstations. For more information, see Chapter 3, "Structuring Solutions and Projects."

Hardware Requirements

The following table lists the minimum recommended requirements for Visual Studio .NET that the build server must meet.

Build Server	Minimum Requirements
Computer/Processor	PC with a Pentium II-class processor, 450 MHz (recommended: Pentium III-class, 600 MHz)
Memory	Microsoft Windows NT® 4.0 Workstation—64 MB, Windows NT 4.0 Server—160 MB (recommended: 96 for Workstation, 192 for Server) Windows 2000 Professional—96 MB; Windows 2000 Server—192 MB (recommended: 128MB for Professional, 256 MB for Server) Windows XP Professional—160 MB (recommended: 192 MB)
Available Hard Disk Space	600 MB on system drive, 3 GB installation drive
Drive	CD-ROM drive
Display	800 x 600, 256 colors (recommended: high color 16-bit)
Operating System	Windows 2000, Windows XP, and Windows NT 4.0
Peripherals	CD-ROM or DVD-ROM Drive. Microsoft mouse or compatible pointing device

Software Requirements

The following software is required on the build server:

- Visual Studio .NET
- Visual SourceSafe 6.0c (client)
- Any additional resource required by the system, such as IIS or Message Queuing.

Developer Workstations

Individual developer workstations must be set up with a consistent file system folder structure and one that matches the build server structure—particularly for the folders that contain Visual Studio .NET solutions and projects.

Use Imaging to Create Developer Workstations

To save time with the configuration of multiple identical development workstations, after you install all of the required applications and tools, consider using Sysprep.exe to create an image of the workstation. You can then deploy the workstation image to other computers with third-party disk-imaging software. The advantage of using Sysprep.exe is that it includes a mini-setup wizard, which detects the actual hardware installed in the new workstation, and can therefore handle hardware inconsistencies, such as different video adapters or network adapters.

It is recommended that you store all user data (including solutions and projects) on a separate partition or physical disk. This way, when there is a major update to the image, you can re-install the image over the existing drive C. You can deploy minor updates and additional software through Group Policy in Windows 2000 Active Directory Services.

Hardware Requirements

Workstations must meet the minimum recommended requirements for Visual Studio .NET. As a result, developer workstations have the same hardware requirements as the build server, as discussed in the Build Server section earlier in this chapter.

Software Requirements

The following software is required on development workstations:

- Visual Studio .NET
- Visual SourceSafe 6.0c client components
- SQL Server 2000 Developer Edition
- MSDN Library (Client)

Note: If your development workstations run Windows XP, make sure IIS is installed to allow local Web development. A default installation of Windows XP doesn't result in the installation of IIS.

Visual Studio Enterprise Templates

You can use Visual Studio Enterprise Templates to help encourage good development procedures and standard practices across development projects. They are particularly useful when applied to large scale distributed system developments in situations where multiple projects will be constructed using a common application architecture.

Enterprise Templates provide architects with three capabilities they can use to help developers be more successful building applications on the .NET platform:

1. Package an architect supplied framework of existing code and components in a way that integrates it into Visual Studio .NET, providing developers with a familiar design experience. Architects can use this capability to provide application infrastructure that should be common and consistent across projects, in areas such as security, error handling, and application instrumentation.
2. Capture architecture and implementation rules and apply them as "active" guidance while developers are building the application. Architects can use these rules to filter out options that are not relevant to the project at hand. These rules can provide immediate reminders when developers add items, references, interfaces, class members, or property values to inappropriate parts of the application, helping them avoid common problems.
3. Help architects provide developers with relevant Help topics on a "just-in-time" basis to reduce the need to read mountains of documentation before they can begin to write code.

For more information about Enterprise Templates, see the "Enterprise Templates for Distributed Applications" topic in the Visual Studio .NET section of the MSDN library.

Backup Server

The backup server is used to maintain a backup of the VSS database. You should backup your VSS databases on a regular (for example, daily) basis.

SQL Servers

SQL Server 2000 Enterprise Edition is installed on these servers. They are used to maintain the individual databases required by the systems that are currently under development, support, and maintenance.

Note that an individual server may be used to host several database instances, for example an integration test and a user test database. This depends to a large degree on the size of your databases and the capacity of your servers.

For further information about how database versioning issues should be addressed and how developers are recommended to connect to databases on this server, see Referencing Databases in Chapter 4, "Managing Dependencies."

Web Server

Web servers host XML Web services that are currently under development. While the development teams responsible for Web services develop them on their local workstations using local instances of IIS, the Web server allows the services to be published centrally, for other developers or development teams to access from client projects. For more information about working with Web services, see Referencing Web Services in Chapter 4, "Managing Dependencies."

Web servers within the team development environment are also used to host Web applications to support integration, system testing, and user testing.

Operating System Requirements

The operating system requirements for a Web server that hosts ASP.NET Web applications and Web services developed with Visual Studio .NET are the following:

- Windows 2000, Windows XP, or Windows .NET Server.
- It is also recommended that you install the Web server on a computer formatted with the NTFS file system.
- The Web server must be running IIS 5.0 or 6.0.

For further details, see **setup\WebServer.htm** on the Visual Studio .NET CD1 or DVD.

Installing and Administering VSS

VSS 6.0c is supplied on a separate disc within the Visual Studio .NET Enterprise box and is not installed by default. You must install VSS on the server and then install the VSS client components on each computer that requires access to a VSS database—typically the development workstations and build server.

The following sections outline guidelines you should follow when installing VSS on your server.

Create a Shared Database on the Server

Run the Setup program on your server and select the shared database installation option. This installs administration tools and the network setup program (Netsetup.exe) that allows VSS users to install the client software.

Share the VSS Installation Folder with Read Access

Share out your VSS installation folder with read access. This allows developers to run Netsetup.exe from the VSS folder. Note that the default VSS installation folder is \Program Files\Microsoft Visual Studio\Vss.

Create at Least One New Database for your .NET Development Project

Use the VSS Administration tool to create a new database for your .NET development project. It is recommended that you create a new database (rather than use the default one) so that you can secure the default database and administration tools (located in the Vss\Win32 folder) separately. This allows you to restrict who has access to the administration tools. Most users should be allowed to connect only to the new database and should be prevented from accessing the administration tools.

Don't create new databases beneath \Program Files. Create VSS databases beneath a separate shared folder, such as \VSS Databases.

Consider Creating Additional VSS Databases

Consider creating additional VSS databases (on the same server but within separate file shares) when the size of a single database approaches 5 GB in size. With this size of database, the administration tools begin to take an excessive amount of time (several hours) to perform their work.

You should also consider using separate databases for completely independent development projects. If you split your Visual Studio .NET solutions and projects into separate VSS databases, you benefit from the following advantages:

- It allows you to improve security by more precisely defining which users can access which projects. A user that has access to a single VSS database is able to view and manipulate all of the VSS projects it contains.
- Routine maintenance does not impact all projects. Maintenance such as backup requires that the database be locked, which prevents regular (non-administrators) from accessing the database.
- If you experience corruption within a database, you need to take it offline to rectify the corruption. With separate databases, you impact only the developers who access the affected database and you do not stop the workflow of all of your developers.
- When a development project ends, you can easily archive the project's database.

Share the Database Folder and Establish the Appropriate Permissions

Share the new database folder and grant your VSS users the full control permission. Your VSS users include developers and an account used by the automated build process.

Create a Windows group for all users who are allowed access to the VSS development database. Allow members of this group to access the development database, by securing your database folder.

If you don't want to grant users the full control permission and require tighter security control, see article Q131022, "INFO: Required Network Rights for the SourceSafe Directories," in the Microsoft Knowledge Base at *http://search.support.microsoft.com/*.

Consider Using VSS Project Security

Consider using the VSS Administration tool to enable project security and remove the **Destroy** right for all developers. This prevents developers from permanently destroying projects within VSS. The VSS administrator is able to recover any project that a developer may delete.

