1

Introducing SQL CLR

COPYRIGHTED MATERIAL

SQL Server's .NET integration is arguably the most important feature of SQL Server 2005 for developers. Developers can now move their existing .NET objects closer to the database with SQL CLR. SQL CLR provides an optimized environment for procedural- and processing-intensive tasks that can be run in the SQL Server tier of your software's architecture. Also, database administrators need a strong knowledge of SQL CLR to assist them in making key administrative decisions regarding it. If you ignore SQL CLR, you're missing out on the full potential SQL Server 2005 can offer you and your organization, thus limiting your effectiveness with the product.

SQL CLR is a very hot topic in the technical communities but also one that is frequently misunderstood. Unquestionably, there will be additional work devoted to SQL CLR from Microsoft and Paul Flessner (Microsoft's senior vice president, server applications), including the support of future database objects being created in SQL CLR. The book you are reading is your one and only necessary resource for commanding a strong knowledge of SQL CLR, including understanding when to use the technology and, just as importantly, when not to use it.

What is SQL CLR?

SQL CLR is a new SQL Server feature that allows you to embed logic written in C#, VB.Net, and other managed code into the body of T-SQL objects like stored procedures, functions, triggers, aggregates and types. Client applications interact with these resulting routines like they are written in native T-SQL. Internally, things like string manipulations and complex calculations become easier to program because you are no longer restricted to using T-SQL and now have access to structured .Net languages and the reuse of base class libraries. Externally, the logic you create is wrapped in T-SQL prototypes so that the client application is not aware of the implementation details. This is advantageous because you can employ SQL CLR where you need it without re-architecting your existing client code.

With SQL CLR, you are also freed from the constraint of logic that applies only within the context of the database. You can with appropriate permissions write logic to read and write to file systems,

use logic contained in external COM or .Net DLLs, or process results of Web service or remoting methods. These capabilities are exciting and concerning, especially in the historical context of new feature overuse. To help you use this new feature appropriately, we want to make sure that you understand how it integrates with SQL Server and where this feature may be heading. In this chapter, we'll give you this type of overview. We'll spend the rest of the book explaining these concepts using real-world SQL CLR examples that you can use today.

The Evolution of SQL CLR

A few years ago, we came across a product roadmap for Visual Studio and SQL Server that mentioned a feature that was described as "creating SQL Server programming objects in managed languages." At the time, we could not comprehend how this feature would work or why Microsoft had chosen to do this. .NET 1.0 had just been released not, and to be able to use these compiled procedural languages to create database objects just did not "compute" to us.

At the time of this writing, the year is 2006 and Microsoft SQL Server 2005 has arrived with a big roar in the database market. *CLR Integration* is the official Microsoft term for the .NET Framework integration into SQL Server. *SQL CLR* was the original term used by Microsoft to refer to this technology and it continues to be used predominantly in the surrounding technical communities.

> **The Common Language Runtime (CLR) is the core of the Microsoft .NET Framework, providing the execution environment for all .NET code. SQL Server 2005 hosts the CLR, thus the birth name of the technology "SQL CLR," and its successor "CLR Integration."**

Pre-SQL Server 2005 Extensibility Options

Before SQL Server 2005, there was a handful of options a database developer could implement to extend beyond the boundaries of T-SQL. As we will discuss in this chapter, SQL CLR is almost always a better environment for these routines. The pre-SQL Server 2005 extensible options are:

- Extended Stored Procedures, C/C++ DLLs that SQL Server can dynamically load, run, and unload.
- sp_oa Procedures, OLE automation extended stored procedures. You can use these system procedures to invoke OLE objects. Even with the arrival of the SQL CLR technology there may be times when you still need to use these procedures for those situations in which you must use a Object Linking and Embedding (OLE) object.

Why Does SQL CLR Exist?

Dr. E. F. "Ted" Codd is the "father" of relational databases and thus Structured Query Language (SQL) as well. SQL is both an American National Standards Institute (ANSI) and International Organization for Standardization (ISO) standard. SQL (and its derivatives, including T-SQL) are set-based languages designed to create, retrieve, update, and delete (CRUD) data that is stored in a relational database

management system (RDBMS). SQL was and still is the natural choice when you only require basic CRUD functionality.

