.NET and C# Binding Marketing Requirements Document (MRD)

Author/Owner:	Leon Guzenda, leon.guzenda@objectivity.com
Version/Date:	v2.6, 3-June-2005

Abstract:
Objectivity has C++, Java, Smalltalk, SQL++ and Python Application Programming Interfaces [APIs]. When Microsoft introduced their .NET framework for building XML and web based applications they also launched a new object oriented language called C#. The new language is a strong derivative of C and C++, with elements of Java and Borland's Delphi environment

Revision History

V1.0	First draft for comment, 17-Jan-2005 (Leon)
V2.0	Incorporating changes, 31-Jan-2005 (Leon)
V2.5	Coordinated with customer feature request and additional research, 12-May-2005 (Todd)
V2.6	Added schema phases and testing requirements. 3-June-2005 (Todd)

Table of Contents

1. Strategy and Overview

1. Goals and Objectives

This MRD addresses the need to incorporate the .NET platform (and the C# language) into Objectivity\DB. Size and growth of .NET depends on how you define what a deployed .NET application is. Excluding backwards compatibility to the Visual language families and .NET application connectivity (e.g. Office), C# represents a smaller market share than applications built on Java. However, the growth rate and eventual size may end up larger than Java. The base of “managed” applications that will eventually migrate is enormous.

2. Strategic Road Map

This MRD is part of the company’s overall plan to penetrate the .NET development community as well as provide an easily downloadable trial product. In addition, it helps us in our core customer base that has been waiting for .NET and C# support.

3. Customer Categories (User Profiles or Personas)

Target customers are expected to fall into the following categories and usage profiles:

3.1. **Current customer base**, looking for access to existing database using .NET or C#. CUNA Mutual, Emerson [Fischer Rosemount], and Siemens Building Technologies. Each of the above companies, as large Microsoft development shops, is representative of our customers that would have an interest.

3.2. **.NET and C# developers interested in Objectivity\DB**, who will be most interested in using our database because of its architectural advantages, but from .NET based web services or from the C# language. This group is presently thought to be small.

3.3. **Internet Downloader's**, having a .NET friendly downloadable product that plugs seamlessly into the Visual Studio development environment could increase our presence in database development projects on the Microsoft platform.

4. Competitive Strengths and Weaknesses

Our competitors have addressed the C# market. The leading RDBMS vendors:, Oracle, IBM, and Microsoft, have suitable .NET, web service, or C# connectivity. Note, RDBMS still have the same limitations and advantages. Versant has both a .NET binding that modifies CIL to provide persistence and a persistence by inheritance model. Messing with CIL has the same disadvantages\advantages as modified object code or bytecode (difficult to debug, unexpected behavior, bloat, etc). db4Objects seems to have converted their pure Java implementation's bytecode into CIL. The bytecode to CIL could have performance problems and bloat issues. Progress Software (ObjectStore) has .NET built into their messaging product but not a native C# interface.

2. Internally Committed Requirements

Within this section, each item is ranked, higher priority coming first.

2.1. Elimination of High-Priority Bugs

No System problem Reports [SPRs] have been identified that prevent the development of a .NET binding.

2.2. Internal Performance Improvements

No bottlenecks have been reported that would jeopardize the .NET binding. Implementation choices may vary in performance (e.g. C# wrappers versus bytecode translation).

2.3. Backward Compatibility

The .NET binding will be built on the Objectivity\DB 9.0 release. Therefore database compatibility will be that of 9.0.

2.4. Architectural Changes

The latest version of AMS uses .NET. It may be desirable to implement the interface to the Lock Server using .NET protocols. There will be a new language interface layer to support the C# API.

2.5. Platforms and Protocols

The .NET binding will support WinNT, Win2k, WinXP, and Windows 2003 Server. Windows 64 bit can be made available based on customer demand. Microsoft .NET Framework 1.1 and Visual Studio will be the development platform initially. However, the .NET Framework 2.0 and Visual Studio 2005 represent such large changes (analog JDK 1.x to 2.0), that movement will be made to those environments as soon as it becomes possible.

2.6. Uptime and Quality of Service

Uptime and QoS will meet the same demands of other Objectivity/DB bindings. Objectivity/HA availability in C# has not been determined.

3. Externally Committed Requirements

This section, discusses customer visible aspects of the .NET\C# Objectivity binding and feature requests . Because of the collaborative nature of the .NET\C# binding development (in conjunction with partners), implementation choices will also be presented. Our partner Viech (nowa part of GE Medical) will influence implementation in so far as it meets their requirements. To some degree, advantages, disadvantages, and perceived complexity of each approach will be included.