With project security enabled, the administrator must purge the database of all deleted items as part of regular maintenance, although this can be automated by a maintenance script.

For more information about default and project security, see "Security Access Rights" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/guides/html/vstskSecurity_Access_Rights.asp*.

Add User Accounts for Developers and the Build Script

Use the VSS Administration tool to open the new database or databases and add the required users with appropriate passwords.

Remember to set up one VSS user for each developer and another for the build process. If you are in a domain environment, it is recommended that you make the user name the same as the domain name. Be sure the network name is used for user login. On the **Tools** menu, click **Options**, and then select the **Use network name for automatic user login** check box.

Restrict Access to Administration Tools

Create a Windows VSS administration group and use this to restrict access to the Vss\Win32 folder. This folder contains the VSS administration tools and you should restrict developer access to these tools.

Finding and Repairing Data Corruption

VSS ships with the Analyze tool (located in the Vss\Win32 folder) that can be used to check for and correct data corruption problems. You should use Analyze as frequently as is practical—once a week is recommended, or at a minimum, once a month. Analyze checks all the files in the VSS Data directory for corruption or disconnected links and often repairs these with proper switch settings.

There are a number of reasons why you may experience database corruption. The most common reasons include:

- General network problems, such as using an unreliable remote connection and dropping the connection mid-way through a file check in.
- Running out of disk space.

Analyze can be slow to run although this depends on the content and structure of the database, such as the amount of sharing and branching, and the total number of files. For best performance, all users should log off VSS before you run Analyze.exe. If you use the -F switch to repair problems, users must log off.

Note: Due to the amount of file I/O required, you can dramatically improve performance by running Analyze locally (that is, on the server) rather than across the network. You should also ensure that no virus protection software is currently running.

More Information

For a full list of command line switches accepted by Analyze.exe, see "Analyze" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/guides/html/vsgrfSS_ANALYZE.asp*.

Consider Installing Fault Tolerant Storage

Redundant array of independent disks (RAID) technology minimizes data loss due to problems accessing a hard disk. RAID is a fault-tolerant disk configuration in which part of the physical storage capacity contains redundant information about the data stored on the disks. You can regenerate the data using this redundant information if one of the disks or the access path to it fails, or if a sector on the disk cannot be read.

- Install RAID Level 1 for the operating system and logs.
- Install RAID Level 5 or RAID 0+1 for data such as your VSS database.

More Information

For information about how to make your server more reliable, see the following resources:

- "Increasing System Reliability and Availability with Windows 2000" at *http://www.microsoft.com/windows2000/techinfo/howitworks/management/relavail.asp*
- "Deploying Windows NT Server for High Availability" at *http://www.microsoft.com/ntserver/techresources/deployment/ntserver/highavail1.asp*
- "HOW TO: Set up SourceSafe for Replication under Windows NT" at *http://support.microsoft.com/support/kb/articles/q128/6/36.asp*

Installing VSS on Clients

To install the VSS client, simply run Netsetup.exe from the network share on any computer that requires VSS access (for example, development workstations and the build server). This enables the integrated source control features within the Visual Studio .NET integrated development environment (IDE). This also enables the **Source Control** menu on the **File** menu.

Consider Using VSS Shadow Directories

A VSS shadow is a directory on a central server that mirrors the latest contents of a VSS project. Whenever a developer updates a file within the VSS project, it is automatically copied to the shadow directory.

If you need users who do not have access to VSS to be able to view source code, consider using shadow directories. For example, you may want members of your test team to be able to view source code to assist with the testing process.

More Information

For a full list of command line switches accepted by Analyze.exe, see "Analyze" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/guides/html/vsgrfSS_ANALYZE.asp*.

For general VSS best practices, see "Microsoft Visual SourceSafe Best Practices" at *http://msdn.microsoft.com/library/?url=/library/en-us/dnvss/html/vssbest.asp?frame=true*.

Microsoft occasionally updates the Analyze tool to include more checking or to improve performance. For the latest version of Analyze, visit the Microsoft Visual SourceSafe Web site located at *http://msdn.microsoft.com/ssafe/default.asp*.

Appendix

BuildIt—An Automated Build Tool for Visual Studio .NET

Summary: BuildIt is a Microsoft® .NET console application that automates the build process outlined in the *patterns & practices* article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN® Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*. BuildIt is designed, developed, and tested by Sapient Corporation and it is reviewed by Microsoft, including team members of Microsoft *patterns & practices* and Visual Studio® .NET development system.

Using BuildIt:

- Eliminates the time required to create, test, and maintain a custom build script.
- Makes a team's build process more repeatable and consistent.

BuildIt is designed to jump-start the build process used for development of .NET distributed applications. The downloadable program provides full source code and comprehensive documentation for Microsoft Visual C#® development tool and Microsoft Visual Basic® .NET development system.

Note: BuildIt currently supports building solutions that contain setup projects developed with Visual Basic .NET, Visual C#, and Visual Studio .NET. It has not been tested with projects written in other .NET languages or with other setup projects (for example, setup projects from Wise or InstallShield).

While BuildIt has undergone testing and reviews and is considered a robust code set, the code and documentation is provided "as-is" for you to use and extend.

This document includes the following sections:

- Downloading and Installing BuildIt
- Testing Your Installation
- User's Guide
- Deployment and Operations

- Known Issues
- Design and Implementation
- Class Reference
- Frequently Asked Questions
- Appendix Summary

Downloading and Installing BuildIt

To download BuildIt, go to *http://microsoft.com/downloads/details.aspx?FamilyId=B32497B0-77F7-4831-9C55-58BF3962163E&displaylang=en*.

The installation process creates a **Microsoft Application Blocks for .NET** submenu on your **Programs** menu. On the **Microsoft Application Blocks for .NET** submenu, there is a **Sapient BuildIt** submenu that includes options to launch the BuildIt Visual Studio .NET solution.

Testing Your Installation

The following steps to build two sample solutions found in Program Files\BuildIt\Code**BuildItTest** help verify that BuildIt is working properly.

► To add items to source control

1. In Visual SourceSafe® (VSS) version control system, create a new project named **BuildItTest**.
2. Set the working folder on **$/BuildItTest** to Program Files\BuildIt\Code**BuildItTest**.
3. Add assembly version files to VSS:
   a. Add Program Files\BuildIt\Code\BuildItTest**AssemblyVersionInfo.cs** to **$/BuildItTest**.
   b. Repeat for **AssemblyVersionInfo.vb**.
4. Create a virtual directory for the Web project (Project2):
   a. Browse to Program Files\BuildIt\Code\BuildItTest\Solution2**Project2**.
   b. Right-click **Project2**, and then click **Properties**.
   c. Click the **Web Sharing** tab, and then click **Share this folder**.
   d. Accept the default alias, and then click **OK** to close the **Edit Alias** dialog box.
   e. Click **OK** to close the **Properties** dialog box.

5. Add solutions to VSS:
   a. Open Program Files\BuildIt\Code\BuildItTest\Solution1**Solution1.sln**.
   b. In Solution Explorer, right-click **Solution1**, and then click **Add Solution to Source Control**.
   c. Browse to **$/BuildItTest**, and then click **OK** (click **Yes** to create the VSS project).
   d. If prompted to specify the location for **Project1**, browse to $/BuildItTest /**Solution1**, and then click **OK**.
   e. Repeat Steps a–d for **Solution2**.
6. Share assembly version files in VSS:
   a. Share $/BuildItTest/**AssemblyVersionInfo.cs** to $/BuildItTest/Solution1 /**Project1**.
   b. Share $/BuildItTest/**AssemblyVersionInfo.vb** to $/BuildItTest/Solution2 /**Project2**.

