ObjectiveView

Objects and Component Journal for the Software Professional

Flash Flowers III – by G. Gerard

FOCUS ON .NET

C#.NET
VB.NET
Managed C++ under .NET

ALSO

EXTREME PROGRAMMING REFACTORED

Interview with Doug Rosenberg and Matthew Stephens on their objections to XP

Published by

RATIO

OO Consultancy – Training – Development –Recruitment
See http://www.ratio.co.uk for back copies

ObjectiveView

CONTENTS

CONTACTS

Editor

Mark Collins-Cope
markcc@ratio.co.uk

Free subscription

Email delivery:
objective.view@ratio.co.uk
(subject: subscribe)

Hardcopy delivery:
objective.view.hardcopy@ratio.co.uk
(include full contact details)

Feedback/Comments/Article submission

objective.view.editorial@ratio.co.uk
Or join ObjectiveView @ Yahoo.com

Fresh Fuel for the Flaming Fires – Managed C++

Microsoft's .NET provides a multi-language component inter-operability environment – but as always supporting C++ has proved a challenge. RICHARD VAUGHAN takes a look at the differences between Managed C++ and the base C++ standard…

Introduction

Those of you with culinary leanings may have noticed a recipe for Warm Spider Crab Linguine in the March 2002 edition of the BBC's Good Food magazine. The instructions for eviscerating a Spider Crab read like a cross between The Spanish Inquisition's Staff Handbook and the memoirs of Joseph Mengele, and one gets a similar feeling upon consulting the Managed C++ Reference on the Microsoft web site. Many changes have been made to traditional C++, in the name of .NET, in order to bring us this variant on the standard and this article examines the major thrust of the offering.

Considering .NET generally, there are some eyebrow-raising claims being made for the technology. The following from the Wrox Press .NET support site is an example:

'*This idea of making programming much much easier by putting the really complex stuff "under the hood" while giving the programmer an intuitive and easy interface to work with is leading to the possibility of non-computer professionals taking up the task of authoring computer programs for their non-information technology knowledge specialties (medicine, civil engineering, etc.). Time will tell if we are at a turning point in the history of electronic digital information technology in which the numbers of persons capable of competently authoring a computer program increases by one or two or more orders of magnitude.*'

This is worrying because precisely the same claims were made for COBOL when that was introduced and (as ever) Object Orientation too. However, the development of non-trivial software is complex and challenging because the systems that we attempt to model on our machines are themselves very complex and challenging. To apply the logic espoused above to other areas of human pursuit is to imply that one day everybody could be a brain surgeon. Yet no amount of nifty tools (and languages are tools as well) will ever deliver us from the conservation of complexity.

Modus Operandi

While there are superficial similarities between the Java model and .NET, they are, in fact, fundamentally different approaches. In the case of Java, high-level source code is compiled into virtual-machine instructions. This allows a single executable to run on any machine, given that a VM is available for that machine. .NET parallels Java, in that high-level source code is compiled into Microsoft Intermediate Language (MSIL), which is similar to Java byte code, however from here the technologies diverge.

Firstly, the intermediate language (IL) does not run on a VM but is Just In Time compiled (jitted) to native machine code prior to execution on the target platform. (The Jitting process being a temporal equivalent to the spatial solution that a VM represents.) The second difference is that IL code can be generated from a variety of common programming languages. (Note that while it is also possible to compile Java byte code from non-Java sources, Sun did not intend this.)

However, all programming languages are not the same and .NET stipulates the Common Language Specification (CLS) to which existing languages must be bent in order to comply with the .NET framework. In essence, the CLS is to these as the XML Infoset is to XML syntax. That is to say XML documents are composed, semantically, of elements but other syntax could be used in place of angle brackets - XML documents could, hypothetically, be coded using C-like syntax.

Given this, we are not actually dealing with cross-language programming, where object code from different compilers is linked to a common executable, but cross-syntax programming. The syntax may appear to be that of your favourite programming language (C++, Eiffel, Smalltalk etc.) but in reality you are conforming to CLS semantics. Given that one is coding to a single underlying object model, this will necessarily place restrictions on the syntactic model that one is using.

Managed code is compiled into 'assemblies', which contain procedural code, versioning information and metadata that describes the types used in the assembly. While traditional C++ development is possible with Visual Studio .NET, sources can be compiled into IL by using the /clr compiler switch. However, this does not automatically transform traditional C++ syntax into a .NET assembly. Firstly, you must include the following at the top of a C++ source file:

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

The `#using` pre-processor directive makes the resources in the core library assembly available to code in the translation unit. The other `using` directive makes the .NET class libraries available. That, however, is only the start as MC++ code differs significantly from unmanaged C++. Let's explore those features and differences.

The MC++ Lexicon

MC++ adds fourteen new keywords to C++ which must be used to qualify otherwise normal C++ syntax. These are:

```
__abstract
__box
__delegate
__event
__gc
__identifier
__interface
__noqc
__pin
__property
__sealed
__try_cast
__typeof
__value
```

Some of these are fairly simple and innocuous in their effect, as in the case of the `__identifier` keyword. This allows one to use other language types where the names of those types are normally C++ keywords. For example, you may wish to use a class called 'switch', which has been declared using another language and which would normally constitute a syntax error were one to use it as the name of a C++ class. Others however have far more profound implications and this is partly because of the .NET dynamic-allocation model, which we shall examine next.

Storage Management

Given that .NET is about interoperability, all .NET applications will hold (at least some) objects in a common heap. It is the characteristics of this that feed back into C++ to yield MC++. The CLR heap is a garbage-collected, compacting heap. Garbage collection absolves the application of the responsibility of deallocating unwanted objects, while 'compacting' means that free space between allocated blocks is coalesced into a single block. This is accomplished by shifting blocks towards the beginning of the storage arena; a technique that was introduced in the storage management policies of early operating systems. **Figure 1** shows the deallocation of an object (Object 2), the gap that it leaves and the subsequent compaction of the free space that remains.

Garbage collection has the advantage that it prevents deallocated storage from being re-accessed, thus precluding rogue pointers. Memory leaks are also impossible, because unreachable objects are recovered automatically, plus it frees the developer from managing object lifetimes. This does incur a cost however because the collector runs as a background thread, which impinges on performance, plus the compaction process incurs a time penalty.

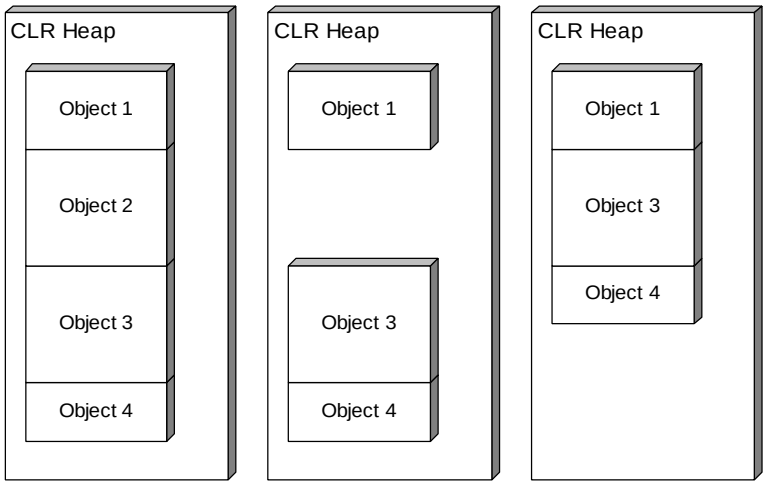


Figure 1 – memory allocation and compaction under .NET

All of this has implications for MC++ because the normal C++ storage management model conflicts with the CLS semantics demanded by .NET. Type declarations that are compiled for the .NET framework must therefore be adorned with MC++ keywords, which makes them known to the CLR.

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

__gc class Simple
{
public:
   int i;
};

int main ()
{
   // Runs forever
   while (true)
   {
      Simple *SimplePtr = new Simple;
   }
   return 0;
}
```

The above code fragment shows a simple managed C++ class, where the `__gc` keyword means that instances of the 'Simple' class are garbage collected. Were this unmanaged C++ then the loop would eventually exhaust the free store because each object that is created would be lost upon the next iteration. Instead the heap is never depleted because the garbage collector recovers the unreachable objects.

Clearly this has a systemic effect on other aspects of normal C++ and this is demonstrated in **figure 2**. Here we can see a set of objects that reside in the managed heap. The first points to the third, which is also pointed to by a stack-based pointer in the application's runtime space. The fourth is pointed to by an object, which resides in the application's non-managed free store.

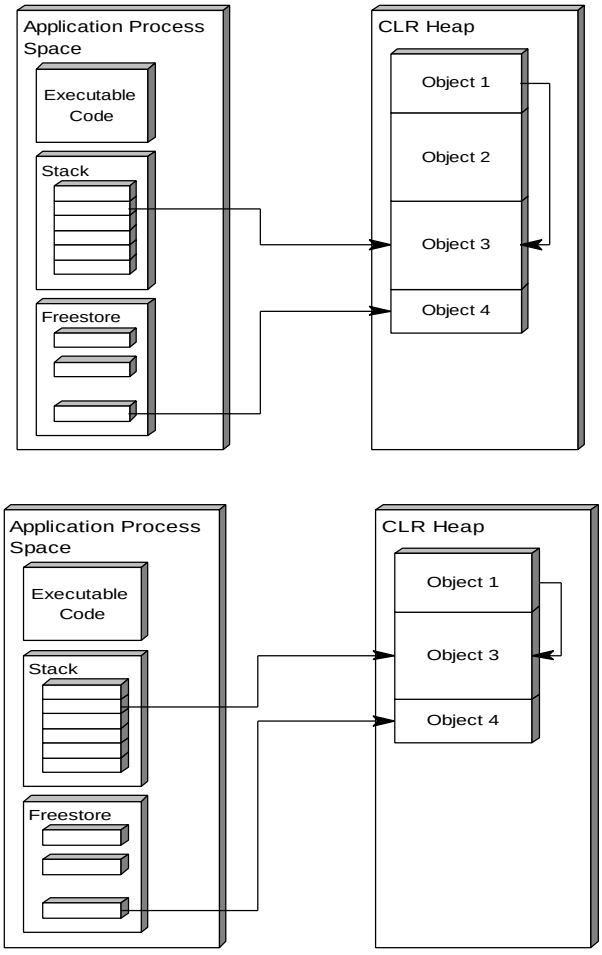


Figure 2 – Pointers in C++ and their relationship to the CLR Heap

Given that compaction necessarily changes the address of an object then this means that any pointers to the object must also be updated accordingly. Given this, any pointer to a CLR heap-object is not an ordinary pointer, as the compactor must be aware of it for it to be updated upon compaction. Therefore MC++ pointers must be declared with the __gc keyword as well as the objects that they point to. Note that the same holds for references as well. Note also that a given object's address can be fixed (thus preventing full compaction) using the __pin keyword.

The code below shows a class that contains a __gc pointer to objects of class Simple:

```
#using <mscorlib.dll>

using namespace System;

__gc class Simple
{
public:
   int i;
};

__gc class OtherSimple
{
   public:
   __gc Simple *SimplePtr;
};
```

In addition to the above, stack-based objects of user-defined type are disallowed (every instance of a __gc class must be dynamically allocated). This in turn means that __gc objects cannot be passed or returned by value. The following example demonstrates this:

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

__gc class Simple
{
public:
   int i;
};

void SomeFunc (Simple SimpleParameter){};

int main ()
{
   Simple InstanceOfSimple;
   // Will not compile
   Simple __gc *SimplePtr = new Simple;
   // Fine
   SomeFunc (InstanceOfSimple);
   // Will not compile
   return 0;
}
```

There is, however, an alternative to __gc classes and that is the concept of __value classes. The __value keyword is intended for small objects that generally have short lifetimes and for which garbage collection would be too costly. As opposed to __gc classes, __value classes can be instantiated on the stack and can be passed by value to functions. They cannot however be allocated on the CLR heap unless they are embedded within a __gc class. For example:

```
__value struct Simple { int i; };
```

```
__gc class Enclosing {
   Simple SimpleMember;
};

Simple SomeFunc (Simple SimpleParam)
// Passed by value
{
   SimpleParam.i += 1;
   // Value at call site unaffected
   return SimpleParam;
   // Returned by value
}

int main ()
{
   Simple SimpleInst_1 = {10};
   Simple SimpleInst_2 =
      SomeFunc(SimpleInst_1);

   Enclosing _gc *EnclosingPtr =
      new Enclosing;
   // Allocated as part of Enclosing
   // instance

   EnclosingPtr->SimpleMember =
      SimpleInst_1;   // Copy value
   EnclosingPtr->SimpleMember.i +=
      SimpleInst_2.i;
   Console::WriteLine (SimpleInst_1.i);
   Console::WriteLine (SimpleInst_2.i);
   Console::WriteLine
      (EnclosingPtr->SimpleMember.i);
}
```

```
Output:

10
11
21
```

This may appear to return us to more familiar unmanaged C++ territory, however it creates a new problem in that one must choose whether a class is going to be __gc or __value - A type cannot be both. Note also that while one can still dynamically allocate a __value object, this can only occur on the normal C++ free store and you must qualify operator new with the __nogc keyword to achieve this.