There are many business problems in the real world that require much more than basic CRUD functionality, however. These requirements are usually fulfilled in another logical layer of a software solution's architecture. In today's current technical landscape, web services, class libraries, and sometimes even user interfaces fulfill the requirements beyond CRUD. Passing raw data across logical tiers (which sometimes can be physical tiers as well) can be undesirable, depending upon the entire solution's requirements; in some cases, it may be more efficient to apply data transformations on the raw data before passing it on to another logical tier in your architecture. SQL CLR allows the developer to extend beyond the boundaries of T-SQL in a safer environment than what was previously available, as just discussed. There are unlimited potential uses of SQL CLR, but the following situations are key candidates for it:

- Complex mathematical computations
- String operations
- Recursive operations
- Heavy procedural tasks
- Porting extended stored procedures into the safer managed code environment

The Goals of SQL CLR

The architects and designers of the CLR integration into SQL Server at Microsoft had a few objectives for the functionality:

- **Reliability:** Managed code written by a developer should not be able to compromise the SQL Server hosting it.
- **Scalability:** Managed code should not stop SQL Server from supporting thousands of concurrent user sessions, which it was designed to support.
- **Security:** managed code must adhere to standard SQL Server security practices and permissions. Administrators must also be able to control the types of resources that the CLR assemblies can access.
- **Performance:** Managed code being hosted inside SQL Server should execute just as fast as if the code were running outside of SQL Server.

Supported SQL CLR Objects

SQL CLR objects are the database objects that a developer can create in managed code. Originally, most people thought that SQL CLR would only allow the creation of stored procedures and functions, but luckily there are even more database programming objects supported. As of the RTM release of SQL Server 2005, you can create the following database objects in a managed language of your choice:

- Stored procedures
- Triggers (supporting both Data Manipulation Language [DML] and Data Definition Language [DDL] statements)

- User-defined Functions (UDF) (supporting both Scalar-Valued Functions [SCF] and Table-Valued Functions [TVF])
- User-defined aggregates (UDA)
- User-defined types (UDT)

Stored procedures are stored collections of queries or logic used to fulfill a specific requirement. Stored procedures can accept input parameters, return output parameters, and return a status value indicating the success or failure of the procedure. Triggers are similar to stored procedures in that they are collections of stored queries used to fulfill a certain task; however, triggers are different in that they execute in response to an event as opposed to direct invocation. Prior to SQL Server 2005, triggers only supported `INSERT`,`UPDATE`, `DELETE` events, but with SQL Server 2005 they also support other events such as `CREATE TABLE`.

Functions are used primarily to return either a scalar (single) value or an entire table to the calling code. These routines are useful when you want to perform the same calculation multiple times with different inputs. Aggregates return a single value that gets calculated from multiple inputs. Aggregates are not new with SQL Server 2005; however, the ability to create your own is new. User-defined types provide you with a mechanism to model your specific data beyond what the native SQL Server data types provide you with.

The .NET Architecture

The *assembly* is the unit of deployment for managed code in .NET, including SQL Server's implementation of it. To understand what an assembly is, you need a fundamental understanding of the CLR (which is the heart of the .NET Framework, providing all managed code with its execution environment). The CLR is the Microsoft instance of the CLI standard. The CLI is an international standard developed by the European Computer Manufactures Association (ECMA) that defines a framework for developing applications in a language agnostic manner. The CLI standard's official name is ECMA-335, and the latest draft of this standard, as of the time of this writing, was published in June 2005.

There are several key components of the CLI and thus the Microsoft implementation of it (the CLR and Microsoft Intermediate Language [MSIL] code):

- Common Type System (CTS)
- Common Language Specification (CLS)
- Common Intermediate Language (CIL)
- Virtual Execution System (VES)
- Just-in-Time compiler (JIT)

Figure 1-1 shows how these components work together.

CTS

The CTS defines a framework for both value types and reference types (classes, pointers, and interfaces). This framework defines how types can be declared, used, and managed. The CTS describes type safety,

how types are agnostic of programming language, and high-performing code. The CTS is a core component in supporting cross-language interoperability given the same code base. In the Microsoft .NET context, this is implemented by the .NET Framework as well as custom types created by users (all types in .NET must derive from `System.Object`).

CLS

The CLS defines an agreement between programming language and class library creators. The CLS contains a subset of the CTS. The CLS serves as a baseline of CTS adoption for designers of languages and class libraries (frameworks). In the Microsoft.NET context, this could be any .NET-supported language including Visual Basic .NET and C#.