► To add assembly version files to projects

1. Add AssemblyVersionInfo.cs to Project1 found in Solution1:
   a. Open Program Files\BuildIt\Code\BuildItTest\Solution1**Solution1.sln**.
   b. In Solution Explorer, right-click **Project1**, point to **Add**, and then click **Add Existing Item**.
   c. Browse to Program Files\BuildIt\Code\BuildItTest\Solution1\Project1 **AssemblyVersionInfo.cs**, and then click **Open**.
   d. In the **Check Out for Edit** dialog box, click **Check Out** to check out **Project1.csproj**.
   e. Open **AssemblyVersionInfo.cs** and note the version number (1.0.0.*).
   f. Press CTRL+SHIFT+B to build the solution.
   g. In Solution Explorer, right-click **Project1**, and then click **Check In Now (Recursive)**.
   h. Repeat Steps a–g to add AssemblyVersionInfo.vb to Project2 found in Solution2.
2. Update configuration settings:
   a. Open Program Files\BuildIt\Code\BuildItTest**BuildIt.exe.config**.
   b. Update the **sourceControl** settings.
3. Deploy BuildIt:
   a. Open Program Files\BuildIt\Code**BuildIt.sln**.
   b. Press CTRL+SHIFT+B to build the solution (make sure you build a **Release** version).
   c. Copy the .exe file and all .dll files from Program Files\BuildIt\Code \BuildIt\bin**Release** to Program Files\BuildIt\Code**BuildItTest**.

4. Execute BuildIt and verify:
   a. Open Program Files\BuildIt\Code\BuildItTest**BuildNumber.xml** and note that the next build number is 1.
   b. Open Program Files\BuildIt\Code\BuildItTest**BuildIt.bat** to review the BuildIt command line.
   c. Run **BuildIt.bat** to build the solutions.
   d. Open Program Files\BuildIt\Code\BuildItTest**BuildIt.log** to check for errors.
   e. Open Program Files\BuildIt\Code\BuildItTest**BuildReport.log** to review results of the build.
   f. Open **BuildNumber.xml** and note that the next build number was incremented to 2.
   g. Open **AssemblyVersionInfo.cs** and **AssemblyVersionInfo.vb** and note that the version numbers were updated from 1.0.0.* to 1.0.1.*.
   h. Browse to Program Files\BuildIt\Code\BuildItTest\Archive**1** to see that the generated assemblies were archived to a folder named after the build number.
   i. Browse to Program Files\BuildIt\Code\BuildItTest**Latest** to see that the generated assemblies were copied to the latest folder.
   j. Show history on **$/BuildItTest** to see that the VSS project was labeled with the build number.

User's Guide

The following topics provide more information about using BuildIt:

- Maintaining Build Numbers
- Building Solutions
- Reviewing the Build Report
- Rebuilding a Solution
- Archiving Builds
- Emailing Build Results
- Versioning Assemblies

Maintaining Build Numbers

BuildIt uses build numbers to label solutions that are source controlled in VSS and to update assembly version numbers (if the appropriate build option is enabled). At the end of each successful build, BuildIt generates a new build number by incrementing the current build number by one (if the build fails, the build number is not incremented). The new build number is then saved to an XML file whose name and location is defined by the **buildNumberFilePath** attribute contained in the **BuildIt.exe.config** file.

The following example shows the contents of a build number XML file called BuildNumber.xml.

```
<buildNumber>1000</buildNumber>
```

Note: The build number XML file must be created and initialized with a starting build number before running BuildIt. Also note that build numbers must be positive integers.

Building Solutions

BuildIt can be configured to build a single solution or multiple solutions. For multi-solution systems that include cross-solution file references, the solutions must be listed in the correct order. For example, if **Solution2** depends on **Solution1**, make sure that **Solution1** gets built first by listing it before **Solution2**.

The following sample from **BuildIt.exe.config** shows how to configure the build tool to build a multi-solution system:

```
<solutions latestRootFolderFullName="c:\System"
 buildNumberFilePath="c:\System\BuildNumber.xml">
  <solution path="c:\System\Solution1\Solution1.sln"/>
  <solution path="c:\System\Solution2\Solution2.sln"/>
</solutions>
```

Reviewing the Build Report

BuildIt generates a build report called **BuildReport.log** in the working folder at the end of each build. The build report can be viewed with any text editor. The build report contains summary information along with a list of solutions that succeeded and a list of those that failed. If any solutions fail to build, Visual Studio .NET can be used to manually build those solutions to find out exactly why they failed.

```
==========================================
Build Report Generated by BuildIt
==========================================
Build Number:      8016
Build Start:       9/12/2002 6:50:09 PM
Build End:     9/12/2002 6:50:43 PM
Build Duration:    1 minute(s)
Archive Folder:    c:\System\Archive\8016\
Latest Folder:     c:\System\Latest\
Build Results:     1 succeeded, 0 failed

==========================================
Succeeded
==========================================
c:\System\Solution1\Solution1.sln

==========================================
Failed
==========================================
```

Rebuilding a Solution

A **rebuild** is required if a build break occurs during a build. You must complete the following steps before rebuilding the solution.

► **To rebuild after a build break**

1. If the fix requires a code change, manually check out the necessary files from VSS to the build computer.

   Note: Use the same VSS credentials specified in the build tool configuration file when checking out files. This helps identify the files that were modified to fix a break.

2. Make the code change, recompile using Visual Studio .NET, and then test the recompiled files.
3. After the problem is resolved, check in the modified files.
4. Manually update the label on each checked-in file with the label that was generated by the previous build, as follows:
   a. Right-click the file name, and then click **Show History**.
   b. In the **History Options** dialog box, click **OK**.
   c. Click the version of the file whose label you want to update, and then click **Details**.
   d. Update the **Label** field with the label used during the build (for example, "Build 100").
5. Manually run BuildIt with the **/rebuild** option.

Archiving Builds

BuildIt can be configured to archive previous builds to a specified location, as described in the "Consider Maintaining Previous Builds" section of the "Team Development with Visual Studio .NET and Visual SourceSafe" article in the MSDN Library at *http://msdn.microsoft.com/library/en-us/dnbda/html/tdlg_ch5.asp*.

When the **archiveBuild** build option is enabled, the build tool copies generated assemblies to a folder named after the current build number located beneath the archive root folder specified by the **archiveRootFolderFullName** attribute in the configuration file.

By default, BuildIt archives output generated from only Visual C# and Visual Basic .NET projects. To archive output from other projects such as Visual Studio .NET setup projects, add **folder** elements to the **additionalFoldersToArchive** element.

The following sample from **BuildIt.exe.config** shows how to enable this option.

```
<options>
  <archiveBuild mode="on" archiveRootFolderFullName="c:\System\Archive">
    <additionalFoldersToArchive>
      <folder fullName=" c:\System\Solution1\Setup1\Debug"
       destFolderName="Debug\Setup1" />
      <folder fullName=" c:\System\Solution1\Setup1\Release"
       destFolderName="Release\Setup1" />
    </additionalFoldersToArchive>
  </archiveBuild>
</options>
```

Emailing Build Results

BuildIt can be configured to send the build results, by way of an e-mail message, to a specified e-mail address, as described in the "Emailing Build Results" section of the "Team Development with Visual Studio .NET and Visual SourceSafe" article at *http://msdn.microsoft.com/library/en-us/dnbda/html/tdlg_ch5.asp*.

When the **sendBuildReport** build option is enabled, the build tool sends a generated e-mail message that contains a build report called **BuildReport.log** to an e-mail address specified by the **toAddress** attribute.

The following sample from **BuildIt.exe.config** shows how to enable this option:

```
<options>
  <sendBuildReport mode="on" smtpServer="255.255.255.255"
   toAddress="you@yourcompany.com"/>
</options>
```

Versioning Assemblies

BuildIt can be configured to update an assembly's version with the current build number stored in an XML file specified by the **buildNumberFilePath** attribute as follows:

```
<solutions buildNumberFilePath="c:\System\BuildNumber.xml">
```

An assembly's version is typically specified in the **AssemblyVersion** attribute (which is defined within the AssemblyInfo.cs or AssemblyInfo.vb file). However, when the build tool is configured to update an assembly's version, the **AssemblyVersion** attribute should be defined in a separate file (for example, **AssemblyVersionInfo.cs** or **AssemblyVersionInfo.vb**) that is shared in VSS across all of the .NET projects in the solution. This allows BuildIt to update one file and have the change propagate to all of the projects in the solution. For more information, see the "Consider Centralizing Assembly Version Numbers" section of the "Team Development with Visual Studio .NET and Visual SourceSafe" article at *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

Note: Two example files called **AssemblyVersionInfo.cs** and **AssemblyVersionInfo.vb** are copied to the **BuildItTest** directory when BuildIt is installed.