Finally, there is another issue lurking behind all of this. Studies indicate that there is no single, optimum, storage-allocation policy, yet .NET takes a 'one size fits all' approach, which cannot suit every application. Traditional C++ gives the ability to customise allocation strategies on a per-class basis, thus yielding up opportunities for significant performance optimisations. Overloads of new are not allowed in __gc classes however and this restriction could compromise many applications.

Storage management aside, let us examine the inheritance model in MC++.

Inheritance

Once again, and in order to get all .NET languages singing from the same hymn sheet, the MC++ inheritance model differs significantly from traditional C++. Firstly, if a class has no super class then it is

implicitly derived from System::Object. This is to provide compatibility with the C# libraries where everything is derived, cosmically, from a single, universal base. Secondly, all inheritance must be public; private and protected inheritance is not allowed. In addition to this, managed classes cannot declare friend classes or functions.

Fourth, one cannot mix `__gc` and `__nogc` classes in an inheritance relationship although it is possible to mix `__gc` classes with classes declared using other languages. For example a C# class can form the base for an MC++ class and that class could then form the base for an Eiffel# class.

Implementation inheritance is available in the single form only - multiple inheritance being available only for interfaces. i.e. a class can have many base classes but only one of these may have attributes; the rest must be populated with member functions alone. How troublesome this is depends upon your view of multiple inheritance. Some believe that full-blown MI, as supported by C++, Eiffel etc. is an entirely good and desirable thing and that for it to be curtailed in .NET is to deny developers their freedom. Meyer, for one, takes this view and feels that .NET's restrictions on MI should (and eventually will) be lifted.

In reality, multiple implementation-inheritance will always incur a small performance penalty and can introduce name conflicts. Moreover, the same effect can always be accomplished using a mix of Is A and Has A relationships with no greater loss of performance. Moreover, as Alexandrescu points out, MI is a simply a syntactic mechanism for combining classes and does not implicitly orchestrate the collection of bases – This is something that the derived class must be made to do. Many developers will therefore be unperturbed by the restrictions that .NET imposes in terms of building new applications from scratch, although it does have significant implications for porting existing systems because a proportion of their code will not compile.

Aside from restrictions on the normal C++ inheritance model, what additions does .NET make in MC++? Firstly, the subclassing of a type can be mandated by the use of the `__abstract` keyword, although an `__abstract` class is not required to have pure virtual functions. Secondly, `__interface` classes can be declared where no data members are allowed (apart from a `__value enum`) and all member functions are implicitly pure virtual. Subclassing of `__interface` classes is therefore also mandatory. **Figure 3** illustrates many of the above points:

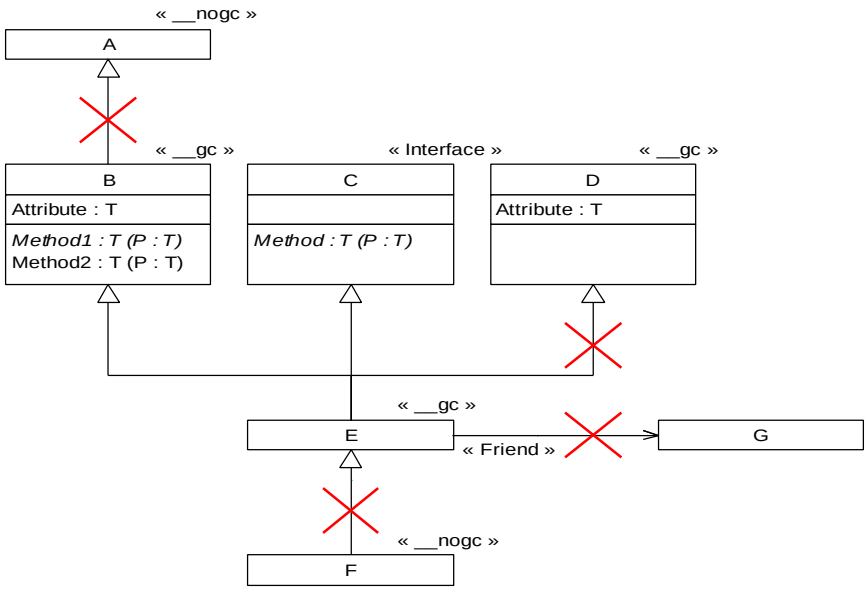


Figure 3 – Summary of inheritance issues in Managed C++

In contrast with the `__interface` and `__abstract` classes, further derivation can be prevented by use of the `__sealed` keyword, although this cannot be applied to an `__abstract` or `__interface` class. `__sealed` can also be applied to virtual member functions thus preventing them from

being overridden. Note that value classes are `__sealed` implicitly.

Member Functions

Member functions also differ in a number of ways from unmanaged C++. Firstly, the garbage collection issue holds important implications for constructors and destructors. One of the great things about these is that they are an excellent way of ensuring that something happens automatically. In the case of destructors, this is true even when an exception is raised and this lends itself to a variety of attractive and useful techniques. However, with garbage collection, destructors become finalisers, which means that they are called just before the garbage collector reclaims the storage, not when the object goes out of scope. Given that one cannot normally guarantee when garbage-collection will occur, it is therefore impossible to know precisely when the finaliser will be called and thus a useful technique is lost.

It is possible to work around this by calling operator delete on a `__gc` object, which will force execution of the destructor there and then. (The CLS guarantees that the destructor will not be called again when the garbage collector subsequently recovers the object.) Alternatively, you can call the destructor explicitly, as in unmanaged C++, and this would appear to reinstate the technique outlined above. In reality, you now have to think explicitly about the death of the object, whereas the original idea was to let the language rules and the compiler take care of everything.

There are other changes to construction semantics to consider as well, such as the fact that user-defined destructors are always virtual. Other changes, however, are more radical. For example, in traditional C++, a call to a virtual function from within a base-class constructor will result in execution of the override that is visible at that class's position in the hierarchy. This occurs irrespective of any overrides that are present in derived classes. In MC++, however, the same call will cause execution of the most overriding version of the function to be executed. Note that member objects are zero initialised before the execution of the constructor and the overriding function is therefore guaranteed not to execute on an object that contains garbage, (although you may not actually want the defaults either).

In addition to this, managed types cannot have a user-defined copy constructor. This implies that copying an object that has a pointer to another dynamically allocated object will result in the CLR creating a copy of that object as well, and so on recursively. (Although the Microsoft reference does not seem to mention this.)

Functions with default arguments are not permitted therefore one must use a wrapping function to achieve the same effect. I.e. for a function with a single default argument one must create a new function that takes the original's non-default arguments as tramp variables and which calls that function supplying a hard-coded default parameter.

Operator Overloading

Operator overloading has also felt .NET' s sweet caress and, while it is (mostly) still possible, the `operator` keyword cannot be used. For example the following constitutes a syntax error:

```
__gc class Simple
{
public:
   bool operator == (Simple &RHS);
   // Error
   // Note: const cannot
   // be used
};
```

Instead one must used a static function with a distinguished name and operator overloads in `__gc` classes must have at least one parameter that is a pointer to the defining class. This function can then be called explicitly by quoting the distinguished name, or it can be invoked implicitly using conventional infix notation. For example, the equality operator is overloaded using `op_Equality`:

```
__gc class Simple
{
public:
   static bool op_Equality (Simple *LHS,
Simple &RHS) {...}
};

bool Compare (Simple P1, Simple P2)
{

   if ( Simple::op_Equality (&P1, P2) )
      return false;  // Explicit
   if ( P1 == P2 )
      return false;  // Implicit
   return true;
};
```

The following is a list of the operators that can be overloaded in MC++ and their conventional symbols

```
Unary operators

op_Decrement                    --
op_Increment                    ++
op_Negation                     !
op_UnaryNegation                -
op_UnaryPlus                    +

Binary operators

op_Addition                     +
op_Assign                       =
op_BitwiseAnd                   &
op_BitwiseOr                    |
op_Division                     /
op_Equality                     ==
op_ExclusiveOr                  ^
op_GreaterThan                  >
```

```
op_GreaterThanOrEqual           >=
op_Inequality                   !=
op_LeftShift                    <<
op_LessThan                     <
op_LessThanOrEqual              <=
op_LogicalAnd                   &&
op_LogicalOr                    ||
op_Modulus                      %
op_Multiply                     *
op_RightShift                   >>
op_Subtraction                  -
```

Note that `__gc` classes cannot overload the address-of operator.

Serious Restrictions

Some features of traditional C++ are preserved in MC++. For example, exception handling is still implemented using the normal try/throw/catch mechanism. Alternatively, one can use Structured Exception Handling by means of the `__try`, `__throw` and `__catch` keywords. SEH also adds the `__finally` keyword, which marks code that will be executed following the throwing of an exception but before the exception object leaves the throw site. MC++ also adds the `__try_cast` keyword, which is similar in operation to the `dynamic_cast` mechanism in normal C++ and is used within a try block (an exception is raised if the cast cannot be performed).

However, there are features of C++ that are simply not supported in MC++ because they would conflict with the CLS. The most onerous of these is the rough treatment that templates receive. While it is possible to instantiate a template with a managed type, templates cannot have a managed type as a parameter type and, worse of all, there is no such thing as a managed template. For example the following code is not allowed:

```
template <typename T> __gc class
SimpleTemplate  // Error
{ … };
```

Given that genericity is not supported then there can be no support for cross-language genericity. For example you cannot use a C++ template in a C# assembly nor can you cannot derive a C# class from a C++ template. The official word from Microsoft is that they are not supported in this release, however for various technical reasons, it is hard to see how they could be supported even in future releases. This will create problems for the adoption of .NET into the C++ community because a lot of very powerful techniques will be impossible – A managed version of the STL is out of the question, for example. Certainly the suppression of templates in MC++ refutes the idea that .NET signals the end of C++ because the need for templates and the STL is too strong to overcome by other existing means.

In addition, RTTI cannot be used. In itself this may not be so bad because the CLR supports a much richer reflection model but it does impinge on porting existing code. A trivial bit of editing and some recompilation will not suffice and many points in existing systems have to be rewritten.

Finally, `_gc` classes cannot have `const` member functions. Nor is `volatile` permitted either. While `volatile` is not such a problem as its role has always been much more in low-level systems development (something that .NET, by definition, detracts from), the lack of `const` is pretty serious because it denies us the benefits of const correctness as denies optimisation opportunities to the compiler.

Conclusion

Managed C++ cannot be covered exhaustively here and, for example, Delegates, Events and Properties (a formalisation of accessor/mutator functions) have not been explored, nor has ‘Boxing’ received an airing. However, core issues are clear. Firstly, anyone who thinks that one can develop for the .NET platform in C++ as if nothing had happened is mistaken - Syntax aside, the semantics are, in places, wildly different. Given this, the term ‘Managed’ in MC++ will be viewed by many as a euphemism for ‘Bastardised’ … or perhaps worse. With MC++ (or any other .NET language) one is actually working with C# semantics (I.e. the CLS), therefore systems that are developed from scratch may as well be coded in C# from the start. In that way you can at least enjoy clean syntax that is devoid of the extra (messy) keywords mandated by MC++.

Secondly, Microsoft cannot walk away from C++ and the principle reason, therefore, for them to support it on .NET is to allow the porting of existing code to the platform. It also wishes (political interests to the fore) to create a sense of ‘old favourite on new platform’. It therefore positions things by saying that you can create managed code that bridges between existing (legacy, in their terms) code and other .NET components - unmanaged code can be moved over incrementally.

Some of the overviews of .NET support this by implying that there is virtually seamless interoperability between C++ and MC++. However, many existing applications will not port by the simple addition of a few keywords, because of semantic impedance. Given this, fresh design-decisions will have to be made or a wrapping approach will have to be taken (same thing), and this can be done in two ways:

a. Embed unmanaged classes within managed classes
b. Use managed pointers to managed objects, which wrap unmanaged objects

However, these workarounds only serve to complicate an already complicated business - good software development seeks to simplify and MC++ runs contrary to this.