CIL

CIL is the output of a compiler that adheres to the CLI standard. ECMA-335 specifies the CIL instruction set. CIL code is executed by the VES (discussed shortly). In the Microsoft .NET context, this is the code you produce when you build a project in Visual Studio or via the command line using one of the supplied compilers such as VBC for Visual Basic .NET and CSC for C#. Microsoft Intermediate Language (MSIL) is the Microsoft instance of CIL.

VES

The VES is the execution environment for CIL code; the VES loads and runs programs written in CIL. There are two primary pieces of input for the VES, CIL/MSIL code and metadata. In the Microsoft .NET context, this occurs during the runtime of your .NET applications.

JIT

The JIT compiler is a subsystem of the VES, producing native CPU-specific code from CIL/MSIL code. As its name implies, the JIT compiler compiles CIL/MSIL code at runtime as it's requested, or just in time. In the Microsoft.NET context, this is during runtime of your .NET applications as you instantiate new classes and their associated methods.

How does the assembly fit into this CLI architecture? In the .NET context, it is when you compile your source code (this is what occurs "behind the scenes" when you build a project in Visual Studio as well) using one of the supported compilers, the container for your MSIL code is the assembly. There are two categories of assemblies in .NET: the EXE and the DLL (SQL CLR assemblies are always DLLs). Assemblies contain both the MSIL code and an assembly manifest. Assemblies can be either single-file- or multi-file-based. In the case of a single-file assembly, the manifest is part of the contents of the assembly. In the case of the latter, the manifest is a separate file itself. Assemblies are used to identify, version, and secure MSIL code; it is the manifest that enables all of these functions.

> *If you're a compiler guru or you just wish to learn more about the internal workings of Microsoft's CLR implementation of the CLI standard, Microsoft has made the Shared Source Common Language Infrastructure 1.0 Release kit available to the general public. You can find this limited open source version of the CLR at* `www.microsoft.com/downloads/details.aspx?FamilyId=3A1C93FA-7462-47D0-8E56-8DD34C6292F0&displaylang=en`.

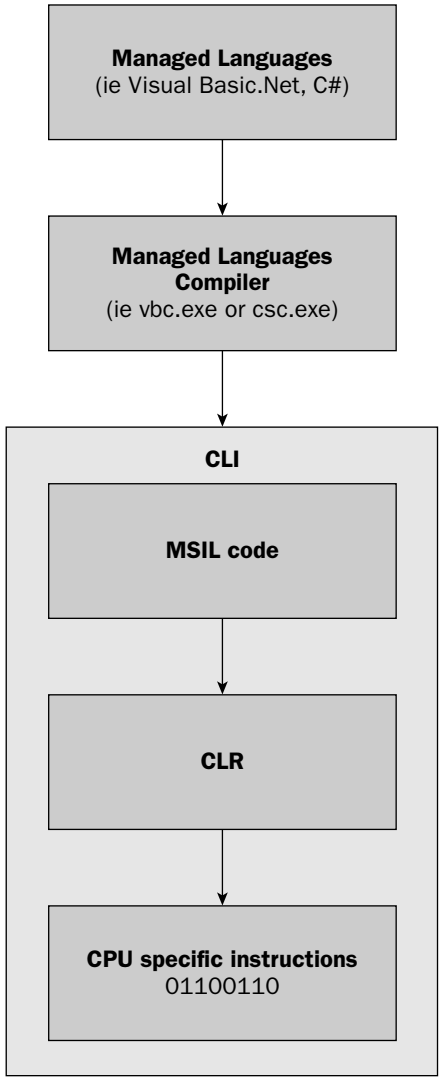

Figure 1-1

Managed Code and Managed Languages

Code that is executed inside the CLR is called *managed code*. Thus, the languages that produce this code are called managed languages. The languages are called *managed* because the CLR provides several run-time services that manage the code during its execution, including resource management, type-checking, and enforcing security. Managed code helps prevent you from creating a memory-leaking application. It is this "managed nature" of CLR code that makes it a much more secure environment to develop in than previous platforms.