The version number is physically represented as a four part number separated with periods, as shown in the following code sample.

```
<major version>.<minor version>.<build number>.<revision>
```

When the **updateAssemblyVersion** build option is enabled, the build tool updates the **build number** segment of the **AssemblyVersion** attribute with the current build number.

The following sample from **BuildIt.exe.config** shows how to enable this option.

```
<options>
  <updateAssemblyVersion mode="on"
   csAssemblyVersionVSSPath="$/System/AssemblyVersionInfo.cs"
   vbAssemblyVersionVSSPath="$/System/AssemblyVersionInfo.vb" />
</options>
```

Deployment and Operations

This section includes the following administrative topics:

- Deploying BuildIt
- Configuring BuildIt
- Securing BuildIt
- Troubleshooting BuildIt

Deploying BuildIt

When deploying any application, it's important to identify any dependencies that exist. BuildIt, which is implemented as a single assembly named BuildIt.exe, has the following dependencies:

- **BuildIt.exe.config** for configuration settings
- **BuildNumber.xml** for maintaining the next build number

Note: The name and location of the build number XML file is defined by the **buildNumberFilePath** attribute in the BuildIt.exe.config file. Also note that the build number XML file must be created and initialized with the starting build number before running BuildIt. For more information, see the "Maintaining Build Numbers" section at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*.

- **Microsoft.ApplicationBlocks.ExceptionManagent.dll** and **Microsoft.ApplicationBlock.ExceptionManagent.Interfaces.dll** for exception management
- **Interop.SourceSafeTypeLib.dll** for interoperating with the VSS API (SSAPI.dll)
- **Visual SourceSafe 6.0c** client programs for accessing a remote VSS database
- **Visual Studio .NET** for building .NET solutions

Armed with this information, a deployment method can be chosen based on the needs of the development team. For BuildIt, there are two deployment methods to consider: XCOPY or Windows Installer deployment. The following sections describe these options in more detail.

Deploying with XCOPY

Deploying BuildIt with the XCOPY command is the easiest method; however, it requires more knowledge from the person deploying the tool. To deploy with this method, use XCOPY to copy the required assemblies, along with the application configuration file (BuildIt.exe.config), to a directory on the target computer. Next, invoke the **ExceptionManagerInstaller** class using the Installutil.exe system utility to create the event sources required by the default exception publisher when it writes to the Windows Event Log. For example, use the following command-line:

```
Installutil.exe Microsoft.ApplicationBlocks.ExceptionManagement.dll
```

For more information about the Exception Management Application Block, see "Microsoft Application Blocks for .NET" in the MSDN library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*.

Deploying with Windows Installer

Deploying BuildIt with Windows Installer requires more work up front than XCOPY, but it requires little knowledge from the person deploying the tool. To deploy using a Visual Studio.NET Setup and Deployment project, add the **BuildIt**, **Exception Management**, and **Exception Management Interfaces** project output to the application folder along with the BuildIt.exe.config file. Next, add the Exception Management project output as a custom action to ensure the **ExceptionManagerInstaller** class is instantiated at installation time.

Configuring BuildIt

The behavior of the build tool is controlled using an application configuration file. Application configuration files are XML-based documents stored in the root directory of the application folder hierarchy. The application configuration file name takes the form *applicationname.applicationextension*.config. For the build tool, the application configuration file is called BuildIt.exe.config.

This section includes the following topics about configuring BuildIt:

- Setting Build Options
- Configuring Tracing
- Configuring Exception Management
- Running with Multiple Versions of Visual Studio .NET

Note: When the configuration file is modified, the changes do not take effect until BuildIt is executed again.

Setting Build Options

You can enable or disable any build option by setting its **mode** attribute to **on/off**, as shown in the following code sample:

```
<options>
  <sendBuildReport mode="on" smtpServer="255.255.255.255"
   toAddress="you@company.com"/>
</options>
```

Note: If the mode attribute is omitted or has any value other than **on/off**, an exception is raised. Also note that the mode attribute values are **not** case sensitive.

Configuring Tracing

BuildIt uses the **Trace** class in the **System.Diagnostics** namespace to generate trace output to the console and to a file called BuildIt.log in the working directory. The following is an example of the trace output.

```
Validating command-line arguments
Reading BuildIt settings from configuration file
Starting build 100 at 11/1/2002 1:57:53 PM
Updating assembly version file $/System/AssemblyVersionInfo.cs
Labelling source
Getting source from label
Backing up latest folder
Building Debug version of solution C:\System\Solution1\Solution1.sln
Building Release version of solution C:\System\Solution1\Solution1.sln
Building Debug version of solution C:\System\Solution2\Solution2.sln
Building Release version of solution C:\System\Solution2\Solution2.sln
2 solution(s) succeeded, 0 failed
Deleting latest backup folder
Archiving additional folders
Generating build report
Not sending build report because the option was not enabled
Ending build at 11/1/2002 1:58:40 PM - 0 minute(s) to complete
```

Tracing can be enabled and disabled using a TraceSwitch named **traceLevel** defined in the BuildIt.exe.config file. Tracing is disabled by setting the TraceSwitch to **0** and enabled by setting the switch to **3**. Setting the TraceSwitch to **4** enables more detailed trace output (verbose mode).

The following code sample shows how to enable tracing.

```
<system.diagnostics>
  <switches>
    <add name="traceLevel" value="3"/>
  </switches>
</system.diagnostics>
```

In addition, BuildIt can be configured to append trace output to the trace file (if one already exists) or it can be configured to overwrite the trace file. The recommended setting is to overwrite the trace file to prevent it from becoming too large. The following code sample shows how to do this.

```
<appSettings>
  <add key="appendTraceOutput" value="off" />
</appSettings>
```

Note: If the **appendTraceOutput** key is omitted, BuildIt overwrites the trace file by default.

Configuring Exception Management

BuildIt uses Microsoft's Exception Management Application Block, which can be configured with XML settings in the configuration file. If the settings are omitted, the Exception Management Application Block publishes exceptions to the application log by default. However, additional publishers can be specified.

At the time of this writing, the configuration file does not contain Exception Management settings. Therefore, all exceptions are published to the application log, which can then be view using the Event Viewer. For more information, see "Exception Management Application Block Overview" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*.

Running with Multiple Versions of Visual Studio .NET

By default, BuildIt uses the latest installed version of Visual Studio .NET to build solutions. However, if multiple versions are installed on the same computer, developers may want BuildIt to use a specific version. To force BuildIt to use a specific version of Visual Studio .NET, include a **visualStudioProgID** key in the **appSettings** section of the configuration file. The following code sample shows how to do this.

```
<appSettings>
  <add key="visualStudioProgID" value="VisualStudio.Solution.7" />
</appSettings>
```

Note: If the **visualStudioProgID** key is omitted, BuildIt uses the latest installed version of Visual Studio .NET by default.

Securing BuildIt

As with any application, you must ensure that all sensitive data and resources are protected from unauthorized access. With respect to BuildIt, there are two main areas to consider:

- Preventing Unauthorized Access to Application Configuration Files
- Access Permissions Required by BuildIt

Preventing Unauthorized Access to Configuration Files

Because the build tool configuration file contains sensitive data such as the VSS user name and password, you should ensure that only authorized users can view or change those settings. You can secure the configuration file by using Windows NTFS file permissions.