Considering .NET generally, there is another issue, which is the 'cultural' impedance that exists between developers with different language-experience. The experience gap between same-language developers creates sufficient difficulties as it is, while language-choice can influence the way that developers think about a problem. .NET combines these factors, so that we end up with heterogeneity in both syntax *and* conception, and this can only add fresh fuel to the language wars - To unify semantics whilst dividing syntax sounds like a recipe for disaster.

These negative conclusions aside, there is a thin silver-lining to all this, which is the vicarious benefit that comes from understanding the compilation issues that apply as a whole to C++/C#/Java etc. and the .NET runtime. The John Gough book mentioned in the bibliography is thoroughly recommended for those who wish to plumb these fascinating depths.

Bibliography

- Compiling for the .NET Common Language Runtime (CLR), John Gough, Prentice Hall, ISBN 0 13 062296 6
- The Bertrand Meyer .NET Video Course
- Modern C++ Design: Generic Programming and Design Patterns Applied, Alexei Alexandrescu, Addison Wesley, ISBN 0 201 70431 5
- www.microsoft.com
- BBC Good Food Magazine - March 2002, BBC Worldwide Publishing

Author Biography

RICHARD VAUGHAN is a software development consultant and lecturer with over 20 years of software development experience. He can be contacted at richardv@ratio.co.uk. He delivers a number of Ratio training courses including:

- **C++,**
- **High performance C++,**
- **Managed C++ under .NET,**
- **OOA/D using UML**.

See back page (40) for further details.

Enterprise UML 2003

Le Meridien Palace Hotel, Manchester - UK May 20 & 21, 2003

An ELM Conference

Enterprise UML is the UK's premier annual UML conference

Featuring Case Studies from Lockheed Martin, Metropolitan Police and Tesco

This annual conference primarily showcases the use of UML in a wide variety of contexts. Dr. Richard Soley, Chairman & CEO of OMG delivers the Keynote address on the future for UML and Steve Cook of IBM provides an overview of the UML 2.0 metamodel and the process employed in its definition.

Topics featured in the conference include UML and Web Services, OMG's Model Driven Architecture (MDA), UML and Agile processes, building Enterprise solutions with UML, Modelling with Templates in UML, UML and architectural layering, Intelligent Agents and UML Models, and UML and data integration.

This conference is aimed at senior developers, analysts and designers of object and component-based systems, software development managers, modelling professionals, consultants and software architects.

See http://www.ericleach.com/uml2003/ for full details

TO BOOK:

Telephone: +44 (0)20 8758 7587/7511

Fax: +44 (0)20 8758 7505

Email: eric_leach@compuserve.com

RATIO

Recruitment Services from the OO experts

Ratio Group are the acknowledged leaders in OO in the U.K.

- We are not an agency and our non-agency approach to our recruitment service means we actually understand the roles that we are asked to provide candidates for,
- We are more than qualified (and we do!) to pre-interview every candidate that we might put forward, we won't send you hundreds of CVs, but every candidate we do send you will have been pre-interviewed by us and will be eminently suitable for the role.

If you have a OO vacancy you need to fill call us on 020 8579 7900 or email us at info@ratio.co.uk

Vacancies from the OO experts

We frequently have vacancies for OO experts both internally and for our clients, and on a contract or permanent basis. People we are looking for now include

- OO Lecturers/Consultants in London and Southeast. Skills needed include OO/UML/Java/C++/.NET
- Software developers Java, C++, min 3 years software development experience

To apply for any role offered by Ratio email your CV to jobs@ratio.co.uk

Sharp intake of C – A C# Overview

C# is one of the family of languages Microsoft has designed to be part of its .NET framework. In this article author JON JAGGER gives a comprehensive introduction to C#.

Introduction

C# is a new language from Microsoft supported under its .NET platform. .NET is Microsoft's implementation of the Common Language Infrastructure (CLI) ECMA standard – a specification jointly submitted by Microsoft, Intel, and Hewlett-Packard.

The CLI is designed for strongly typed languages and has 5 partitions. Partition 1 specifies the CLI foundation: the Common Type System (CTS), the Virtual Execution System (VES), and the Common Language Specification (CLS).

Compiling a C# program does not create a native executable. Instead it creates a program in Common Intermediate Language (CIL, specified in partition 3 of the CLI). A compiled C# program also contains a block of metadata (data about the program itself) called a manifest (specified in partition 2). This metadata facilitates powerful reflection capabilities.

The VES translates the CIL into native executable code (which can be done just-in-time or at installation). The CTS is a set of *types* to designed to allow language interoperability. All CTS types are either value types or reference types.

The CLS is a set of *rules* designed to allow language interoperability. For example, unsigned integer types are not in the CLS so your C# programs must not expose unsigned integers if you want them to be fully interoperable.

Hello World

The obligatory console Hello World in C# looks like this:

```
class HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine
           ("Hello, world!");
    }
}
```

C# has a sensibly limited preprocessor. There are no macro functions. What you see is what you get. A C# source file is not required to have the same name as the class it contains. Identifiers should follow the camelCasing or PascalCasing notation depending on whether they are private or non-private respectively. Hungarian notation is officially *not* recommended.

C# is a case sensitive language so Main must be spelled with a capital M. A C# program exposing two identifiers differing only in case is not CLS compliant. The CLS supports exception handling and C# accesses these features using the try/catch/finally keywords.

Exceptions are used extensively in the SDK framework classes. C# also supports C++ like namespaces as a purely a logical scoping/naming mechanism. You can write using directives to bring the typenames in a namespace into scope.

```
using System; // System.Exception

class HelloWorld
{
    static void Main()
    {
        try {
            NotMain()
        }
        catch (Exception caught) {
            ...
        }
    }
    ...
}
```

C# Fundamentals

type	bits	CLS?	signed	sig figs
byte	8	yes	no	
ushort	16	no	no	
uint	32	no	no	
ulong	64	no	no	
sbyte	8	no	yes	
short	16	yes	yes	
int	32	yes	yes	
long	64	yes	yes	
float	32	yes		7
double	64	yes		15
decimal	128	yes		28

Figure 1 – Overview of C# types

Numeric Types

C# supports 8 integer types (not all of which are CLS compliant) and three floating point types – see figure 1. The floating point literal suffixes for these three types are F/f, D/d, and M/m (think m for money).

C# expressions follow the standard C/C++/Java rules of precedence and associativity. As in Java, the order of operand evaluation is left to right (in C/C++ it's unspecified), an expression must have a side effect (in C/C++ it needn't) and a variable can only be used once it has definitely been assigned (not true in C/C++).

Checked Arithmetic

The CLS allows expressions or statements that contain integer arithmetic to be checked to detect integer overflow. C# uses the checked and unchecked keywords to access this feature. An integer overflow throws an OverflowException when checked. (Integer division by zero *always* throws a DivideByZeroException.) Floating point expressions never throw exceptions (except when being cast to integers). For example:

```
class Overflow
{
   static void Main()
   {
      try {
         int x = int.MaxValue + 1;
         // wraps to int.MinValue

         int y = checked
            (int.MaxValue + 1);
            // throws
      }
      catch (OverflowException caught){
         Console.WriteLine(oe);
      }
   }
}
```

Control Flow

C# supports the if/while/for/do statements familiar to C/C++/Java programmers. As in Java, a C# boolean expression must be a genuine boolean expression. There are *never* any conversions from a built in type to true/false. A variable introduced in a for statement initialization is scoped to that for statement. C# supports a foreach statement which you can use to effortlessly iterate through an array (or any type that supports the correct interface).

```
class Foreach
{
   static void Main(string[] args)
   {
      foreach (string arg in args) {
         System.Console.WriteLine(arg);
      }
   }
}
```

The C# switch statement does not allow fall-through behavior. Every case section (including the optional default section) must end in a break statement, a return statement, a throw statement, or a goto statement. You are only allowed to switch on integral types, bools, chars, strings and enums (these types all have a literal syntax).

Methods and Parameters

C# does not allow global methods; all methods must be declared a struct or a class. C# does not have a C/C++ header/source file separation; all methods must be declared inline. Arguments can be passed to methods in three different ways:

- *copy*. The parameter is a *copy* of the argument. The argument must be definitely assigned. The method cannot modify the argument.
- *out*. The parameter is an *alias* for the argument. The argument need not be definitely assigned. The method must definitely assign the parameter/argument.
- *ref*. The parameter is again an *alias* for the argument. The argument must be definitely assigned. The method is not required to assign the parameter/argument.

The ref/out keywords must appear on the method declaration *and* the method call. For example:

```
class Calling
{
    static void Copies  (int param)
    {...}
    static void Modifies( out int param)
    {... }
    static void Accesses( ref int param)
    {...}

    static void Main()
    {
        int arg = 42;

        Copies (arg) ;
        // arg won't change

        Modifies(out arg);
        // arg will change

        Accesses(ref arg);
        // arg might change
    }
}
```

C# supports method overloading but not return type covariance. Unlike Java, C# does *not* support method throw specifications (all exceptions are effectively unchecked).

Value Types

C# makes a clear distinction between value types and reference types. Value type instances (values) live on the stack and are used directly whereas reference type instances (objects) live on the heap and are used indirectly. C# has excellent language support for

declaring user-defined value types (unlike Java which has none).

Enums and Structs

You can declare enum types in C#. For example:

```
enum Suit
   { Hearts, Clubs, Diamonds, Spades }
```

You can also declare a user-defined value type using the struct keyword. For example:

```
struct CoOrdinate
{
    int x, y;
}
```

Unlike C++, the default accessibility of struct fields is private. You control the initialization of struct values using constructors. You use the static keyword to declare shared methods and shared fields. The readonly keyword is used for fields that can't be modified and are initialised at runtime. The const keyword is used for fields (and local variables) that can't be modified and are initialised at compile time (and is therefore restricted to enums and built in types). As in Java, each declaration must repeat its access specifier.

```
struct CoOrdinate
{
    public CoOrdinate
      (int initialX, initialY)
    {
        x = rangeCheckedX(initialX);
        y = rangeCheckedY(initialY);
    }
    public const int MaxX = 600;
    public static readonly CoOrdinate
      Empty = new  CoOrdinate(0, 0);
    ...
    private int x, y;
}
```

The built in value type keywords are in fact just a notational convenience. The keyword int (for example) is an alias for System.Int32, a struct called Int32 that lives in the System namespace. Whether you use int or System.Int32 in a C# program makes no difference.

Operator Overloading

C# supports operator overloading. Enum types automatically support most operators but struct types do not. For example, to allow struct values to be compared for equality/inequality you must write == and != operators:

```
struct CoOrdinate
{
    public static bool operator==
      (CoOrdinate lhs, CoOrdinate rhs)
    {
        return lhs.x == rhs.x &&
           lhs.y == rhs.y;
    }

    public static bool operator!=
      (CoOrdinate lhs, CoOrdinate rhs)
    {
        return !(lhs == rhs);
    }
    ...
    private int x, y;
}
```

Operators must be public static methods (so polymorphism is never an issue). Operator parameters can only be passed by copy (no ref or out parameters). One or more of the operator parameter types must be of the containing type so you can't change the meaning of the built in operators. The increment (and decrement) operator can be overloaded and works correctly when used in either prefix and postfix form. C# also supports conversion operators which must be declared using the implicit or explicit keyword. Some operators (such as simple assignment) cannot be overloaded.