Hosting the CLR

The CLR was designed from the beginning to be hosted, but only since the arrival of SQL Server hosting has the concept received so much attention. All Windows software today begins unmanaged, that is, running native code. In the future, the operating system will probably provide additional CLR services so that applications can actually begin as managed software as opposed to what we have today. Thus, in today's Windows environment any application that wishes to leverage the CLR and all its advantages must host it in-process, and this means invoking it from unmanaged code. The .NET Framework includes unmanaged application program interfaces (APIs) to enable this loading and initialization of the CLR from unmanaged clients. It is also worthwhile to mention that the CLR supports the notion of running multiple versions of itself side by side on the same machine, but a CLR host can only load one version of the runtime; thus, it must decide which version to load before doing so.

There are several pieces of software that you use today that host the CLR, including ASP.NET, Internet Explorer, and shell executables.

SQL CLR Architecture

SQL Server 2005 hosts the CLR in a "sandbox"-like environment in-process to itself, as you can see Figure 1-2. When a user requests a SQL CLR object for the first time, SQL Server will load the .NET execution engine `mscoree.dll` (which is the CLR) into memory. If you were to disable SQL CLR (see Chapter 2 for more details) at a later point in time, the hosted CLR would immediately be unloaded from memory.

This contained CLR environment aids SQL Server in controlling key CLR operations. The CLR makes requests to SQL Server's operating system (SQLOS) for resources such as new threads and memory; however, SQL Server can refuse these requests (for example, if SQL Server has met its memory restriction, and doesn't have any additional memory to allocate to the CLR). SQL Server will also monitor for long-running CLR threads, and if one is found, SQL Server will suspend the thread.

> **SQLOS is not a topic for the novice; we are talking about the "guts" of SQL Server here. SQLOS is an abstraction layer over the base operating system and its corresponding hardware. SQLOS enables SQL Server (and future server applications from Microsoft) to take advantage of new hardware breakthroughs without having to understand each platform's complexities and uniqueness. SQLOS enables such concepts as locality (fully utilizing local hardware resources to support high levels of scalability) and advanced parallelism. SQLOS is new with SQL Server 2005.**
>
> **If you wish to learn more about SQLOS you can visit Slava Oks's weblog at `http://blogs.msdn.com/slavao`. Slava is a developer on the SQLOS team for Microsoft.**

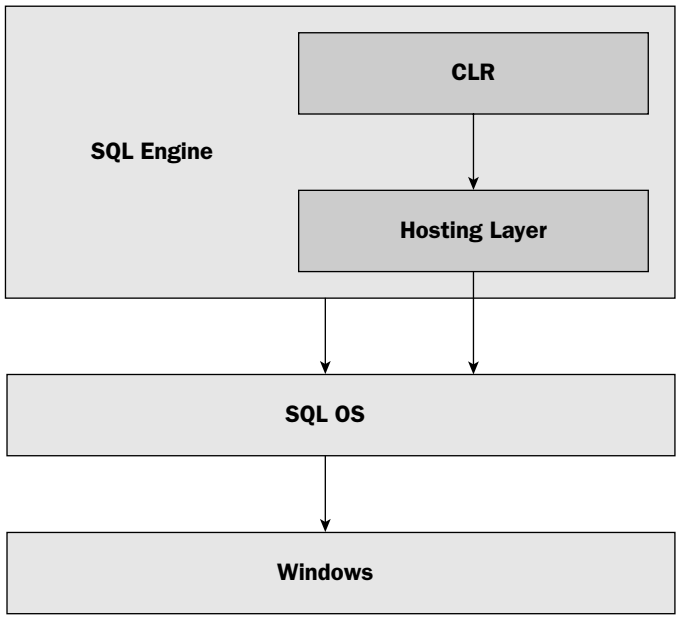

Figure 1-2: The SQL CLR Architecture

Application Domains

You can think of an application domain as a mini-thread, that is an execution zone. Application domains are managed implicitly by the CLR for you. Application domains are important because if the CLR code encounters a fatal exception, its corresponding application domain will be aborted, but not the entire CLR process. Application domains also increase code reliability, including the SQL Server hosting of it. All assemblies in a particular database owned by the same user form an application domain in the SQL Server context. As new assemblies are requested at runtime they are loaded into either an existing or new application domain.

The CLR Security Model

Microsoft's CLR security model is called code access security (CAS). The primary goal of CAS is to prevent unauthenticated code from performing tasks that should require preauthentication. The CLR identifies code and its related code groups as it loads assemblies at runtime. The code group is the entity that assists the CLR in associating a particular assembly with a permission set. In turn, the permission set determines what actions the code is allowed to perform. Permission sets are determined by a machine's administrator. If an assembly requires higher permissions that what it was granted, a security exception is thrown.