Access Permissions Required by BuildIt

BuildIt uses Microsoft's Exception Management Application Block to publish exceptions to the event log. In addition, the BuildIt reads/writes to the file system. Therefore, you need to ensure that the security principal used by BuildIt has appropriate permissions to conduct these operations.

Troubleshooting BuildIt

There are two mechanisms available to help developers troubleshoot the build tool: tracing and exception management. When tracing is enabled, BuildIt generates a trace file in the working folder called BuildIt.log that can be used to determine which steps were performed by the build tool. This is useful when trying to determine why the build tool is not behaving as expected.

When an exception occurs, BuildIt uses Microsoft's Exception Management Application Block to publish exceptions to the application log. Developers can then use the Event Viewer to view detailed information about the exception.

When used together, the two mechanisms provide information that can be used to troubleshoot the build tool.

Known Issues

BuildIt is very effective in most cases- but there are a few situations in which it has some known issues. Table 1 lists known issues with BuildIt.

Table 1: Known issues with BuildIt

Issue	Description	Resolution
BuildIt throws an error when building two or more solutions that have projects with the same name.	An error occurs when BuildIt tries to copy project output to the latest folder if that output already exists. This can happen if BuildIt encounters two projects with the same name, even though they reside in different solutions. It can also happen if BuildIt encounters the same solution a second time during a single run of the tool.	Make sure all projects are uniquely named across your system and make sure the App.config file does not contain duplicate solution entries.
Set Project Location dialog box appears during build.	The **Set Project Location** dialog box appears during the build, prompting the user to set the location for the working copy of a Web project. This happens when trying to load a Web project that is not mapped to a corresponding virtual directory.	To prevent the dialog box from appearing while BuildIt is running, make sure all Web projects are mapped to virtual directories before running BuildIt. One easy way to check is to manually open each solution in Visual Studio .NET before running BuildIt. If the dialog box does appear while BuildIt is running, simply respond to the dialog box by setting the location for the working copy of the Web project. Visual Studio .NET then creates the virtual directory automatically.

Design and Implementation

This section includes the following design and implementation topics:

- Problem Description
- Design Goals
- Solution Description
- Enhancements

Problem Description

BuildIt is designed to solve the following problems:

- Developers lose precious development time creating, testing, and maintaining custom build scripts that are often not reusable from one project to the next.
- Developers are often under tight timeframes, leaving little time for thorough testing and documentation of their build scripts.

Design Goals

The design goals of BuildIt are:

- The solution must adhere to the steps outlined in the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.
- The solution must be well-documented, maintainable, and easy to use.
- The solution must be flexible; it should be configurable without needing to re-code and re-compile.
- The solution must be robust; it must handle any exceptions that occur during the build process.

Solution Description

BuildIt is a console application that consists of a number of classes that work together to automate the build process. For example:

- **BuildInitializer.** Validates command-line options, retrieves build tool settings by way of the BuildItSectionHandler and initiates the build process by way of the BuildManager.
- **BuildItSectionHandler.** Retrieves settings from the tool's configuration file. These settings are used to alter runtime behavior.
- **BuildManager.** Orchestrates the build process.
- **SourceSafeHelper.** Exposes VSS operations (for example, Check-in and Checkout) used during the build process.
- **BuildItResourceManager.** Provides type-safe access to error messages stored in a resource file.
- **BuildItCommandLineArgs.** Encapsulates the command-line arguments supported by BuildIt.

Figure 1 illustrates how these classes work together.

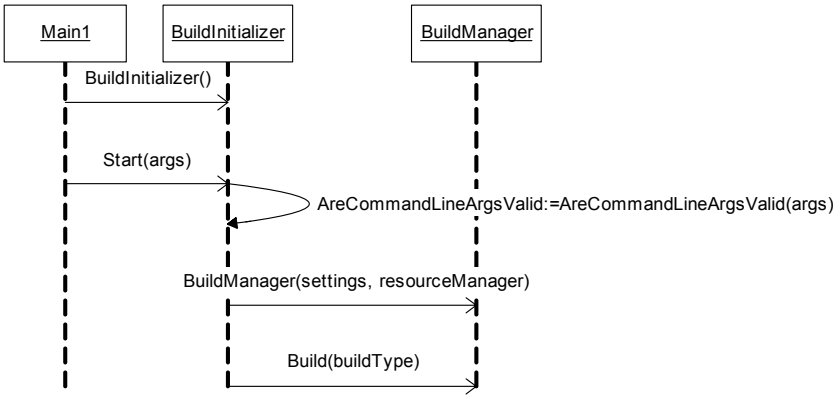


Figure 1

Build process flow diagram

The flow of logic is as follows:

1. BuildIt is launched with the appropriate command-line arguments.
2. The **Main1** class calls the **Start** method of the **BuildInitializer** class.
3. The **BuildInitializer** class validates the command-line arguments and then uses the **BuildItSectionHandler** class to retrieve settings from the tool's configuration file. The **BuildItSectionHandler** returns the settings by way of a type-safe structure called **BuildItSettings**.
4. The **BuildInitializer** creates an instance of the **BuildManager**, passing the **BuildItSettings** structure to the **Constructor**.
5. The **BuildInitializer** then calls **Build** on the **BuildManager** to initiate either a build or a rebuild, dictated by the command-line arguments.

Configuration File Settings

The BuildIt configuration file, called BuildIt.exe.config, contains settings that are read at runtime. This allows the tool's behavior to be modified without the need to recompile the solution.

Settings in the file indicate:

- The solutions being built.
- Information needed to connect to a specified VSS database.
- Whether to update assembly versions with the build number.
- Whether to archive previously built assemblies to a specified location.
- Whether to send the build results in an e-mail message to a specified e-mail address.

Note: BuildIt throws an exception if any of the settings are missing or specified incorrectly, unless stated otherwise in this document.

The following code sample shows the format of the configuration file settings.

```
<configuration>
  <configSections>
    <section name="BuildIt"
     type="Sapient.Framework.Tools.BuildIt.BuildItSectionHandler,
     BuildIt"/>
  </configSections>

  <appSettings>
    <add key="appendTraceOutput" value="off" />
    <add key="enableCustomMessageFilter" value="on"/>
    <add key="visualStudioProgID" value="VisualStudio.Solution.7"/>
  </appSettings>

  <buildIt>
    <sourceControl username="username" password="password"
     iniFilePath="c:\Program Files\Microsoft Visual Studio\VSS\srcsafe.ini"
     srcVSSRootFolder="$/System" srcFileRootFolder="c:\System"/>

    <solutions latestRootFolderFullName="c:\System"
     buildNumberFilePath="c:\System\BuildNumber.xml">
      <solution path=" c:\System\Solution1\Solution1.sln"/>
      <solution path=" c:\System\Solution2\Solution2.sln"/>
    </solutions>

    <options>
      <sendBuildReport mode="on/off" smtpServer="255.255.255.255"
       toAddress="you@yourcompany.com"/>
      <archiveBuild mode="on/off"
        archiveRootFolderFullName="c:\System\Archive">
        <additionalFoldersToArchive>
          <folder fullName="c:\System\Solution1\Setup1\Debug"
           destFolderName="Debug\Setup1" />
          <folder fullName=" c:\System\Solution1\Setup1\Release"
           destFolderName="Release\Setup1" />
        </additionalFoldersToArchive>
      </archiveBuild>
      <updateAssemblyVersion mode="on/off"
       vbAssemblyVersionVSSPath="$/System/AssemblyVersionInfo.vb"
       csAssemblyVersionVSSPath="$/System/AssemblyVersionInfo.cs"/>
    </options>
  </buildIt>
</configuration>
```

ConfigSections Element

The **configSections** element is used to link a section in the configuration file with a particular configuration section handler class. The **section** element in the preceding code is used to link the **BuildIt** section with the **BuildItSectionHandler** class in the **Sapient.Framework.Tools.BuildIt** namespace.