Properties

Rather than using a Java Bean like naming convention, C# uses properties to declare read/write access to a logical field without breaking encapsulation. Properties contain only get and set accessors. The get accessor is automatically called in a read context and the set accessor is automatically called in a write context. For example (note the x and X case difference):

```
struct CoOrdinate {
   ...
   public int X
   {
      get {
         return x;
      }
      set {
         x = rangeCheckedX(value);
      }
   }
   ...
   private static int rangeCheckedX (int arg)
   {
      if (arg < 0||arg > MaxX) {
         throw new ArgumentOutOfRange(
            "x"
         );
      }
      return argument;
    }
    ...
    private int x, y;
}
```

Indexers

An indexer is an operator like way to allow a user-defined type to be used as an array. An indexer, like a property, can contain only get/set accessors. For example:

```
struct Matrix
{
    ...
    public double this [ int x, int y ]
    {
        get {
            ...
        }
        set {
```

```
        }   ...
    }
    public Row this [ int x ]
    {
        get {
            ...
        }
        set {
             ...
        }
    }
    ...
}
```

Reference Types

Classes

Classes allow you to create user-defined reference types. One or more reference type variables can easily refer to the same object. A variable whose declared type is a class can be assigned to null to signify that the reference does not refer to an object (struct variables cannot be assigned to null). Assignment to null counts as a Definite Assignment. Classes can declare constructors, destructors, fields, properties, indexers, and operators. Despite identical syntax, classes and structs have subtly different rules and semantics. For example, you can declare a parameterless constructor in a class but not in a struct. You can initialise fields declared in a class at their point of declaration, but fields declared in a struct can only be initialized inside a constructor. Here is a class called MyForm that implements the GUI equivalent of Hello World in C#.NET.

```
using System.Windows.Forms;

class Launch
{
   static void Main()
   {
      Application.Run(new MyForm());
   }
}

class MyForm : Form
{
   public MyForm()
   {
      Text = captionText;
   }
   private string captionText =
      "Hello, world!";
}
```

Variables whose declared type is a class can be passed by copy, by ref, and by out exactly as before.

```
class WrappedInt
{
   public WrappedInt(int initialValue)
   {
      value = initialValue;
   }
   ...
   private int value;
}

class Calling
{
   static void Copies  (WrappedInt param)
      { ... }
   static void Modifies(
      out WrappedInt param
   ) { ... }

   static void Accesses(
      ref WrappedInt param
   ) { ... }

   static void Main()
   {
      WrappedInt arg = new  WrappedInt(42);
      Copies  (arg);  // arg won't change
      Modifies(out arg); //arg'll change
      Accesses(ref arg); //arg may change
   }
}
```

Strings

C# string literals are double quote delimited (char literals are single quote delimited). Strings are reference types so it is easy for two or more string variables to refer to the same string object. The keyword string is an alias for the System.String class in exactly the same way that int is an alias for the System.Int32 struct.

```
namespace System
{
    public sealed class String : ... {
        ...
        public static bool operator==(
          string lhs, string rhs
        ) { ... }
        public static bool operator!=(
           string lhs, string rhs
        ) { ... }
        ...
        public int Length { get { ... } }
        public char this[int index]
           { get { ... } }
        ...
        public CharEnumerator
           GetEumerator() { ... }
   }
}
```

The String class supports a readonly indexer (it contains a get accessor but no set accessor). The C# string type is an immutable type (just like in Java). The string equality and inequality operators are overloaded but the relational operators (< <= > >=) are not. The StringBuilder class is the mutable companion to string and lives in the System.Text namespace. You can iterate through a string expression using a foreach statement.

Arrays

C# arrays are reference types. The size of the array is not part of the array type. Rectangular arrays of any rank can be declared (unlike Java which only supports one dimensional rectangular arrays).

```
        int[]  row;
        int[,] grid;
```

Array instances are created using the new keyword. Array elements are default initialised to zero (enums

and numeric types), false (bool), or null (reference types).

```
        row  = new int[42];
        grid = new int[9,6];
```

Array instances can be initialised:

```
int[] row = new int[4]{ 1, 2, 3, 4 };
      // longhand

int[] row = { 1, 2, 3, 4 };
      // shorthand

      row = new int[4]{ 1, 2, 3, 4 };
      // okay

      row = { 1, 2, 3, 4 };
      // compile time error
```

Array indexes start at zero and all array accesses are bounds checked (IndexOutOfRangeException). All arrays implicitly inherit from the System.Array class. This class brings array types into the CLR and provides some handy properties and methods:

```
namespace System
{
    public abstract class Array : ...
    {
        ...
        public int Length { get { ... } }
        public int Rank { get { ... } }
        public int GetLength(int rank)
           { ... }
        public virtual IEnumerator
           GetEnumerator() { ... }
        ...
    }
}
```

The element type of an array can itself be an array creating a so called "ragged" array. Ragged arrays are *not* CLS compliant. You can use a foreach statement to iterate through a ragged array or through a rectangular array of any rank:

```
class ArrayIteration
{
   static void Main()
   {
      int[] row = { 1, 2, 3, 4 };
      foreach (int number in row) {
            ...
      }

      int[,] grid = {{ 1, 2 }, { 3, 4 }};
      foreach (int number in grid) {
            ...
      }

      int[][] ragged =
      {
         new int[2]{1,2},
         new int[4]{3,4,5,6}
      };

      foreach (int[] array in ragged) {
         foreach (int number in array) {
               ...
         }
      }
   }
}
```

Boxing

An object reference can be initialised with a value. This does not create a reference referring into the stack (which is just as well!). Instead the CLR makes a copy of the value on the heap and the reference refers to this copy. The copy is created using a plain bitwise copy (guaranteed to never throw an exception). This is called boxing. Extracting a boxed value back into a local value is called unboxing and requires an explicit cast. When unboxing the CLR checks if the boxed value has the exact type specified in the cast (conversions are not considered). If not, the CLR throws an InvalidCastException. C# uses boxing as part of the params mechanism to create type safe variadic methods (methods that can accept a variable number of arguments of any type).

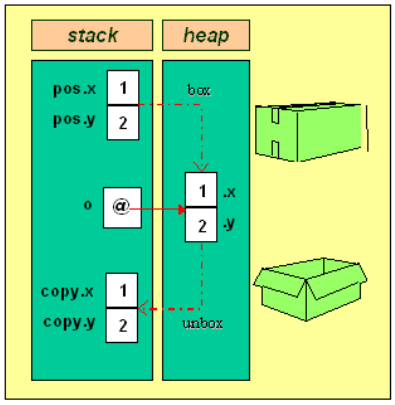


Figure 2 – Boxing in C#

```
struct CoOrdinate
{
   ...
   private int x, y;
}
class Boxing
{
   static void Main()
   {
      CoOrdinate pos ;
      pos.X = 1;
      pos.Y = 2;
      object o = pos;  // boxes
      ...
      CoOrdinate copy = (CoOrdinate )o;
      // cast to unbox
   }
}
```

Type Relationships

Inheritance

C# supports the same single inheritance model as Java; a class can extend at most one other class (in fact a class always extends exactly one class since all classes implicitly extend System.Object). A struct cannot act as a base type or be derived from. A derived class can access non-private members of its immediate base class

using the base keyword. Unlike Java (and like C++) by default C# methods, indexers, properties, and events are *not* virtual. The virtual keyword specifies the *first* implementation. The override keyword specifies *another* implementation. The sealed override combination specifies the *last* implementation.

```
class Token
{
    ...
    public virtual CoOrdinate Location
    {
        get {
            ...
        }
    }
}

class LiteralToken : Token
{
    ...
    public LiteralToken(string symbol)
    {
        ...
    }

    public override CoOrdinate Location
    {
        get {
            ...
        }
    }
}

class StringLiteralToken : LiteralToken
{
    ...
    public StringLiteralToken(
      string symbol
      ) : base(symbol)
    {
        ...
    }

    public sealed override CoOrdinate
    Location
    {
        get {... }
    }
}
```

Interfaces

C# interfaces contain only the names of methods. Methods bodies are not allowed. Access modifiers are not allowed (all methods are implicitly public). Fields are not allowed (not even static ones). Static methods are not allowed (so no operators). Nested types are not allowed. Properties, indexers, and events (again with no bodies) are allowed though. An interface, struct, or class can have as many base interfaces as it likes.

```
interface IToken
{
    ...
    CoOrdinate Location  { get; }
}
```

A struct or class must implement all its inherited interface methods. Interface methods can be implemented implicitly or explicitly.

```
class LiteralToken : IToken
{
    ...
    public CoOrdinate Location
    // implicit implementation
    {
        get {
            ...
        }
    }
}

class LiteralToken : IToken
{
    ...
    CoOrdinate IToken.Location
    // explicit implementation
    {
        get {
            ...
        }
    }
}
```

You use the abstract keyword to declare an abstract class or an abstract method (only abstract classes can declare abstract methods). You use the sealed keyword to declare a class that cannot be derived from. The inheritance notation is positional; base class first, followed by base interfaces.

```
interface IToken
{
    ...
    CoOrdinate Location  { get; }
}

abstract class DefaultToken
{
    ...
    protected DefaultToken(
       CoOrdinate where
    )
    {
        location = where;
    }
    public CoOrdinate Location
    {
        get {
            return location;
        }
    }
    private readonly CoOrdinate location;
}

sealed class StringLiteralToken :
DefaultToken, IToken
{
    ...
}
```

Runtime type information is available via the is, as, and typeof keywords as well as the object.GetType() method.

Resource Management

You can declare a destructor in a class. A C# destructor has the same name as its class, prefixed with a tilde (~). A destructor is not allowed an access modifier or any parameters. The compiler converts your destructor into an override of the object.Finalize method. For example, this:

```
public class StreamWriter : TextReader
{
```

```
    ~StreamWriter()
    {
        Close();
    }
    public override void Close()
    {
        ...
    }
}
```

is converted into this: (You can use the ILDASM tool to see this transformation in CIL.)

```
public class StreamWriter : TextReader
{
    ...
    protected override void Finalize()
    {
        try {
            Close();
        }
        finally {
            base.Finalize();
        }
    }
    public override void Close()
    {
        ...
    }
}
```

You are not allowed to call a destructor or the Finalize method in code. Instead, the generational garbage collector (which is part of the CLR) calls Finalize on objects sometime after they become unreachable but definitely before the program ends. You can force a garbage collection using the System.GC.Collect() method. C# does not support struct destructors (although CIL does). However, C# does have a using statement which you can use to scope a resource to a local block in an exception safe way. For example, this:

```
class Example
{
   void Method(string path)
   {
      using (LocalStreamWriter exSafe =
         new StreamWriter(path))
      {
          StreamWriter writer =
             exSafe.StreamWriter;
          ...
      }
   }
}
```

is automatically translated into this:

```
class Example
{
   void Method(string path)
   {
      {
         LocalStreamWriter exSafe =
            new StreamWriter(path);
         try {
            StreamWriter writer =
               exSafe.StreamWriter;
            ...
         }
         finally {
            exSafe.Dispose();
         }
      }
   }
}
```

which relies on LocalStreamWriter implementing the System.IDisposable interface:

```
public struct LocalStreamWriter :
   IDisposable
{
   public LocalStreamWriter(
     StreamWriter decorated
   )
   {
       local = decorated;
   }
   public static implicit operator
     LocalStreamWriter(
      StreamWriter decorated
     )
   {
      return new LocalStreamWriter(
         decorated
      );
   }
   public StreamWriter StreamWriter
   {
      get { return local; }
   }
   void IDisposable.Dispose()
   {
      local.Close();
   }
   private readonly StreamWriter local;
}
```

Applications

Delegates and Events

The delegate is the last C# type. A delegate is a named method signature (similar to a function pointer in C/C++). For example, the System namespace declares a delegate called EventHandler that's used extensively in the Windows.Forms classes:

```
namespace System
{
    public delegate void EventHandler
      (object sender, EventArgs sent);
    ...
}
```

EventHandler is a now a reference type you can use as a field, a parameter, or a local variable. Calling a delegate calls all the delegate instances attached to it.

```
namespace Not.System.Windows.Forms
{
    public class Button
    {   ...
        public EventHandler Click;
        ...
        protected void OnClick(
           EventArgs sent
        )
        {
            if (Click != null) {
                Click(this, sent);
               // call here
            }
        }
    }
```

```
}
```

All delegate types implicitly derive from the System.Delegate class. You use the event keyword to modify the declaration of a delegate field. Event delegates can only be used in restricted, safe ways (for example, you can't call the delegate from outside its class):

```
namespace System.Windows.Forms
{
    public class Button
    {
        ...
        public event EventHandler Click;
    }
}
```

You create an instance of a delegate type by naming a method with a matching signature and you attach a delegate instance to a matching field using the += operator.

```
using System.Windows.Forms;

class MyForm : Form
{
    ...
    private void initializeComponent()
    {
        ...
        okButton = new Button("OK");
        okButton.Click += new
           EventHandler(this.okClick);
          // create + attach
    }
    private void okClick(
       object sender, EventArgs sent
    )
    {
        ...
    }
    ...
    private Button okButton;
}
```

Assemblies

You can compile a working set of source files (all written in the same supported language) into a .NET module. For example, using the C# command line compiler:

```
csc /target:module /out:ratio.netmodule *.cs
```

The default file extension for a .NET module is .netmodule. A .NET module contains types and CIL instructions directly and forms the smallest unit of dynamic download. However, a .NET module cannot be run. The only thing you can do with a .NET module is add it to an assembly. An assembly contains a manifest (a module does not). The manifest is metadata that describes the contents of the assembly and makes the assembly self describing. An assembly knows:

- the assembly identity
- any referenced assemblies
- any referenced modules
- types and CIL code held directly
- security permissions
- resources (eg bitmaps, icons)

You create a .NET DLL (an assembly) using the /target:library option from the command line compiler (there are various other options for adding modules and referencing other assemblies):

```
csc /target:library /out:ratio.dll *.cs
```

You create a .NET EXE (an executable assembly) using the /target:exe options on the command line compiler (one of the structs/classes must contain a Main method).

```
csc /target:exe /out:ratio.exe *.cs
```

Assemblies comes in two forms. A private assembly is not versioned, and is used only by a single application. A shared assembly is versioned, and lives in a special shared directory called the Global Assembly Cache (GAC). Shared assembly version numbers are created using an IP like numbering scheme:

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

Shared applications that differ only by version number can co-exist in the GAC (this is called side-by-side execution). The particular version of an assembly that an individual application uses when running can be controlled from an XML file. For example:

```
…
<BindingPolicy>
    <BindingRedir Name="ratio" ...
                  Version="*"
                  VersionNew="6.1.1212.14"

UseLatestBuildRevision="no"/>
</BindingPolicy>
...
```

You can edit this config file to choose your binding policy. For example:

- Safe: exactly as built
- Default: major.minor as built
- Specific: major.minor as specified.