3.1. Architectural Choices

There are several architectural choices for adding object persistence to the .NET platform. Adding persistence can be implemented by modifying compiled C# (CIL), adding bolt-on web services that respond to queries\serialization requests, or building a robust C# API that wraps calls to the managed extensions to the C++ Objectivity\DB engine (w/ optional dynamic schema support).

Adding persistence by modifying CIL comes in two forms. The first is to modify the CIL produced after the programmer compiles his source with persistent calls. The second is producing a CIL persistence layer from translated bytecode. The translated CIL is then compiled down into a DLL that can be used directly by .NET applications. The translated CIL assemblies are referenced by .NET applications and Objectivity persistent API classes can be used as if they were .NET classes. Persistence by CIL modification has been ruled out as a valid approach because generally such a method of persistence is difficult to debug, performs poorly, and tends to increase the memory footprint of the application unnecessarily (bloat).

The bolt-on web service approach to .NET platform persistence has also been eliminated. Requests for object serialization and query requests tend to work for small datasets or simple queries. Additionally such a method of persistence is not conducive to the retrieval of large composite objects, fast throughput, or to the navigational power of object databases.

The best approach .NET persistence is a C# API that wraps calls to the managed C++ Objectivity/DB engine (with optional dynamic schema support). Such an approach gives the programmer flexible control over the persistence of the application and allows specific native types of the language to be persistent (e.g. collections could be templates in C++, generics in Java, or Jagged Arrays in .NET).

Specific advantages of using a managed C++ (MC++) extension to Objectivity\DB:

- The best performance of generated IL code because of both optimizations of the generated IL and less IL generated. This is specifically because MC++ is the only .NET compiler with a full optimizer back end, which is pretty much the same one that is used by the unmanaged compiler.

- MC++ is your language of choice if you want full control of the .NET environment:
  - Allows one to use all seven levels of CTS member access. C# allows only six.
  - Allows direct access to interior GC pointers, useful in a whole class of system applications such as system and .NET utilities.
  - Offers explicit control of expensive operations like boxing.
  - Supports multiple indexed properties on a type, unlike C#.
- MC++ is currently the only managed language that allows you to mix unmanaged and managed code, even in the same file. This leads to several other points:
  - Allows a developer to keep performance-critical portions of the code in native code.
  - Gives seamless access to all unmanaged libraries, such as DLLs, statically-linked libraries, COM objects, template libraries, and more.
  - Leverages existing investments in C++ programming skills and legacy C++ code.
  - Porting unmanaged code to .NET: MC++ allows you to take existing unmanaged code and compile it to managed code (with the /clr compiler switch and IJW).
  - Gives the ability to port code at one's own rate rather than re-write all at once.
  - Provides the easiest way to add .NET support to your existing native C++ Windows applications, by allowing you to bridge the gap between the two environments with as little work on your behalf as possible, and with the lowest performance penalty.
- MC++ is currently the only language that allows some form of multi-paradigm design and development with full support for generic programming and templates. This can lead to more options and better designs and implementations.

Disadvantages of Managed C++ extensions to Objectivity\DB:

- Managed C++ code is non-verifiable, since C++ can perform unsafe operations. The implication of this is that MC++ code may not run in restricted environments that will not run code that is non-verifiable.
- Some minor features of the .NET platform are not supported yet.

Another practical advantage of using MC++ is that it has already been reviewed and tested by Objectivity engineering. Moreover, there are presently large sized customer implementations using Objectivity C++ as a managed extension.

With late binding interpreted languages like Java, Smalltalk, and Python, persistent objects are discovered (converted to database schema) and retrieved as persistent objects without the need of a preprocessing step. Early binding statically compiled languages like C++, however, require a preprocessing step were the object is inspected and stored as schema. Therefore, to handle schema in C#, a schema discovery layer must be implemented in addition to C# wrappers to the general Objectivity API.

3.2. Schema Discovery Layer

The schema discovery layer will probably evolve through three phases:

1. We do the schema in C++ and then treat it as any other managed code.

2. We do the equivalent in C# of RTTI/Reflection and then create the schema using Active Schema (maybe wrapping AS API first in C#). Error out of the situation of pre-existing schema or class names.

3. Build in advanced capabilities, such as handling existing schemas, evolution, conversion, etc. and dealing with wrapping handles (automate it or hide within the design).

3.3. Quality Assurance

Presently the testing of the C# | .NET binding will modeled or ported from the Java binding. The thought is that this binding is the most robust, modern, and relevant to a language interface like C#. Standardized testing frameworks like JUnit (C# equivalent?) should be considered.