So how does the CLR security model apply to SQL CLR? There are two methods of applying security in SQL CLR, CAS permission sets and Role-Based Security (RBS) also known as Role-Based Impersonation (RBI), both of which can be used to ensure that your SQL CLR objects run with the required permissions and nothing more.

SQL CLR CAS Permission Sets

When you load your assemblies into SQL Server, you assign those assemblies a permission set. These permission sets determine what actions the corresponding assembly can perform. There are three default permission sets you can assign to your assemblies upon loading them into SQL Server (see Chapter 8 for more details):

- **SAFE** indicates that computational or string algorithms as well as local data access is permitted.
- **EXTERNAL_ACCESS** inherits all permissions from SAFE plus the ability to access files, networks, registry, and environmental variables.
- **UNSAFE** is the same as EXTERNAL_ACCESS without some of its restrictions and includes the ability to call unmanaged code.

RBS/RBI

RBS is useful when your SQL CLR objects attempt to access external resources. SQL Server will natively execute all SQL CLR code under the SQL Server's service account; this can be an erroneous approach to enforcing security because all users of the SQL CLR object will run under the same Windows account regardless of their individual permissions. There are specific APIs to allow SQL CLR developers to check and impersonate the user's security context (see Chapter 8 for more details).

> *RBS is only applicable when the user is logged in to SQL Server using Windows Authentication. If your SQL CLR code determines the user is using SQL Server Authentication, it will automatically deny them access to resources that require authentication.*

Key SQL CLR Decisions

With the arrival of the SQL CLR technology, you're now faced with several key decisions to be made regarding when and how to use it. These key decisions are some of the hottest topics regarding the technology in the online and offline technical communities.

Using SQL CLR or T-SQL

The one decision always coming to the foreground is now, "Should I implement this object in T-SQL or managed code?" Do not discount T-SQL just because you now have a more secure alternative in SQL CLR. T-SQL has been greatly enhanced in SQL Server 2005. If you take away only one concept from this chapter, let it be that SQL CLR is not a replacement for T-SQL—SQL CLR complements T-SQL. The following table compares the various attributes of each environment.

Attribute	T-SQL	SQL CLR
Code Execution	Interpreted	Compiled
Code Model	Set-Based	Procedural-Based
Access to subset of the .NET Framework Base Class Libraries (BCL)	No	Yes
SQL Server Access	Direct Access	In-Process Provider
Support Complex types	No	Yes
Parameter Support	Input and Output Parameters	Input and Output Parameters

Be aware that the CLR natively does not support `VarChar` *or* `Timestamp` *datatypes.*

Based on this information, you can draw some general conclusions about when to use T-SQL for a solution and when to employ SQL CLR: In general, if you are performing basic CRUD functionality you should use traditional T-SQL for these tasks; for anything else, such as intensive cursor processing, accessing external resources, you should use SQL CLR.

Using SQL CLR or Extended Stored Procedures

Extended Stored Procedures (XPs) are typically written in C/C++ against the SQL Server Extended Procedure API, and produce a DLL that SQL Server can load, execute, and unload at runtime. XPs are notorious for causing memory leaks and compromising the integrity of the SQL Server process. In addition, XPs will not be supported in future versions of SQL Server. SQL CLR provides a much safer alternative to using XPs. We recommend that all new development that requires such functionality employ SQL CLR and not XPs. We also recommend that you immediately begin the process of converting your existing XPs to the CLR (see Chapter 4 for more details).

One of the strongest reasons to adopt SQL CLR is to port existing XPs to the safer environment of the CLR.

Using SQL CLR or OLE Automation Procedures

Since SQL Server 6.5 standard edition, SQL Server has supported the functionality of calling Component Object Model (COM) objects from within T-SQL; this feature is known as OLE Automation. OLE Automation is implemented in SQL Server using several XPs prefixed with `sp_oa`, `oa` meaning OLE Automation. By leveraging these XPs in your T-SQL code you can instantiate objects, get or set properties, call methods, and destroy objects.