Attributes

You use attributes to tag code elements with declarative information. This information is added to the metadata, and can be queried and acted upon at translation/run time using reflection. For example, you use the [Conditional] attribute to tag methods you want removed from the release build (calls to conditional methods are also removed):

```
using System.Diagnostics;
```

```
class Trace
{
    [Conditional("DEBUG")]
    public static void Write(string message)
    { ... }
}
```

You use the [CLSCompliant] attribute to declare (or check) that a source file conforms to the Common Language Specification:

```
using System;

[assembly:CLSCompliant(true)]
...
```

You can use the [MethodImpl] attribute to synchronize a method:

```
using System.Runtime.CompilerServices;

class Example
{

[MethodImpl(MethodImplOptions.Synchronized)]
    void SynchronizedMethod()
    {
        ...
    }
}
```

The attribute mechanism is extensible; you can easily create and use your own attribute types:

```
public sealed class DeveloperAttribute :
Attribute
{
    public DeveloperAttribute(string name)
    {
        ...
    }
}
…
[Developer("Jon Jagger")]
public struct LocalStreamWriter : IDisposable
{
    ...
}
```

Summary

C# programs compile into Common Intermediate Language (CIL). CIL types that conform to the CLS (Common Language Specification) can be used by any .NET language. For example, the types in the System namespace are implemented in the mscorlib.dll assembly. Programs written in C#, in VB.NET, or in managed C++, can all use this assembly (there isn't one version of the assembly for each language).

CIL programs are translated into executable programs either at installation time or just-in-time as they are executed by the VES (Virtual Execution System). The CLI (Common Language Infrastructure – the CTS, the VES, the CLS, and the metadata specification) is an ECMA standard and efforts are already underway to implement the CLI on non Windows platforms (eg http://www.go-mono.com).

C# is a modern general purpose programming language. It has clear similarities to Java (reference types, inheritance model, garbage collection) and to C++ (value types, operator overloading, logical namespaces, by default methods are not virtual). It has no backward compatibility constraints (as C++ does to C) and avoids/resolves known problems in Java. The CTS (Common Type System) makes a clear distinction between value types and reference types. The more I use C# the more I like it and the more I appreciate the careful and consistent decisions taken during its design. C# is my language of choice for .NET development. In roughly keeping to the allotted word count I have necessarily omitted numerous important aspects of C#. Nevertheless I hope this article has given you a flavour of C# and its relationship to .NET.

Author Biography

JON JAGGER is the co-author of Microsoft Press's **Microsoft Visual C# .NET Step by Step**. Jon wrote and delivers the Ratio Group **C# course**, and also lectures in **OOA/D Using UML** for Ratio. He can be reached at jonj@ratio.co.uk.

Pearson Education books related to topics in this issue of ObjectiveView

From Java to C#

From Java to C#: A Developer's Guide enables you to use your existing knowledge of object-oriented concepts to learn C# efficiently and quickly. Features of C# that are totally absent in Java are given the detailed description they warrant.

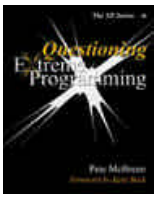
Use Case Driven Object Modeling with UML

Applied Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example provides a practical, hands-on guide to putting use case methods to work in real-world situations.

Extreme Programming Explained

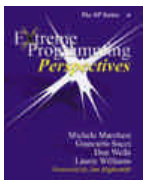
The book that started it all off.

Questioning Extreme Programming



After reading this thought-provoking book, software developers can make informed decisions about Extreme Programming, and whether it is suitable for their organization. Readers will also be able to determine whether Extreme Programming is inappropriate for a particular project. The author challenges you to look past the hype and start asking the hard questions about how software is built

Extreme Programming Perspectives



Extreme Programming Perspectives presents 47 articles that represent the insights and practical wisdom of the leaders of the XP community. Encompassing a wide variety of key topics on XP and other agile methodologies, this book offers experience-based techniques for implementing XP effectively and provides successful transitioning strategies.

See also - A Pattern Language for Web Usability

A Pattern Language for Web Usability is a practical guide for web designers and managers of website development projects and can be used as a simple checklist to aid the design process and ensure that websites are usable and successful.

Go to www.ratio.co.uk/bookstore.html to purchase these books at a discount.

Interview - Refactoring Extreme Programming


Matt

MARK COLLINS-COPE interviews DOUG ROSENBERG (author Use Case Driven Modeling with UML) and MATT STEPHENS (co-author with Doug of the book XP Refactored - due for release in Summer 2003), on their objections to Extreme Programming.


Doug

[Mark Collins-Cope] Hello Matt and Doug,
{Doug Rosenberg and Matt Stephens] Hello Mark,

General

[Mark] Doug, you've been vocal in your opposition to XP for some years yet, so I don't think it'll come as a great surprise to many people you're authoring a book called XP Refactored. As a starting point for this discussion, could you summarise the message of the book for us, then perhaps we can delve into the issues you raise in more detail.

[Doug] To start with, the complete title (which is "XP Refactored – The Case Against Extreme Programming") will give you an idea of what Matt and I are doing. By way of background, I got engaged in the XP debate originally on OTUG (the Object Technology User Group), mostly with Bob Martin and Ron Jeffries a few years ago (this culminated in quite an intense discussion about the Chrysler C3 project, and whether its cancellation meant success (as they claimed) or failure (as it seemed to many of us). Subsequently I've made a few attempts at satirical humor that have been pretty well received, notably "Alice in Use Case Land" which was a keynote speech I gave at UML World and recently repeated at the Rational User Conference, "Fragile Methods" which was a talk I gave at SD West, "Emperor's New Code", and then my son Rob and I have a lot of fun rewriting old Beatles songs, which we've compiled into "Songs of the Extremos."

I ran across Matt's article "The Case Against Extreme Programming", in which he systematically picks apart much of the XP hype and circular logic, step-by-step, piece-by-piece, and often in a hysterically funny way, about a year or so ago. I was reading his discussion related to those who have never done pair programming having no right to say they wouldn't like it, when suddenly I read a line that started me laughing so hard it brought tears to my eyes, and I knew then we had to write a book together. You'll have to read the article to know the line that I mean, it starts out "I have never dipped my head in" Matt and I met earlier this year over in England, and I told him that writing that line was going to change his life. He told me he had agonized over whether he should pull it out of the article. Good thing he left it in.

*So the book is based around many of the points in Matt's "Case Against" article, with a whole chapter showing some pretty amazing hype vs. reality statements about the C3 project (to paraphrase Churchill**: "never has so much hype been achieved by so few over such a dismal failure"**), and we've tried to bring this off with as much humor as we can. Sometimes the humor is as simple as taking self-contradictory quotes from XP Gurus and placing them next to one another (sometimes the quotes seem really funny to us on their own like for example "schedule is the customer's problem", and "Extreme Programmers are not afraid of Oral Documentation"), and sometimes we make pretty intensive use of satire and sarcasm.*

But underneath it all we are exposing some very real weaknesses about XP, such as (to pick one that we've currently been discussing) the continuous delegation of responsibility for minor items like requirements management and project schedule away from the programming team, and onto the "customer". So the (overloaded) onsite customer becomes a single-point of failure, while the programmers just keep on refactoring away and go home at 5PM every day. Another big issue that we both have is the expressed and implied opinion that upfront thought in software development is somehow a waste of time, and that the lack of upfront thought is compensated for by short-iterations and refactoring. These are just a couple of issues we'll be addressing.

Process and notation

[Mark] I put this question to Robert Martin in a previous interview – I' m interested in your response… "Process is a hot topic in the software development community at present. We have RUP, Iconix, XP etc., all of which seem to offer something of a contradictory view of the world. What is the average software developer or manager supposed to make of this? Is there such a thing as a right and a wrong approach to software development?"

[Doug] Well, our friends at Rational would tell you that RUP's view of the world can be tailored to fit just about anything, and they have actually done some pretty interesting work in making the RUP easier to tailor. In fact, we've recently released an "ICONIX Process" plug-in for RUP, which essentially installs the approach I wrote about in "Use Case Driven Object Modeling"

into RUP for you in about 10 minutes (and deletes a lot of stuff out of the RUP installation) – we call it QuickStart.

Also, I believe ObjectMentor has recently produced a RUP plugin for XP. The concept of a RUP plugin for XP is very amusing to me. I am not certain about this but I think the XP plugin pretty much deletes all the analysis and design activities. I personally don't think there's a snowball's chance in hell of your average extreme programmer buying a copy of RUP, but the existence of an XP plugin for RUP serves to legitimize XP as a "real process" in the minds of some folks, I guess.

[Mark] So XP is now an instance of RUP, the ICONIX process is now an instance of RUP… seems like RUP is definitely all things to all men ☺! Can such a wide reaching process really be of value?

[Doug] Yeah…the whole world is an instance of RUP. Except that I don't think RUP is an instance of RUP. RUP is a process framework, not a process instance :) Seriously, though, we built our plugin because ICONIX Process is a streamlined path through use cases and UML, we've got a lot of clients that use Rose for UML, and we thought it would be beneficial to be able to "install" the process from "Use Case Driven Object Modeling" into a RUP installation. I think having a process that everyone on the team can access via a website is a useful thing, especially so on larger projects. Actually, I'd prefer to stop short of saying that our process is an instance of RUP – I'd say that you can install our process into RUP quickly and easily and sort of change RUP's personality with this plugin technology. It's a pretty cool concept really, you build a UML model of your process (mostly activity diagrams for the various steps), run it through the RUP Builder tools and it produces a file which you can install into RUP which changes what your RUP installation looks like. I think that the plugin concept is useful because you can set up YOUR process and use RUP as a vehicle to deploy it. So yes, I think there's value in there.

[Mark] The UML notations have become popular over the last few years, to the degree that there are now many tool vendors out there selling tools to support it. In some ways I can see XP is a reaction to increased marketing pressure from these companies – "you must have a tool to develop software." Isn't it true you can develop software quite adequately without highly expensive tools and that XP shows the way in doing this?

[Doug] UML doesn't necessarily require an expensive tool. We teach a lot of classes with Visio these days, for example. And I taught one class recently where people posted "sticky paper" all over the walls and drew UML diagrams on them with markers, then had one person scribe them all into Visio. It's the thinking that goes into the UML model before you jump into code that's really important. I don't think anyone will ever convince me that software development is more likely to be successful by eliminating forethought, although I have seen the argument made.

[Matt] The design process itself certainly doesn't need any software tools, expensive or otherwise. It does pay to write up the design afterwards though, because this in itself is a form of design review – an extended thought process. You might write up the design using Word and Visio, or Rational Rose, or whatever fits your company's budget. For example, I'm currently involved in a project where a lot of our design work is being carried out by sketching lines and boxes on whiteboards. Sometimes this is UML, other times it's ad-hoc notation. Whatever fits the thought process at the time. So paper and pen, or whiteboard and marker, are essential tools in the design process. But at some stage you need to use a software tool to document your design, so that other people can read, understand and review this thing that you're planning to implement.

In fact, I often think of better designs, or flaws in the current design, whilst documenting the design. There's something involved in this whole process that is effective in producing a good, simple design before you start writing production code. Sure, the design is under constant review whilst coding too – but that up-front design & review process saves us a lot of refactoring later. We can only do that with proper design documents – which don't take very long to write anyway. The trick is to document just enough to be able to move forward safely. The larger amount of time is spent coming up with the design, not documenting it.

[Mark] But isn't "just enough" what XP recommends as well?