AppSettings Element

The **appSettings** element contains application-specific settings that can be represented as key-value pairs. BuildIt currently defines three specific settings:

- **appendTraceOutput.** When this key is set to **on**, BuildIt appends trace output to the trace file named BuiltIt.log.

  Note: If this key is omitted or if the value is not set to **on**, BuildIt overwrites the trace file.

- **enableCustomMessageFilter.** When this key is set to **on**, BuildIt enables a custom message filter to handle outgoing COM messages while waiting for a response from synchronous calls. The implementation of this filter retries rejected calls to Visual Studio .NET automation in the event that the objects are busy. If the Visual Studio .NET automation objects are busy and this filter is not enabled, an exception occurs.

  Note: If this key is omitted or if the value is not set to **on**, BuildIt does **not** enable the message filter.

- **visualStudioProgID.** When the value of this key is set to a progID, BuiltIt.NET is forced to use a specific version of Visual Studio .NET when building solutions.

  Note: If this key is omitted, BuildIt uses the latest installed version of Visual Studio .NET.

SourceControl Element

The **sourceControl** element contains information about the VSS database used during the build process. This element has the following attributes:

- **username.** The name of the VSS user used to log on to the database.
- **password.** The password of the VSS user.
- **iniFilePath.** The VSS .ini file path used to locate the database (for example, C:\Program Files\Microsoft Visual Studio\VSS\srcsafe.ini).
- **srcVSSRootFolder.** The VSS root folder that contains the source code being built (for example, $/System). BuildIt does a recursive **Get** operation from this VSS project.
- **srcFileRootFolder.** The file system root folder where the source code is built (for example, C:\System). BuildIt copies files from the recursive **Get** operation to this directory.

Solutions Element

The **solutions** element contains information about the solutions being built. This element has the following attributes:

- **latestRootFolderFullName.** The name of the root folder that contains the latest generated assemblies after the build is complete (for example, C:\System). For more information, see the "Copying Output to the Latest Folder" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

> **Note:** The article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp* states that "single solution or partitioned-single solution systems do not require a Latest folder because you use project references rather than file references to refer to assemblies." However, because a single solution system can become a multi-solution system, BuildIt **always** copies build output to a latest folder. This allows developers to use file references when referring to assemblies generated from another solution.

- **buildNumberFilePath.** The XML file used to generate the next build number (for example, C:\System\BuildNumber.xml). For more information, see the "Generating Build Version Numbers section" of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

> **Note:** The article "Team Development with Visual Studio .NET and Visual SourceSafe" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp* does not dictate the way in which build numbers should be generated. BuildIt does it by incrementing the build number stored in a specified XML file at the end of each successful build. A successful build is one that does not contain any build errors.

Solution Element

The **solution** element contains information about a specific solution. This element has one attribute:

- **path.** The solution file being built (for example, C:\System\Solution1\Solution1.sln).

Options Element

The **options** element contains child elements that correspond to various build options. Each build option is represented by a different element.

SendBuildReport Element

The **sendBuildReport** element controls whether the build report is sent to a specified e-mail address when the build completes. For more information, see the "Email Build Results" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

This element contains the following attributes:

- **mode.** If "on," BuildIt sends the build report to the specified e-mail address.
- **smtpServer.** If IP address of the SMTP server used to forward the e-mail message.
- **toAddress.** The e-mail address to receive the build report.

Archive Build Element

The **archiveBuild** element controls whether the generated assemblies are archived to a folder named after the build number. For more information, see the "Consider Maintaining Previous Builds" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

This element contains the following attributes:

- **mode.** If "on," BuildIt copies generated assemblies to a folder named after the build number.
- **archiveRootFolderFullName.** The name of the root folder containing previously archived builds (for example, C:\System\Archive).

Additional Folders to Archive Element

By default, BuildIt archives output from only Visual C# and Visual Basic .NET projects. To archive output from other projects such as Visual Studio .NET setup projects, add **folder** elements to the **additionalFoldersToArchive** element.

The folder element contains the following attributes:

- **fullName.** The full name of the source folder being archived (for example, C:\System\Solution1\Setup1\Debug).
- **destFolderName.** The name of the destination folder to archive to (for example, Debug\Setup1).

Note: If the **archiveBuild** option is enabled, BuildIt archives the files in each specified folder to the specified destination folder under the archive root folder.

UpdateAssemblyVersion Element

The **updateAssemblyVersion** element controls whether the generated assemblies are versioned using the build number. For more information, see the "Controlling Assembly Version" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com /library/default.asp?url=/library/en-us/dnbda/html/tdlg_ch5.asp*.

This element contains the following attributes:

- **mode.** If "on," BuildIt versions the generated assemblies with the build number.
- **vbAssemblyVersionVSSPath.** The VSS path containing the assembly version file shared across all of the Visual Basic .NET projects being built (for example, $/System/AssemblyVersionInfo.vb). Leave this attribute value blank if your solution does not contain any Visual Basic projects. For more information, see the "Consider Centralizing Assembly Version Numbers" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us /dnbda/html/tdlg_ch5.asp*.
- **csAssemblyVersionVSSPath.** The VSS path containing the assembly version file shared across all of the Visual C# projects being built (for example, $/System/AssemblyVersionInfo.cs). Leave this attribute value blank if your solution does not contain any Visual C# projects. For more information, see the "Consider Centralizing Assembly Version Numbers" section of the article "Team Development with Visual Studio .NET and Visual SourceSafe" in the MSDN Library at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us /dnbda/html/tdlg_ch5.asp*.

Enhancements

This section contains a list of enhancements that were considered but ultimately scoped out of the first release of BuildIt due to time constraints. These are shared to expose the underlying thinking involved with the development of BuildIt, to suggest areas where you could expand BuildIt, and to invite you as a member of the community to contribute ideas—and code—using the feedback section at the end of the document. Potential enhancements include:

- Define an interface to allow BuildIt to work with source control providers other than Visual SourceSafe (for example, **ISourceControlProvider**). In doing so, developers could configure BuildIt to use another source control provider such as Rational ClearCase.
- Simplify the configuration of BuildIt by implementing default behavior if a particular setting is missing. Currently, BuildIt throws an error if a particular build setting is missing from the application configuration file. A more forgiving version of BuildIt would simply implement some default behavior if a setting were omitted.

- Further simplify the configuration of BuildIt by utilizing relative paths instead of using fully qualified ones. Currently, all file paths must be fully qualified. Fully qualified paths are more flexible than relative ones, but at the expense of being cumbersome. Instead, simply configure by making some paths relative to the source root folder (for example, solution files, build number XML file, and so on).
- Allow developers to specify an indefinite number of assembly version files with the **updateAssemblyVersion** build option. Currently, BuildIt allows developers to specify one Visual C# and one Visual Basic .NET assembly version file. This limitation is based on the assumption that developers will share the Visual C# assembly version file with all Visual C# projects and the Visual Basic .NET assembly version file with all Visual Basic .NET projects in VSS. However, developers may prefer to update individual assembly version files rather than relying on VSS file sharing.
- Define a "post-build" event that developers can use to extend the build process. Currently, the only way to extend the build process is to alter the BuildIt source code. A more flexible design would be to implement an event that is raised at the end of the build. Developers could then wire event-handlers to this event to execute custom logic. This strategy could also be used to allow developers to customize other parts of the build process (for example, pre-build, build, post-build).
- Allow developers to specify the name of the BuildIt configuration file. Currently, BuildIt looks for configuration settings in the standard application configuration file called BuildIt.exe.config. Because of this, developers cannot use BuildIt to build two different solutions at the same time. Instead, developers have to copy the BuildIt assemblies to a different directory. A more friendly solution would be to allow developers to specify the name of the configuration file as a command-line argument, rather than using the standard application configuration file.
- Have BuildIt automatically create a virtual directory for any Web project that is not already mapped to one. Currently, Visual Studio .NET prompts the user to set the location for the working copy of a Web project if one does not exist. Instead of prompting the user to accept the default, BuildIt could simply create the virtual directory automatically.