Similarly to using XPs, the COM objects you invoke with `sp_oa` run in the same address space as the database engine, which creates the possibility of compromising the entire database engine. OLE Automation has been known to cause memory leaks too; there have been knowledge base articles from Microsoft about these XPs leaking memory natively without poor coding techniques. So again, the main benefit of SQL CLR as compared to older technologies is stability and security.

Using the Data Tier or Application Tier for Business Logic

Prior to SQL Server 2005, business logic would typically reside on another server in the form of a COM object or a web service. With the advent of SQL CLR middle-tier developers now have a choice regarding where they want their routines to "reside." When you're deciding where your business logic should be placed, there is one key issue you must consider: the cost of bandwidth versus the cost of computing resources. Generally speaking, you are going to want to consider leveraging SQL CLR for your business logic when your data access returns a lot of data. On the other hand, when your business logic returns minimal data, it probably makes more sense not to employ SQL CLR. Again, the key here is which resources do you have more of, and which resource do you wish to use to support your business logic processing? Maybe you have excessive amounts of bandwidth to spare, but your SQL Server's CPUs and memory are already maxed out. In this case, it may still make more sense to send a lot of data across the wire as you would have done previously.

The other consideration is whether or not you wish to keep your business logic close to the database to help promote its global use. Typically, developers will create a logical middle tier to encapsulate their business logic, but one drawback of this approach is that your business logic is not as "close" to the database it's accessing. If you were to employ SQL CLR for your business logic it would help, but not enforce, your middle-tier logics use. The bottom line here is that each environment and software solution is unique, but these are the issues you should be contemplating when making these crucial decisions.

SQL CLR Barriers of Entry

Not only are there crucial decisions that must be made about a new technology, but there are also barriers of entry in order to properly use and obtain the benefit provided by the technology. SQL CLR is no different in this aspect. There are security, implementation, performance, and maintenance tasks that should be addressed.

Security Considerations

We realize that security from a developer's standpoint is more of a nuisance and you'd rather leave this task to the DBAs. But our experience as DBAs tell us that security is a from-the-ground-up thought process with TSQL and even more so with SQL CLR. If you're a developer, we have some shocking news: you are going to have to understand SQL CLR security because it's very important in dictating what your assemblies are allowed to do. Organizations that are successful in deploying SQL CLR assemblies will foster teamwork between its DBAs and developers (more than the typical organization does).

As an example of the concepts that you'll be deciding about the security of SQL CLR code is how TSQL and SQL CLR will enforce and respect security between shared calls. *Links* or *Comingling* is the term assigned to the relationship formed between T-SQL and managed code calling one another. You should already be aware of CAS permission sets as they were briefly covered earlier in the chapter, as well as Role-Based Impersonation. You have also heard a bit about application domains; just be aware that application domains, in a security context, are important because they form a level of isolation for your managed code. So, if a piece of SQL CLR tries to perform an illegal operation, the entire application domain gets unloaded. Bottom line, security is always important, but in the SQL CLR context an open dialogue between both developers and DBAs is even more important. Chapter 9 will explore SQL CLR security in depth.

The DBA Perspective on SQL CLR

The DBA is typically very cautious about giving developers flexibility, as they should be. One piece of poorly written code could compromise the entire SQL Server instance executing it. As we previously mentioned, XPs are notorious for causing SQL Server problems, and this has contributed to DBAs enforcing strong policies on developers. After all, the DBA is ultimately responsible for the overall health of a SQL Server, and not the developers.

SQL CLR's arrival is forcing DBAs to consider the question of "should I enable this thing or not?" It has been our experience with SQL Server 2005 thus far that DBAs will leave this feature turned off unless it is explicitly required to fulfill certain requirements of the organization. SQL CLR is turned off, by default upon a fresh installation of SQL Server 2005, part of Microsoft's "secure by default" strategy.

Although leaving SQL CLR off is not necessarily a bad option, we also think enabling SQL CLR and restricting what your developers can do via CAS permissions is a better solution. By allowing (and even promoting) SQL CLR in your SQL Server environment, developers will have a safe, secure, and managed alternative to T-SQL with permissions designated by you. After reading this book, DBAs will be well equipped to be proactive about SQL CLR and at the same time be confident in what you are allowing your developers to do with the technology.