[Matt] Unfortunately, "just enough" is quite a subjective measurement. Just enough for what? That all boils down to your own view of which practices (or inactions) increase software risk, which practices decrease it, and what the acceptable level of software risk should be. Documenting just for the sake of it increases risk because it's yet another document to maintain, and you might not really gain anything from doing it. When writing up your design, you need to have a set of guidelines in the back of your mind. For example, the Agile Modeling (AM) principle Model With a Purpose – if you do it right, you definitely get more out of documenting your design than what you put in. That same AM principle also recommends that you know your audience, so that you write up your design in the format that best suits them. This is where XP errs – their belief is that source code is the clearest expression of the design if your audience is a programmer. Sure, it's the most "truthful" documentation of a design, because the design "is" whatever the code does, but it may not be the most expressive. Also, who's to say that

your code is infallible? Just because the code does something, and passes a few unit tests, doesn't mean that this was what it's really supposed to be doing. Source code is too complex to be a clear statement of intent. We need something simpler than code and unit tests to define our design – a Word document, or a class model, for example.

[Mark] Isn't it true that some very good software engineers just aren't happy with graphical notations? Isn't XP a good solution for these people?

[Doug] I believe there are people who can write efficient code and don't like graphical notation. I would stop short of calling these people "good software engineers". I actually have an engineering degree (in Electrical Engineering). Nobody would ever have graduated from the EE program I went through if they couldn't read and write schematic diagrams. These diagrams are enormously useful in the communication of ideas about a system under development. Schematics and logic diagrams for circuits, UML for software. It's all part of the job.

[Matt] UML, or whatever graphical notation you're using, has its place even in an XP project. The difference is simply in the amount to which the graphical designs are utilised. Having said that, I find it inconceivable that someone who can't visualise designs could decide to become a programmer. The industry is moving ever further towards using modeling tools as a way of producing working code. CASE tools are becoming better at generating source code; TogetherJ has been generating Java class skeletons from UML for years. The difference now is that these generators, such as Sun's ACE project, are promising to create complete, working code from a graphical business spec. Enterprise tools such as Javelin Software's JGenerator are generating complete applications from textual notation already. The ironic thing is, these tools, which involve big up-front requirements gathering and big up-front design, are enabling projects to be much more agile, not less. This is because the generators encapsulate the entire architecture. Should I generate a dotnet solution? Hang on, I'll change this attribute and generate a J2EE solution instead… Okay, so we're probably not quite at <u>that</u> level of flexibility yet, but like it or not it's the way the industry is going! If a person can't visualise a design, they shouldn't be a programmer – and they definitely won't stand a chance in a few years' time…

Software Architecture

[Mark] One criticism you've levelled at XP is its concept of an emergent architecture – an architecture evolving over multiple iterations of a project. Isn't this a much more realistic view of architecture as it is used in practise?

[Doug] Not to me. Perhaps this relates to having some clients in the aerospace industry. I've worked on, trained people and provided tools to people on ballistic missile defense projects, the Space Station, the Hubble telescope, avionics systems, helicopter projects, military flight planning systems, command and control systems, etc. I was talking to a client of mine from a very large jet fighter project yesterday and I suggested they let the architecture of the fly-by-wire system evolve incrementally. We got a good laugh out of it. I think if people on these kinds of projects can manage to develop architectures and then design within the architecture, people doing business or e-commerce systems can, too.

I can see, on small projects, where letting some portions of the architecture evolve a bit during a series of incremental builds, might be a workable strategy, but if you read the Wiki Web you find a quote by Kent Beck suggesting that "the bigger the system the more you need emergent architecture". As with many such quotes, it just leaves me shaking my head.

[Matt] Emergent architecture can work, but as Doug suggests this is only for small-scale projects. You could perhaps get around this by dividing a project's architecture into smaller sub-architectures, then letting those evolve incrementally; but then there's not likely to be any real cohesion between the different parts of the project. Especially when multiple teams are involved, you end up with an "alphabet-Spaghetti" architecture where nothing quite fits together properly. You could counter this effect by adding in extra layers of process and documentation, but then you end up losing the supposed benefits of XP. It's much better to begin with a process that is designed to scale up in the first place.

[Mark] XP has the idea of a system metaphor to replace architecture. Is this not perhaps a better approach that rather ill defined "boxes and lines with pretty pictures of PCs approach" to architecture we see in being used in many organisations? Most of these type of diagrams have no meaning whatsoever.

[Doug] As I understand it, the XP system metaphor maps fairly closely to what we call a "domain model" – that is, you're identifying some of the more important nouns (conceptual objects) in the problem domain. Except a domain model (which is a simplified UML class diagram) is quite a bit more precise than a system metaphor. And the technical architecture tends to evolve out of this domain model but includes quite a bit more detail depending on things like the programming language, the GUI toolkit, the underlying database, the communication protocol, etc. etc. of the system.

[Matt] The System Metaphor is a useful thing to have, even in non-XP projects. It's quite loosely defined in the XP books, so you could (for example) apply the

Metaphor principles to domain modeling, to come up with a consistent set of object names and descriptions. If you're following the ICONIX process, you would be doing this anyway. All subsequent documentation and code is then written in the context of the domain model. On its own though, XP's Metaphor is simply not enough to guarantee consistency and cohesion across different parts of the project. For that you would need a proper design, one that's written down.

[Mark] But coming back to the point about many architecture "diagrams" being boxes and lines with little or no semantics, what are we to do about describing architecture (assuming we can actually agree what it is ☺)

[Doug] Well, for myself, I'm happy seeing the use cases, domain models, robustness diagrams and any other supplementary diagrams that the architects feel are useful. I have no problems with boxes and lines without precise semantics. Architects renderings of buildings don't have real precise semantics either, but I can appreciate their value.

Design

[Mark] 'Do the simplest think that could possibly work' is the design maxim of XP. Software engineers are renowned for over-engineering solutions – a solution looking for problem – and doesn't this XP maxim have an appropriate place in any software process?

[Matt] Simplicity rules. If only the designers of the EJB spec had thought about this some more. However, it's possible to take simplicity a bit too far. With XP, what you end up with is an evolutionary prototyping approach, where you are simply designing in what you think you need for the software to work right now. The problem is, prototypes tend towards the incomplete. To take a use case analogy, prototypes tend to be the "happy day" scenario. None of the difficult stuff, like what happens when things go wrong, what happens when the user makes a mistake or decides to do something valid but unexpected – typically 80% of an application – gets factored into the prototype. The difficult, hidden stuff – stuff that doesn't appear to be of value to the customer, but which will come back to bite you if you don't put it in – gets left out, because the team are just interested in getting something working right now and shipping it to the customer.

[Doug] Well, there are many variations on this theme. KISS (Keep It Simple, Stupid) is one of them. And keeping it simple is, of course, a good idea in most cases. But let's examine the phrase "the simplest thing that could possibly work". Think carefully about what this means. This phrase goes beyond normal "keep it simple" advice. There's a quote from an old "Distributed Computing" article about the Chrysler C3 project (this was one of the articles that put XP on the map) where the intent is stated more specifically: "We do not build generality into the system in expectation of future requirements. We build the simplest objects that can support the feature we're working on right now."

This is a very very different idea from just "keeping it simple". This says "don't think ahead about other requirements, just slap some code together for what you're building RIGHT NOW". This advice, in the hands of most programmers that I've ever met (and I earned my living writing code for about 15 years), can prove to be incredibly dangerous. What the Extemos ask you to believe is that "constant refactoring after programming" (do your own acronym) makes that all OK. It's OK to hack stuff together with rubber bands, bubblegum, and scotch tape today because you're going to refactor it tomorrow anyway. Uh-uh. Not for me.

What gets even more fascinating, is that this very same article claimed that C3, which was a Y2K project to replace Chrysler's mainframe payroll systems before Jan 2000, would be paying 86,000 employees at Chrysler by mid-1999. This was a somewhat audacious claim of success before success was actually achieved. In actual fact, when C3 was cancelled in Jan 2000, it was only paying 10,000 employees (the same as when the article had been written a year or so previously). I recommend that everybody read the page called CthreeProjectTerminated on the Wiki Web for themselves. Hopefully we can provide some URLs for your readers at the end of this interview.

So, in fact, "doing the simplest thing that could possibly work" didn't work. Doing the simplest thing that could possibly work did just about what you'd expect: it generated a quick illusion of success, which couldn't be sustained. What makes the XP phenomenon so amazing, is the success they had in creating the impression of success, when in fact it had not been achieved. We saw articles like Chet Hendrickson's "DaimlerChrysler: The Best Team in the World" appearing in IEEE Computer Oct. 1999 (this would be about 2 months before project cancellation). But where the XP hype machine truly outshines all competition is in managing to sustain this hype AFTER the project got cancelled. Matt found an article from "The Economist" published in December 2000 (almost a full year after cancellation) that cites C3 as a success, claiming that it was paying 86,000 employees at Chrysler, and attributing its success to Mr. Beck. We wonder about how stuff like that gets printed. Certainly when I got involved in the debate on OTUG (after reading about C3's cancellation on the Wiki Web), vociferous claims were still being made about what a success it was. And, of course, you know that most of the XP authors (Beck, Jeffries, Hendrickson, Wells, etc.) came from the C3 project.

[Mark] So you have no objection to simplicity, but you dislike the minimal window of lookahead in XP, is that right?

[Doug] I guess that's a fair statement. Simple designs are good. Turning a blind eye towards future requirements and reciting YAGNI instead of building infrastructure is not so good, as far as I am concerned.

[Mark] The XP approach to design is to let it emerge over time, based on refactoring to meet new requirements as they become available. XPers are particularly anti- Big Design Up Front (BDUF). Isn't this antagonism to BDUF justified – hasn't experience shown that spending ages on design upfront is mostly a waste of time as by the time implementation comes around, the requirements have changed and the design is no longer valid.

[Matt] Shifting requirements is a universal issue that is faced by both XP and non-XP projects. The solution is pretty much the same in both cases – you reduce the risk by breaking the project into smaller iterations. It reduces the "big bang" delivery that project managers and customers equally dread.

Shorter iterations also means that there's not too much of a BDUF feel, or a waterfall feel, to the whole process. Scrum has it about right with their 1-month Sprint. XP's 2-3 week iteration is pushing it a bit, but certainly isn't the worst aspect of XP.

Each deliverable may or may not be intended for end-users at that stage. If it is (which XP recommends), then the problem of course is that the project goes into maintenance mode very quickly, before maybe 10 or 20% of the overall functionality has been written. From that point on, inevitably development slows down, because you've got an installed user base, and letting the design emerge over time suddenly doesn't seem like such a good idea. Changes to the design, or to the overall architecture, have untold complications when you have users to support and retrain, legacy data to maintain, help files to update, data migration scripts to test. The best way to counter this problem is to get the design right up-front – or at least as right as you can get it. Sometimes there's no getting away from it: this involves thinking ahead, beyond the next increment. Getting it right means spending more time on it than XP recommends.

[Doug] My personal experience in developing software has always been that an hour spent in thinking about the problem upfront saved at least 3 hours in coding and testing. But it is also possible to carry upfront design to an extreme, which results in what we call "Analysis Paralysis". But it's actually pretty straightforward to do upfront analysis and design without catching a case of analysis paralysis. That's what "Use Case Driven Object Modeling" is all about. We've had many discussions on this, I think Matt has an acronym EDUF for Enough Design Up Front, which is what we're generally looking for. Rob and I have a song about the Extremo mindset here….it goes to the tune of "Nowhere Man" by the Beatles….and it starts out "Here's a cool project plan…..jump to code quick as you can….don't need no requirements from nobody"

[Mark] So how do you know when you've done "Enough Design Up Front?"

[Doug] Well, at the risk of repeating what I've written in my first two books; I want to know what the required behavior is for all the scenarios we're going to implement, I want to know what the domain objects are, I want a reasonable effort made to identify "rainy day scenarios", exceptional behavior, what we call 'alternate courses of action", I want to see the scenario descriptions validated with a robustness diagram, and then a sequence diagram that shows me how the objects communicate at runtime. It really isn't that imposing a list of things to do. I've been teaching people to do this for 10 years now. Generally, if you understand what you're going to build, you should be able to write a 2 paragraph use case description in 15 minutes or so, spend another 10 minutes checking that you've got it right with a robustness diagram, then maybe half an hour on a sequence diagram. So if you invest about an hour per scenario that you're going to build, the time payback is usually several multiples of that hour. Of course if you don't understand the behavior requirements, it takes longer because you have to find out what the behavior is supposed to be. But I'd rather know that before I've coded the thing up.