3.4. API Coverage

The Objectivity API that is supported for the .NET\C# platform could include the entire primitive types of .NET / C#, Objectivity/DB data types, and the Objectivity application programming interface. The .NET / C# primitives will not be discussed until further evaluation (testing and programming is required). The Objectivity\DB data types are any type supported by the schema. The schema of a federated database specifies the Objectivity/DB type of every attribute of every persistence-capable class whose objects can be stored in the federated database. When an application reads or writes a persistent object, Objectivity/DB automatically maps data between its own data types and the data types native to the application. Objectivity/DB types fit the general categories:

- Data in the Federated Database
- Object Identity
- Missing Data
- Objectivity/DB Primitive Types
- Object-Reference Types
- Embedded-Class Types
- String Classes
- Date and Time Classes
- Array Classes
- Scalable and Non-scalable persistent collections

The API support, using the Objectivity for Java interface as an analog, would include all of the classes in com.objy.db. The methods for those classes can be found in the "Methods Index" in the Objectivity for Java Programmer's Guide.

3.5. Product Feature Support

In addition to the general database API, Objectivity includes add-on packages for enhanced features. Therefore, the .NET\C# binding may include feature and support the following packages:

- Active Schema
- In-Process Lockserver [IPLS]
- SQL++
- High Availability (including Data Replication and Fault Tolerance)

3.6. Customer Feature Requests (Redacted)

Redacted modeled their .NET API feature requests on the JDO interface (but for .NET). We are not verbally or contractually required to implement these features. Some of these requests would be implicit in our binding across languages; some are more difficult (e.g. automatic remote-ing). The requested features are as follows:

- Persistence by reachability from root objects.
- Manage object identity and instance uniqueness.
- Automatically fetching data when it is accessed via a native object reference.
- Automatically flushing dirty object state to the data store when the transaction is committed.
- Automatic acquisition of appropriate locks on objects when their state is referenced or modified in a transaction.
- Automatically hollowing the state of a stale object in the CLR instance when a transaction boundary is crossed.
- Invocation of user-implemented callbacks on persistent objects for lifecycle events (fault, flush, hollow).
- Traversal of user-specified fetch groups when fetching object state.

Lower priority:

- Automatically marking objects dirty when their state is modified (so their persistent state will be updated when the transaction is committed).
- Providing an optimistic transaction policy that detects conflicts at commit time.
- Ability to access already-fetched data outside a transaction (granting that it may be stale).
- Ability to fetch data outside a transaction (granting that it may be stale).
- Automatic garbage collection of any persistent objects that are not reachable from root objects or referenced by any clients.

System requirements would include:

- Ability to operate in single-user mode (with optimizations).
- Optional use of a large shared cache on any machine hosting multiple clients (in addition to session-specific caches). Redacted clients concurrently access a lot of the same data.
- Optional object-level locking (as opposed to page- or higher-level), especially when using optimistic transactions.
- Automatic recovery from temporary network failure between client and server.
- Support for 64-bit Windows.

Multi-tier application server and Synchronization:

Redacted also requested stronger support for multi-tier architectures. This requirement should be handled in a separate MRD, but it is recorded here for completeness. It is not an essential requirement for an initial .NET/C# implementation and some of the requests would be better handled by an application or by the Parallel Query Engine.

Code could be executed wherever is most efficient, whether on the end user's computer or on the data server. That is, develop app server-like functionality with processes running on the data server. So, what would be called the database "client" is now the app server process running on the data server, while the new client is a separate process running on a remote user's computer. The app server will be built from the same code base as the client, all in C#. It would be really, really great if Objectivity provided a tool for the implementation of such three-tier architectures (a fairly common problem). This tool would essentially extend the reach of Objectivity .NET binding as follows:

- *Provide a transparent mechanism to replicate .NET object state from one CLR instance to another (in a master-slave relationship, where the app server process is the master and the client process the slave, from a persistence standpoint). Automatically replicate persistent objects to the client as needed due to the traversal of native object references. Manage object identity and instance uniqueness.*
- *Automatically synchronize the state of corresponding objects between the CLR instances.*
- *Provide a seamless way to "remote" object behavior between CLR instances so that data-intensive operations can be performed close to the data store and small result sets replicated back to the client.*
- *Allow automatic garbage collection of object in the master CLR instance when the corresponding replica is no longer referenced in the slave.*

Logging and event notification:

- *Logging of important operational information*
- *Complete documentation of log messages. Documentation of corrective and preventative actions for error messages.*
- *Ability to automatically send alerts (e.g., via email) in response to error events.*