Implementation Considerations

Before you even think about using SQL CLR (whether you're a DBA or developer), you must learn either VB.NET or C#. This is not an option. VB.NET and C# are the only officially supported SQL CLR languages. (Managed C++ is somewhere in between not supported and supported, because it does have a Visual Studio template for SQL CLR projects, but it must also be compiled with a special `/safe` switch for reliability). We encourage you to explore using other managed languages for creating SQL CLR objects, but be aware that they are not officially supported as of the time of this writing.

Logically you're probably now wondering what we mean by officially supported managed languages. Remember, all managed code gets translated into MSIL, so really we are misleading you here. The language you choose is largely about a choice of syntax, but what is supported and not supported is what your managed code does and what types it uses. The reason VB.NET and C# are the officially supported managed languages for SQL CLR is that they have built-in project templates that are guaranteed to generate code that is safe for SQL CLR execution. If you create a new VB.NET/C# Database/SQL Server project, and select "Add Reference," you will notice that not all of the .NET Framework classes or types appear which brings us to the next implementation point. You do not have access to the complete .NET Framework Base Class Library (BCL) in SQL CLR, as most people presume. In reality, you have access to a subset of the BCL that adheres to the SQL Server host requirements.

Host Protection Attributes (HPAs) are the implementation for a CLR host's requirements, which determine what types can be used when the CLR is hosted as opposed to when the CLR is running natively. Additionally, HPAs become more important when coupled with the various CAS permission sets you can apply to your SQL Server assemblies. For now, just understand that there are CLR host requirements, which assist SQL Server in determining if it should allow the use of a particular type or not.

> *If you use an unsupported managed language to create an assembly that you then load into SQL Server as an UNSAFE assembly and that assembly does not adhere to the HPAs, the assembly could attempt an operation that could threaten the stability of the SQL Server hosting it. Any time the stability of*

SQL Server is threatened by an assembly, the offending assembly's entire application domain gets unloaded. This fact goes a long way in proving that stability was designed and implemented well in SQL CLR.

Finally, there is the relation of the CAS permission sets to implementation considerations (which we started to discuss above). The CAS permission set assigned to your SQL CLR assemblies will ultimately determine just how much you can do in the assembly. One of the most common questions about SQL CLR is whether you can call native code in a SQL CLR assembly. The answer is based on the permission set assigned to the assembly. In Chapter 3, we will thoroughly go through all of this material, but realize that if you assign the UNSAFE permission set to an assembly it can attempt practically any operation (including calling native code). Whether the host of the CLR executes "allows" it or not is another matter.

Performance Considerations

Performance is always important, but even more so when you're talking about code running inside of a relational database engine designed to handle thousands of concurrent requests. Consider these SQL CLR performance issues:

First and foremost, if your routine simply needs to perform basic relational data access, it will always perform better when implemented in T-SQL than in SQL CLR. If there is a "no brainer" performance decision relating to SQL CLR, it is when your code just needs to perform CRUD functionality, use T-SQL 100% every time.

Second, transactions are important for your SQL CLR performance considerations. By default, all SQL CLR code is enlisted in the current transaction of the T-SQL code that called it. Thus, if you have a long-running SQL CLR routine, your entire transaction will be that much longer. The point is, be very hesitant about creating long-running routines in SQL CLR (see Chapter 3 for more details on the handling of transactions).

Last, once you have elected to use SQL CLR for a particular routine, be aware of the various resources you have at your disposal to monitor SQL CLR performance (see Chapter 9 for more information). These resources include:

- System and Dynamic Management Views
- SQL Trace Events (captured with SQL Server Profiler or system stored procedures for custom monitoring needs)
- Performance Counters

Maintenance Considerations

If we had a dollar for every SQL Server instance we have worked on that was poorly maintained . . . we wouldn't be rich, but we'd be well off. Databases can easily become unwieldy beasts if you don't take a proactive approach to maintenance (this is usually the case when we work on SQL Server installations that did not have an official DBA assigned to them). SQL CLR assemblies and their related database objects are no different.

The DBA is usually the title of the person in an organization who is responsible for database maintenance, so naturally this aspect of SQL CLR is going to affect him or her more than the developer. If the

DBA chooses to enable SQL CLR in one of the SQL Servers, he or she should also be willing to keep track and monitor the inventory of SQL CLR assemblies. Not only would it be a serious security risk to allow developers to "push" SQL CLR assemblies to a production box without first consulting the DBA, but you could also end up with potentially hundreds of assemblies stored in your database that no one uses anymore or even knows their purpose.