Refactoring

[Mark] Isn't it true that until XP came along – everyone simply ignored the fact that on any iterative/incremental style project (and during maintenance on waterfall style projects) refactoring is a necessary fact of life. Put another way, design is an optimisation problem whereby we come up with a solution that fits a particular set of requirements. Next iteration, we have new requirements and our previous design is going to be sub-optimal. We must refactor if we're going to stop software rot setting in – no?

Also, in most software engineering projects there is a learning process being undertaken by the team. They may be learning the domain, they may be learning new technology, and so on. Isn't refactoring important given this context – as we learn we realise we can do things in better ways – so shouldn't we update our code to reflect this?

[Doug] Refactoring definitely has its place. I'm not against refactoring in general, I'm against refactoring

as a replacement for upfront design. I'm against the mindset that we call "Constant Refactoring After Programming", wherein the goal is to get the simplest thing that can possibly work up and running ASAP, and then refactor the design into existence. Refactoring is not evil, but refactoring is a poor replacement for forethought.
[Mark] So you would agree that XP is the process that has put refactoring onto the software agenda – and as such it has made a valuable contribution?

[Doug] I'm quicker to agree with the first part of your sentence than the second part. I don't see the adoption of refactoring as a replacement for upfront design as a valuable contribution. I think if refactoring is used as a supplement to a reasonable amount of upfront design, then refactoring can be useful. I don't think XP uses refactoring that way, and so I regard XP's contribution in this area to be a net loss rather than a net gain. I do think XP has done something genuinely useful in putting unit testing on the map in a big way, though, if that helps at all.

Team and personnel issues

[Mark] Pair programming is another contentious aspect of XP. Aren't there substantial benefits to pair programming – in terms of transfer or knowledge and skills, in terms of "the whole is greater than the sum of the parts"

[Doug] I don't have a problem with people working in pairs if they voluntarily want to work that way. So I'd never prohibit pair programming. But having worked as a programmer for 15 years, I place a pretty high value on peace, quiet, and space to think in. I once saw a study from, I think it was IBM Santa Teresa labs, where they found that putting programmers in private offices with doors that closed was a huge boost to productivity over cubicles. So the idea of all the programmers in a big room seems like it would be a huge detriment to productivity, to me. I think it would drive me nuts, personally.

I've also worked with some incredibly talented programmers who would have been awful to pair with. One in particular comes to mind who was hypoglycemic and had a tendency to get upset very easily when his blood sugar dropped. He's a brilliant programmer and produces prodigious amounts of code, which is always efficient and well structured. He and I worked very well together but I would never dream of sharing a desk and a keyboard with him. And I'd resist any process that wouldn't allow me, as a manager, to make use of his skills. So I would never mandate pair programming.

And of course, my big issue with XP's approach is that pair programming is used as an excuse for not doing upfront design. That's completely bogus, in my opinion.

[Mark] The XP approach to roles (architect, lead designer, etc) is to either not have them, or to allow the roles to emerge (that word gets used a lot ☺) naturally within the team. Surely this is better – from a team perspective – than assigning magnificent titles to people before they' ve proved their worth on the team.

[Matt] [laughs] Hopefully they've proved their worth somewhere in the company to get that magnificent title… Actually, a nice aspect of XP is that the individuals get to sign up for the tasks that interest them the most in each increment, so their natural role in the team really does emerge. I would recommend doing that in any project, XP or otherwise – letting people identify the things that interest them the most. People are much more highly motivated when they are good at what they are doing, hence more productive. Of course you have to draw the line somewhere though: there are always "chores" that somebody just needs to knuckle down with and get done. And an added danger is that in a roomful of programmers, no one wants to do design, or test what they've written, or write things down. They just want to get coding: everything's a prototype, nothing is customer-facing, unless they are told otherwise. Sometimes you need someone in a position of higher authority (a team leader, say) to make sure the not-so-fun stuff gets done too. Otherwise, it's like having a house full of kids who are allowed to watch TV and eat ice-cream all day, but the washing-up never gets done…

[Mark] Sounds like my house actually… But Isn't the XP approach to development, whereby something is being delivered all the time, highly motivating to software engineers – who after all like to "make something." Isn't this one of the strongest arguments for a highly incremental evolutionary approach to creating software. Surely a well motivated team will always produce a better product than a badly motivated team, regardless of the process being used?

[Doug] Well, by that reasoning we could provide free beer and dancing girls to help us produce a better product. Hey, you know … we might be onto something here. Seriously, it's great to motivate people. Oddly enough, I find the people that I teach are often highly motivated to learn some techniques that help them to do a better job. These folks generally have realized that it's not all about code, and that there are really more than 4 important things about software…that requirements matter, that schedules matter, etc. There are lots of ways to motivate people. And I'm certainly not against getting something built. My own preference is not to build the same thing 15 times, though.

Integration

[Mark] Doesn't XPs emphasis on continual integration have many benefits – in that any 'integration errors' will come fairly quickly to the attention of the team.

[Matt] A daily build is definitely a good thing to do. Scrum recommends doing that, for example. It's true that if you leave integration for too long, the codebase starts to diverge, and it becomes increasingly difficult to fit things together. In my mind, XP's approach of integrating as often as possible takes things a bit too far though. I believe that you get diminishing returns. Even if the build & integration process is really quick – say, 5 minutes – it's still more time than no time at all. It all mounts up. If the team is communicating well anyway, and the design is well defined, code divergence becomes less of a problem – in which case, getting everyone to integrate once a day is plenty.

Testing

[Mark] XP has a strong focus on automated functional and unit testing, and related practises like test first programming. Isn't this a good thing, and in particular doesn't test first programming make developers think through class and/or component interface designs thoroughly before implementing them. In other words – doesn't test first programming ensure a focus on clean interface design and cut down the amount to which interfaces are polluted by implementation details?

[Doug] You know, I'm a big fan of unit testing, and I think the increased emphasis on rigorous and automated unit testing is absolutely the best thing to come out of the whole XP phenomenon. I don't have anything in particular against writing the test before writing the code (the way we teach use cases is to write the user manual before writing the code, so it's kind of familiar in a way) although I think that writing the test first is less necessary if you've done a good design using, let's say, sequence diagrams and class diagrams, and I definitely do not agree that writing the test first is an adequate replacement for forward-looking OO analysis and design. And this point (that the unit test is the design) is where I have a difference in philosophy with the Extremos. We call this " Design After First Testing" (roll your own acronym again).

So, unit testing: a very important and very good thing. Automated test suites: also good and important. Writing the test before the code: fine by me. Unit tests as a replacement for design: nope.

It's interesting. Way back when XP first came out, I proposed combining the front-end use case driven OOAD approach that I follow with XP's back end code and test techniques. And I still think this is where the industry will wind up eventually.

When I proposed this synthesis of ideas on OTUG, as I recall, I received a personal response from Mr. Beck whose tone I found somewhat surprising, informing me that all the XP techniques had to be used together as a set, and by-the-way, how much code had I ever written, anyway. For about 3 days afterwards, I kept getting offline EMAILs from extremo fanatics asking me how much code I had ever written. They all went away when I informed them that I had been employed as a programmer for about 15 years. But it was my first introduction to the Extremo culture of attacking the character of anyone who dared disagree with their point of view.

But I still think that the combination of a use-case driven approach from the front and the aggressive unit testing on the back would make for a successful project. And, as I said,our next book: " Agile Modeling with ICONIX Process" will be expanding on this theme.

As we've been doing this, a song that Rob and I have been working on for the last couple of days has been running thru my head (not even Matt has seen this one yet, it's new, but it hits a few of the points we've been discussing). If you have space, maybe your readers might get a kick out of this one….it's called Unit Test Writer, and it goes to the tune of " Paperback Writer" by the Beatles.

Unit Test Writer

Don't like UML
Man it's much too hard
Rather scribble some notes on an index card

Hey design is dead
And things couldn't be better
I just got a job
And I'm gonna be a unit test writer
Unit test writer
Unit test writerrrrrrrr

Well, requirements
Are a pain in the neck
Good thing that I found this book by Kent Beck
They're the customer's problem
It says so right here
So I don't care too much
Cause I'm gonna be a unit test writer
Unit test writer
Unit test writerrrrrrrr

Don't do architecture
Haven't got the urge
Rather just write code and let it emerge
At 5PM each day
You know I'm on my way
Schedule's not our job
Man it's fun to be a unit test writer
Unit test writer

Unit test writerrrrrrrr

I think that kind of sums quite a few thing up. And, if you enjoyed that, you'll definitely like the book..

Documentation

[Mark] 'The code is the documentation' Is an XP maxim. Isn't it true that most project documentation gets to be so completely out of date as to be damaging, and that therefore it's a waste of time producing it in the first place.

[Doug] Here's one of my favorite stories. A couple of years ago, Mark, you published an interview with "Uncle Bob" Martin where he stated "Extreme Programmers are not afraid of Oral Documentation". This is one of my favorite extremo quotes of all time. It disguises an attitude that I'd sum up as "we're too lazy to document our work" and makes it sound like this is somehow an act of bravery and courage. What a load of crap! So here's the story…this article came out about 3 years ago, because my son Rob was about 10 years old then, and he was sitting in my office after school when he saw me busting a gut laughing at this line. A couple of days later I'm driving him to school in the morning when he asks me "Dad, is oral documentation oxymoronic…..or is it just plain moronic?" I quoted him in one of my UMLWorld keynote speeches and the whole audience broke up laughing.

The next year at UMLWorld I was on a panel that included Uncle Bob and Martin Fowler, and the subject came up again, from the audience. I remember getting my speaker evals back from that panel and seeing one that said "Panel members, except for Doug, have lost touch with reality".

Freedom from documentation is, however, almost certainly one of the prime factors in XP's wild popularity among cowboy coders. These are the folks who say they are doing XP, but aren't pair programming, nor unit testing, etc. They're just "bravely" not documenting their work.

[Mark] But coming back to my point, isn't the issue of documentation becoming out of date very quickly one of genuine concern? What are we to do in these circumstances – spend valuable time updating the documents or get on with more development? Isn't there a ' half way house' whereby just a minimal set of documentation can be maintained, with other documents being discarded having served their purpose of either helping us think issues through, or helping us communicate across the team?

[Doug] I'm pretty much a minimalist. I don't like big 600 page documents. I'm much more concerned with the upfront thought process than I am with documenting things to death. I like whatever documentation that gets produced to be a natural by-product of doing the design.

[Matt] Requirements, in whatever form they're in – use cases, user stories, bullet points, executable acceptance tests – really do need to be kept up-to-date, because without that check-list it becomes very difficult to test whether a requirement has been implemented – especially if no one remembers what the requirement was supposed to cover. I think XPers would agree on this one. The story cards might end up slipping a bit, but the acceptance tests would always need to be kept up-to-date. Of course this raises the question of whether a bunch of executable scripts are an acceptable method of documenting your requirements – but I think we've already answered that one for one of the other questions…

It also pays to keep the design up-to-date – at least at a high level (i.e. the architecture). I've never understood why some programmers have such a problem with updating their design documentation. I think the problem is simply that they don't enjoy it; it's a chore. But it really doesn't take very long, especially if you've taken a little extra time getting the design right in the first place – in which case, the design should be quite stable anyway. An effective way of stabilising a pre-code design is (paradoxically, perhaps) to write some code first: nothing pretty, just some throwaway code so that you get a feel for what you're doing. Just a small amount of prototyping in this way (perhaps just spending a day or two on this) can have a profound effect on the design. It's the effect that XP is supposed to provide (i.e. increased understanding of the design through coding), but you really don't need to "push the dial up to 10" in order to get it. (By that I mean you don't need to be prototyping all the time).

One very useful benefit of keeping the design up-to-date is that, just like with writing it down in the first place, the act of updating it is like a design review. Just refactoring a class diagram is bound to reveal some flaws or possible improvements to the design, because all the "noise" that you get with source code is hidden away. Modeling is pure design; programming is… well, it's design plus noise. Another useful benefit of writing the design down is that it's there for everyone else in the team. No need to spend an hour, or a few days, reciting the design to the new guy. Instead, just say "here's the design… any questions, I'll be at my desk 6 feet away…" The alternative is very like "Fahrenheit 451". How long would it take to recite an entire book, anyway? And then to do it all over again the next day for someone else in the team, and then for Q.A, and again for the customer's due diligence rep, and so on.

More general

[Mark] Scalability has been raised as a concern about XP .... what are your thoughts on this?