Class Reference

This section includes information about the following class references:

- BuildInitializer
- BuildItSectionHandler
- BuildManager
- SourceSafeHelper
- BuildItResourceManager
- BuildItCommandLineArgs

BuildInitializer

The **BuildInitializer** class validates command-line options, retrieves build tool settings by way of the **BuildItSectionHandler** and initiates the build process by way of the **BuildManager**.

Remarks

The **BuildInitializer** class implements a single **public** method named **Start** that is called from the console application's **static/Shared Main** method. The **Start** method first validates any command-line arguments by calling a **private** method named **AreCommandLineArgsValid**. If the arguments are valid, the **BuildInitializer** retrieves build tool settings using the **BuildItSectionHandler** class. The final step is to initiate the build process by calling **Build** on the **BuildManager**.

Constructors

BuildInitializer	Initializes a new instance of the **BuildInitializer** class.

Public Methods

Start	Reads settings from the build tool configuration file using the **BuildItSectionHandler** class and then starts the build process by way of the **BuildManager** class.

Private Methods

AreCommandLineArgsValid	Determines whether the given command-line arguments are valid.
IsCommandLineArgValid	Determines whether the given command-line argument is valid.
Usage	Writes usage information to the console. Called if any of the command-line arguments are invalid.

BuildItSectionHandler

The **BuildItSectionHandler** class populates and returns a **BuildItSettings** structure with settings read from the build tool configuration file.

Remarks

The **BuildItSectionHandler** class, which implements the **IConfigurationSectionHandler** interface, parses the configuration settings in the **BuildIt** element of the build tool configuration file. It loads the information in the **BuildIt** element into a structure named **BuildItSettings**.

The **BuildItSettings** structure defines several properties that provide type-safe access to the configuration settings defined in the configuration file. For example, the **BuildItSettings** structure defines three properties that return different build options: **OptionSendBuildReport**, **OptionArchiveBuild**, and **OptionUpdateAssemblyVersion**. The **BuildItSettings** structure also defines a property called **Solutions** and one called **SourceControlInfo** that correspond to specific elements in the configuration file.

Constructors

BuildItSectionHandler	Initializes a new instance of the **BuildItSectionHandler** class.

Public Methods

Create	Populates and returns a **BuildItSettings** structure with settings read from the build tool configuration file.

Private Methods

CreateBuildOption	Creates and populates a **BuildOption** structure with settings read from the build tool configuration file.

BuildManager

The **BuildManager** class orchestrates the build process.

Remarks

The **BuildManager** class is used by the **BuildInitializer** to coordinate the build process. The **BuildManager** implements a constructor that takes two parameters: one of type **BuildItSettings** and one of type **BuildItResourceManager**. When an instance of the **BuildManager** class is created, it initializes private member variables with the parameters passed to its constructor. This makes settings and error messages available to the **BuildManager** during the build process.

The **BuildManager** also exposes a single **public** method named **Build** that takes a **BuildType** enum used to determine whether to initiate a Build or a Rebuild. The BuildType enum is defined as follows.

```
[C#]
public enum BuildType {Build, Rebuild};
```

A **Rebuild** is defined as a system build that takes place after a build break is resolved. The distinction is made because some steps are omitted when doing a Rebuild. For example, the **BuildManager** creates a new label in VSS when doing a **Build**, but not when doing a Rebuild.

After the **BuildType** is determined, the **BuildManager** calls several **private** methods that perform various parts of the build process.

Constructors

BuildManager	Initializes a new instance of the **BuildManager** class.

Public Methods

Build	Orchestrates the build process.

Private Methods

ArchiveAdditionalFolders	Copies the contents of folders specified in the **additionalFoldersToArchive** section of the build tool configuration file to the archive root folder.
BackupLatestFolder	Backs up the latest folder that contains the latest generated assemblies. The location of the latest folder is specified in the build tool configuration file.
BuildAssemblyVersionRegEx	Builds a regular expression that is used to replace the build number component of an **AssemblyVersion** attribute. The expression is dependent upon whether the **AssemblyVersion** attribute is a Visual Basic or Visual C# attribute.
BuildSolutionAndCopyOutput	Builds a solution and copies the generated output to the given destination directories.
BuildSolutionConfig	Builds the given solution using the specified build configuration (for example, Release, Debug, and so on).
BuildSolutions	Builds the given solution(s).
CopyFiles(string, string)	Copies all files from the given source folder to the given destination folder by calling the overloaded version of the method with **excludeDebugFiles** flag set to false.

(continued)

(continued)

CopyFiles(string, string, bool)	Copies all files from the given source folder to the given destination folder. If the **excludeDebugFiles flag** is set to true, copies all files except debug files.
CopyVSProjectOutput	Copies output (Release or Debug) for all Visual Basic and Visual C# projects to the given destination root folder.
CopySolutionOutput	Copies solution output (Release or Debug) to the given destination root folders.
DeleteArchiveFolder	Deletes the archive folder if it exists. The location of the archive folder is specified in the build tool configuration file.
DeleteLatestBackupFolder	Deletes the latest backup folder if it exists. The latest backup folder is located in the same place as the latest folder.
DeleteLatestFolder	Deletes the latest folder if it exists.
GenerateBuildReport	Generates a build report called **BuildReport.log** and saves the report to the working folder.
GenerateLabel	Generates a label based on the build number.
GetSourceFromLabel	Gets the source root project from VSS using the given label.
GetVSProjectLangType	Determines the language used to create the specified file (Visual Basic or Visual C#). Uses the file extension to make this determination.
IncrementBuildNumber	Increments the build number by one and then writes it to the build number XML file by calling **WriteBuildNumber**.
LabelSource	Labels the source root project in VSS with the given label.
ReadBuildNumber	Reads the build number from the build number XML file. The name and location of this file is specified in the build tool configuration file.
RestoreLatestFolder	Restores the latest folder from the latest backup folder if it exists.
SendBuildReport	Send the given build report to the appropriate email recipient if the option is enabled in the build tool configuration file.
UpdateAssemblyVersionInfo	Gets the VSS path of the Visual Basic and Visual C# **AssemblyVersion** files (specified in the build tool configuration file) and calls the overloaded version of the method once for each file if the option is enabled in the build tool configuration file.
UpdateAssemblyVersionInfo(string)	Gets the specified **AssemblyVersion** file from VSS, updates the **AssemblyVersion** attribute with the build number using a regular expression, and then checks the file back into VSS.
WriteBuildNumber	Writes the given build number to the build number XML file.

SourceSafeHelper

The **SourceSafeHelper** wraps VSS automation objects to make it easier to work with VSS.

Remarks

The **SourceSafeHelper** class is used by the **BuildManager** to interact with VSS during the build process. It exposes several **public** methods, such as **Checkin** and **Checkout**, that hide the implementation details of working with VSS automation objects.

Note: The **SourceSafeHelper** class does not expose methods for every VSS operation. It exposes only those operations needed by the build tool.

Constructors

SourceSafeHelper	Initializes a new instance of the **SourceSafeHelper** class.

Public Methods

Checkin	Checks the given item back into VSS.
Checkout	Checks the given item out from VSS.
GetFromLabel	Gets a specific version of a given item from VSS. If the item is a project, the get operation is recursive.
Label	Labels the given VSS item with the specified label.
UndoCheckout	Removes checkout status on a given VSS item.

Private Methods

GetItemsRecursively	Recursively gets items from a versioned **VSSItem** (a project or file). The versioned item can be a project or a file. If the VSSItem is a file, this method simply gets the file. Otherwise, it recursively gets all items in the project.
GetVSSItem	Gets a reference to a **VSSItem** (that is, a project or file).

BuildItResourceManager

The **BuildItResourceManager** class provides type-safe access to error messages stored in a resource file.