SQL Server 2005 SQL CLR support

All nonportable editions of SQL Server 2005 support SQL CLR, including SQL Server Express. We also found that SQL Server 2005 Mobile does not offer similar functionality via hosting the .NET Compact Framework's CLR in-process on portable devices. We will be using the Express edition of SQL Server 2005 for the creation of our code samples in this book. SQL Server Express is the free successor to MSDE (the older SQL Server 2000–based engine that had a workload governor on it to limit concurrent access). SQL Server Express can be found at `http://msdn.microsoft.com/vstudio/express/sql`. In addition, there is a free management tool built explicitly for the Express edition called SQL Server Management Studio Express, which can also be found at `http://msdn.microsoft.com/vstudio/express/sql`. The following are the system requirements for SQL Server Express:

Resource	Required	Recommended
Processor	600-megahertz (MHz) Pentium III–compatible or faster processor	1–gigahertz (GHz) or faster processor
Operating System	Windows XP with Service Pack 2 or later Windows 2000 Server with Service Pack 4 or later Windows Server 2003 Standard, Enterprise, or Datacenter editions Windows Server 2003 Web Edition Service Pack 1 Windows Small Business Server 2003 with Service Pack 1 or later	N/A
Memory	192 megabytes (MB) of RAM	512 megabytes (MB)
Hard Disk Drive	Approximately 350 MB of available hard-disk space for the recommended installation	Approximately 425 MB of additional available hard-disk space for SQL Server Books Online, SQL Server Mobile Books Online, and sample databases
Display	Super VGA (1,024x768) or higher-resolution video adapter and monitor	N/A

Visual Studio 2005 SQL CLR support

Visual Studio 2005 is the preferred development environment to create, debug, and deploy your SQL CLR routines in, however Visual Studio 2005 is not required. You could even use Notepad to create your source code, compile the source code with your managed language's corresponding command-line compiler, and then manually deploy it to SQL Server. If you do wish to use Visual Studio 2005, you will need

at least the Professional Edition of Visual Studio 2005 to be able to create, deploy, and debug your SQL CLR objects in Visual Studio. Thus, you need either Visual Studio Professional Edition, Visual Studio Tools for Office, or Visual Studio Team System. We will be using the Professional Edition of Visual Studio 2005 for the creation of our code samples in this book. The following are the requirements for Visual Studio 2005 Professional Edition:

Resource	Recommended
Processor	600 mHz
Operating System	Windows XP with Service Pack 2 or later Windows XP Professional x64 Edition (WOW) Windows Server 2003 with Service Pack 1 Windows Server 2003 x64 Edition (WOW) Windows Server 2003R2 Windows Server 2003R2 x64 Edition (WOW) Windows Vista
Memory	192 MB of RAM or more
Hard Disk	2 GB available
Drive	DVD-ROM drive

Required Namespaces for SQL CLR Objects

There are four namespaces required to support the creation of SQL CLR objects. The required namespaces are:

- `System.Data`
- `System.Data.Sql`
- `System.Data.SqlTypes`
- `Microsoft.SqlServer.Server`

All of these namespaces physically reside in the `System.Data assembly`. `System.Data` is part of the BCL of the .NET Framework and resides in both the Global Assembly Cache (GAC) as well as in the .NET 2 Framework directory: `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll`.

> `System.Data` *is automatically referenced for you when using Visual Studio 2005 for creating your SQL CLR objects. Also, the above-mentioned namespaces are automatically imported for you as well.*

Summary

In this introductory chapter, we have covered a lot of information that you need to comprehend before you begin creating your SQL CLR objects. You learned of the database objects you can create in the SQL CLR technology. We have shown the architecture of the .NET Framework and when it is being hosted by

SQL Server. We have addressed the biggest decisions developers and administrators have to make regarding SQL CLR. We also wanted to inform you that we are using SQL Server Express coupled with Visual Studio Professional for the production of this book's sample code, we choose these editions of the tools because they are the "lightest" editions of each product that support the SQL CLR technology, and we feel the vast majority of Microsoft developers use Visual Studio as opposed to the command-line compilers. Now that you have a strong foundation in how .NET and SQL CLR works, its time to begin creating your first SQL CLR objects. In Chapter 2, we are going to be covering how to create, deploy, and debug a managed stored procedure.