Remarks

The **BuildItResourceManager** class, derived from the **ResourceManager** class in the **System.Resources** namespace, is used to retrieve error messages from the resource file contained in the build tool assembly called BuildIt.exe. It exposes several **public** methods, such as **GetBuildSolutionFailedString**, that provide type-safe access to resource values. This helps eliminate errors that can occur when calling the ResourceManager.**GetString(string name)** method with an invalid resource name.

Constructors

BuildItResourceManager	Initializes a new instance of the **BuildItResourceManager** class.

Public Methods

GetBuildSolutionFailedString	Gets the BuildSolutionFailed error message from the build tool resource file. The rest of the get methods perform a similar function; they provide type-safe access to error messages stored in the build tool resource file.

Private Methods

None	

BuildItCommandLineArgs

The **BuildItCommandLineArgs** class encapsulates a collection of command-line arguments supported by the build tool.

Remarks

The **BuildItCommandLineArgs** class, derived from the **StringCollection** class in the **System.Collections.Specialized** namespace, represents the set of command-line arguments supported by BuildIt. It is used by the **BuildInitializer** to determine whether or not a command-line argument is valid.

Constructors

BuildItCommandLineArgs	Initializes a new instance of the **BuildItCommandLineArgs** class.

Public Methods

ToString	Overrides the **System.Object::ToString()** method by providing a string representation of the valid command-line arguments.

Private Methods

None	

Frequently Asked Questions

Can I modify the BuildIt source code?

Yes, you can modify and extend the BuildIt source code. For example, you could change the Visual Source Safe-specific calls and build a Helper class that works with your Source Code Control product. Please tell us what changes you made! See the Feedback section below to share your experience with us.

Do I need to have Visual Studio installed where BuildIt runs?

Yes, you need to have Visual Studio installed on the server or workstation where you do the builds with BuildIt—having just the .NET Framework redistributable installed is not enough.

Appendix Summary

This appendix has described how BuildIt is built with simplicity and extensibility in mind so that you can adapt it to your specific build-process needs. Please use the e-mail address provided under "Feedback" to tell us your thoughts about BuildIt.

About the Author

Michael Monteiro is a technology consultant for Sapient Corporation and .NET Microsoft Certified Professional with more than eight years of experience. He currently works on a team responsible for providing architectural guidance to the technical community within Sapient. He can be reached at *mmonte@sapient.com*.

About Sapient

Sapient, a leading business and technology consultancy, helps Global 2000 clients achieve measurable business results through the rapid application and support of advanced technology on a fixed-price basis. Founded in 1991, Sapient employs more than 1,500 people in offices in Atlanta, Cambridge (Mass.), Chicago, Dallas, Dusseldorf, London, Los Angeles, Milan, Munich, New Delhi, New York, San Francisco, Toronto and Washington, D.C. More information about Sapient can be found at *http://www.sapient.com/*.

Collaborators

Many thanks to the following contributors and reviewers: Bernard Chen (Sapient Corporation), Brett Keown, Craig Skibo, David Lewis, Deyan Lazarov, Dimitris Georgakopoulos (Sapient Corporation), Edward Jezierski, Filiberto Selvas Patiño, Jeff Pflum, Joe Hom (Avanade), Ken Hardy, Kenny Jones, Korby Parnell, Martin Born, Mick Das, Mike Pietraszak, Niel Sutton, Oded Ye Shekel, Rajiv Sodhi (Sapient Corporation), Ray Escamilla, Rich Knox, Russell Christopher, and Sumit Sharma (Sapient Corporation).

Feedback

Questions? Comments? Suggestions? To give feedback on BuildIt, please e-mail *buildit@sapient.com*.

If you want to contact Microsoft for feedback, please e-mail *devfdbck@microsoft.com*.

Collaborators

Many thanks to the following contributors and reviewers:

Michael Day, Martyn Lovell, Brad Bartz, Izzy Gryko, Bill Hiebert, Jeff Pflum, Bernard Chen (Sapient), Michael Monteiro (Sapient), Dimitris Georgakopoulos (Sapient), Korby Parnell, Susan Warren, Chris Falter, Joel West, Dave Quick, Allan Hirt, Cathan Cook, Chong Lee, Milind Lele, Chris Brooks, Martin Petersen-Frey, J.D. Meier, Edward Jezierski, Jacquelyn Schmidt, Jeremy Bostron, Frank Hackländer (Siemens), Reinhold Kienzle-Press (Siemens), Sharon Bjeletich, Andrew Roubin (Voresite)

Microsoft®

patterns & practices

Proven practices for predictable results

Patterns & practices are Microsoft's recommendations for architects, software developers, and IT professionals responsible for delivering and managing enterprise systems on the Microsoft platform. Patterns & practices are available for both IT infrastructure and software development topics.

Patterns & practices are based on real-world experiences that go far beyond white papers to help enterprise IT pros and developers quickly deliver sound solutions. This technical guidance is reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. Organizations around the world have used patterns & practices to:

Reduce project cost

- Exploit Microsoft's engineering efforts to save time and money on projects
- Follow Microsoft's recommendations to lower project risks and achieve predictable outcomes

Increase confidence in solutions

- Build solutions on Microsoft's proven recommendations for total confidence and predictable results
- Provide guidance that is thoroughly tested and supported by PSS, not just samples, but production quality recommendations and code

Deliver strategic IT advantage

- Gain practical advice for solving business and IT problems today, while preparing companies to take full advantage of future Microsoft technologies.

To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices
Proven practices for predictable results

patterns & practices

Proven practices for predictable results

Patterns & practices are available for both IT infrastructure and software development topics. There are four types of patterns & practices available:

Reference Architectures

Reference Architectures are IT system-level architectures that address the business requirements, operational requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems and are most useful for architects.

Reference Building Blocks

References Building Blocks are re-usable sub-systems designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development.

Reference Building Blocks focus on the design and implementation of sub-systems and are most useful for designers and implementors.

Operational Practices

Operational Practices provide guidance for deploying and managing solutions in a production environment and are based on the Microsoft Operations Framework. Operational Practices focus on critical tasks and procedures and are most useful for production support personnel.

Patterns

Patterns are documented proven practices that enable re-use of experience gained from solving similar problems in the past. Patterns are useful to anyone responsible for determining the approach to architecture, design, implementation, or operations problems.

To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices current titles

December 2002

Reference Architectures

Microsoft Systems Architecture—Enterprise Data Center *2007 pages*
Microsoft Systems Architecture—Internet Data Center *397 pages*
Application Architecture for .NET: Designing Applications and Services *127 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning *92 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment *128 pages*
Enterprise Notification Reference Architecture for Exchange 2000 Server *224 pages*
Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001 *124 pages*
UNIX Application Migration Guide *694 pages*
Microsoft Active Directory Branch Office Guide: Volume 1: Planning *88 pages*
Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations *195 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning *227 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment *135 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning *306 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment *166 pages*

Reference Building Blocks

Data Access Application Block for .NET *279 pages*
.NET Data Access Architecture Guide *60 pages*
Designing Data Tier Components and Passing Data Through Tiers *70 pages*
Exception Management Application Block for .NET *307 pages*
Exception Management in .NET *35 pages*
Monitoring in .NET Distributed Application Design *40 pages*
Microsoft .NET/COM Migration and Interoperability *35 pages*
Production Debugging for .NET-Connected Applications *176 pages*
Authentication in ASP.NET: .NET Security Guidance *58 pages*
Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication *608 pages*

Operational Practices

Security Operations Guide for Exchange 2000 Server *136 pages*
Security Operations for Microsoft Windows 2000 Server *188 pages*
Microsoft Exchange 2000 Server Operations Guide *113 pages*
Microsoft SQL Server 2000 Operations Guide *170 pages*
Deploying .NET Applications: Lifecycle Guide *142 pages*
Team Development with Visual Studio .NET and Visual SourceSafe *74 pages*
Backup and Restore for Internet Data Center *294 pages*

For current list of titles visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices
Proven practices for predictable results