[Doug] There are lots of issues with scalability in XP. No written requirements specs, all the programmers in a big room, no models except code, etc. etc. The only argument I've ever heard for scalability of XP is "hey, look at C3 – it worked on a big project there". Except that it didn't work. I once asked on OTUG if anybody had or knew of any large project XP success stories. Nobody responded. I asked several times. I still haven't heard of one. Have you?

Summary

[Mark] So could you summarise your views on XP for me.

[Doug] For me, the extremos lost credibility a couple of years back when they tried to take a dismal failure (C3) and pass it off as a resounding success. Remember "everyone else is selling something"? I think the community at large got sold something, but not by "everybody else". Back then we were being asked to "suspend our disbelief" that this admittedly bizarre approach wouldn't work. As we've been working on this book, we've found some stuff that just leaves us shaking our heads. How does C3 get cancelled in Jan 2000 and a reputable publication like The Economist cite it as a complete success almost a full year after its cancellation? How does that happen? At any rate, I have a great deal of difficulty giving credibility to further claims of success by these folks.

I can also remember when the "onsite customer" was supposed to be the cornerstone of the "good communication" supposedly at the core of XP. But the definition of "onsite customer" keeps changing. Last I saw (recent quote from Kent Beck) it is now supposed to be "a team as big or bigger than the programming team". So I guess if you hire a team of extremos to write your code, you have to devote a bunch of your own people as large as the programming team to work with the coders. So as near as I can follow it at the moment it's supposed to be two programmers at every keyboard and two customers for each pair of programmers? And this is supposed to be more efficient than writing specifications?

[Matt] An unfortunate consequence of criticising XP is that people tend to see you as anti-agility. That couldn't be further from the truth for us. I think the point about XP is that their goals are noble enough: give the customer what they want; give them the chance to change their mind, even late in the project; give them results early so that they can give you feedback early; and so on. But the way that XP goes about achieving these goals makes my hair stand on end. The manner in which XP is evolving and being redefined, or "fixed" (for example the changing definition of on-site customer that Doug just mentioned; and the amount of up-front design that they recommend doing) suggests that something in its basic design is just fundamentally wrong.

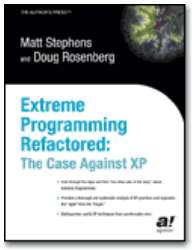
[doug] Yeah, XP is redefined at the convenience of whatever discussion the extremos happens to be in at the time.… basically they're just making it up as they go along.

[Mark] Doug and Matt, thankyou for your time.

[Doug and Matt] You're welcome.

Biographies/References

For further information see the following title:



See also:

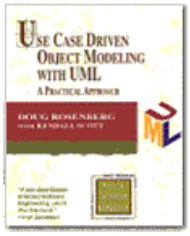
- http://c2.com/cgi/wiki?ChryslerComprehensiveCompensation
- http://c2.com/cgi/wiki?CthreeProjectTerminated
- "The Best team in the World": http://www.computer.org/SEweb/Dynabook/DaimlerChryslerSdb.htm
- Distributed Computing, October 1998. Reprinted on xprogramming.com: http://www.xprogramming.com/publications/dc9810cs.pdf

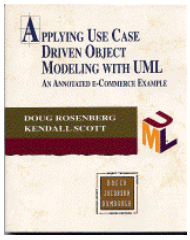
MATT STEPHENS - Matt Stephens has been a software developer for over ten years (or twenty+ years if you count his first experience with programming at the age of 11). Much of his commercial work has involved Systems Integration, enterprise architecture, and team leading. Matt is the editor of (and regular contributor to) http://www.softwarereality.com, a satirical website for software developers and managers.

DOUG ROSENBERG - Doug Rosenberg of ICONIX Software Engineering, Inc. has been providing system development tools and training for nearly two decades, with particular emphasis on object-oriented methods. He developed a Unified Booch/Rumbaugh/ Jacobson design method in 1993 that preceded Rational's UML by several years.

He has produced more than a dozen multimedia tutorials on object technology, including

COMPREHENSIVE COM and COMPLETE CORBA, and is the author of *Use Case Driven Object Modeling with UML* and *Applying Use Case Driven Object Modeling with UML*.





Visual Basic Comes of Age – VB.NET

Visual Basic used to be derided for not being a "real" programming language. Not any more says PAUL HATCHER - as he explains how VB has joined the big boys.

Introduction

It has been just over a year since Microsoft released its new .NET toolset and we are looking forward to its first patch in April ("Everett"). This article is going to describe the background to Microsoft's .NET architecture, introduce you to VB.NET, and give you some pointers on how to adapt if you are coming from a Visual Basic background to what is going to be a major change to your working environment over the next few years.

The Windows platform's lower levels are old, and have been developed using a variety of styles of programming that have evolved over the years. The early design was performed during the 1980s, before object-oriented programming became mainstream, and it shows in the API: a flat namespace with thousands of methods; use of such conventions as Hungarian notation; languages did not support abstract data types; and support for a variety of naming schemes.

During the early 1990s, object-oriented programming became more popular and C++ libraries such as MFC were layered over the top of the API; then Visual Basic came along to allow us to get on with writing business applications rather than plumbing. But still, fundamentally underlying the whole structure is that old API.

Support for component-based programming arrived in the mid-1990s, and significant additions to the flat API were provided in the form of OLE and COM. However the underlying flat C APIs were never originally intended to support dynamic object creation or to support the typical object lifecycle – and this left us with the joys of COM reference counting! Although Visual Basic relieved some of the tedium of programming with COM, you still had to be aware of the issues to produce stable, scalable applications.

What is .NET?

.NET is Microsoft's next generation software that provides an environment similar to a Java VM in which code is executed – called the Common Language Runtime (CLR). The CLR was designed with a number of goals in mind, in particular:

- Portability – it will run on PCs and on small devices such as PDAs.
- Provision of low-level support for modern object-oriented, component-based programming - general enough to support a variety of languages
- Provision of a rich, fine-grained security model where you can limit execution of code based on who wrote it, where the code came from and what code is trying to call it.

Above the CLR sits the .NET Framework, a class library comprising over 6,500 classes and interfaces just waiting to be used. It covers everything from dictionary classes such as hashtables and collections through to XML serialization of business objects and Web Services, of which more later.

The .NET Framework is subdivided by means of namespaces – which enable the libraries to be divided into areas of responsibility. The name of a class need only unique within a given namespaces, freeing the developer to use their own names without fear of clashing with base types. The following table summarises some of the main .NET namespaces

System	Fundamental classes, base class implementations, common value and reference data types, interfaces, events and exceptions
System.IO	Synchronous and asynchronous IO on files and data streams
System.Drawing	Provides access to the GDI+ library, including such items as basic drawing primitives, meta-file and printing support.
System.Windows.Forms	Base classes for creating Windows-based applications using rich (and heavy) user interface on a Windows client.
System.Web	Enables browser-server applications to be developed, System.Web.UI defines controls that can be used in ASP.NET applications to develop rich user-interfaces.
System.EnterpriseServices	The way into the COM+ service architecture for .NET components
System.Globalization	Defines culture and language information, including collation orders, calendar types, e.g. Julian, Hebrew and Japanese are all catered for.
System.Data	Defines ADO.NET, the replacement for ADO, see section below for details

VB.NET

Language inter-operability

VB.NET is the direct replacement for Visual Basic 6.0 though, as we will see shortly, due to the degree of change it is a cousin rather than a child of that language. It provides many new and improved features such as inheritance, overloading and compliance with the common language specification (CLS) – which standardizes such things as data types and how objects are exposed and inter-operate; classes, components and services defined in VB.NET may be used by any CLS-compliant language and vice-versa.

OO Credentials

VB.NET, like most other .NET languages, is an object-oriented language supporting single inheritance, multiple interfaces and structured exception handling, and supports fine grained control of inheritance characteristics such as visibility of attributes and properties. In the following code example we define some interfaces and then implement some classes based upon them:

```
'Define some interfaces
Public Interface IAnimal
    Property Name() As String
End Interface

Public Interface ILandAnimal
    Inherits IAnimal
    Sub Walk()
End Interface

Public Interface IAquaticAnimal
    Inherits IAnimal
    Sub Swim()
End Interface

Public Interface IAquaticLandAnimal
    Inherits ILandAnimal, IAquaticAnimal
End Interface

' Define some classes that implement them
Public Class AnimalBase Implements IAnimal
    Private p_name As String

    ' Call to constructor
    Public Sub New(ByVal name As String)
        Me.Name = name
    End Sub

    ' Properties require Get and Set
    ' implementations
    Public Property Name()
       As String Implements IAnimal.Name
       Get
          Name = p_name
       End Get
       Set(ByVal Value As String)
          p_name = Value
       End Set
    End Property
End Class
Public Class Elephant : Inherits AnimalBase
    Implements ILandAnimal

    Sub New(ByVal name As String)
        ' Must call base class constructor
        MyBase.New(name)
    End Sub
    Sub Walk() Implements ILandAnimal.Walk
        ' Walk implementation goes here
    End Sub
End Class

Public Class Fish : Inherits AnimalBase
    Implements IAquaticAnimal

    Sub New(ByVal name As String)
        ' Must call base class constructor
        MyBase.New(name)
    End Sub

    Sub Swim() Implements IAquaticAnimal.Swim
        ' Swim implementation goes here
    End Sub
End Class

Public Class Otter : Inherits AnimalBase
    Implements IAquaticLandAnimal

    Sub New(ByVal name As String)
        ' Must callbase class constructor
        MyBase.New(name)
    End Sub

' #Region is instruction to Visual Studio .NET
' Enables this region to be collapsed
#Region " Public methods"
    Sub Walk()
       Implements IAquaticLandAnimal.Walk
        ' TODO Walk implementation goes here
    End Sub

    Sub Swim()
       Implements IAquaticLandAnimal.Swim
       ' TODO Swim implementation goes here
    End Sub
#End Region
End Class
```

The first thing to note is that multiple interface inheritance *is* supported. This can be very useful where you need to design a class which can be used polymorphically in two interface contexts – but note that you cannot inherit the behavior.

The second thing to note is that this code shows a language pattern (or idiom) that is very common in the .NET Framework. This can be summarised as follows:

- Firstly an interface called *IAnimal*, *and* sub-interfaces (e.g. ILandAnimal) are defined
- Then a base *implementation* of the base interface (IAnimal) is provided – see IAnimalBase
- Then we define the animal classes we're really interested in – *Elephant*, *Otter*, *Fish*, etc. These use implementation inheritance to get the default *IAnimal* implementation provided by the *AnimalBase* class. Other methods (Swim, etc.) must be implemented explicitly in these classes.

Note that with the code shown, it is valid to create classes of type AnimalBase, which is probably not what we want. To rectify this problem we can put the *MustInherits* keyword before the class name.

We may also define virtual methods inside a class descendant class must implement by using the *MustOverride* keyword, or provide a default

implementation that is changeable by child classes using the *Overridable* keyword.

One welcome addition to VB.NET is the use of explicit constructors (see the *New* methods in the code above). Use of constructors *forces* client code to initialise a class correctly – thus removing a likely source of bugs.

Those of you who know VB6.0 might be asking, "So where's the equivalent of *Class_Terminate* – the destructor function", and the short answer is that it does not exist. The CLR, like the Java VM, uses a garbage collection scheme to detect when an object goes out of scope and so you have very little control over when a class is finally destroyed. If it is using essential resources such as database connections or file handles, then you have to implement the IDisposable interface and create your own disposer to release the resources.

You will also notice the new style of property declaration that includes the Get/Set code inside a Property wrapper. To produce a read-only or write-only property you simply use the keywords ReadOnly or WriteOnly before the property definition and exclude the Get/Set as appropriate. This means that the Get/Set will always have the same level of visibility - so you can no longer have a public-scope property reader and a friend-scope property setter. This may mean that your coding style will need some changes if you're familiar with VB6.0.

A major change from VB6.0 is that Property Let is gone for ever and you no longer use the keyword Set to assign to objects to variables - everything is an object in .NET anyway.

One style guideline that I would like to introduce you to is shown on the classes that inherit from AnimalBase thus:

```
Public Class Elephant : Inherits AnimalBase
```

Classes may implement many interfaces but inherit one implementation – so unless you take care it can be difficult to "see" the base implementation. Following the convention of putting the base class implementation inheritance on the same line as the class declaration helps avoid this.

One useful feature of Visual Studio .NET is how the "TODO" lines (in the code) are handled: they appear in a task list at the bottom of the screen, and can be filtered out if required. So you can leave notes to yourself and your colleagues indicating areas requiring further attention; the tool will automatically bring them to your attention.

Error Handling

With the introduction of structured exception handling, we say farewell to Vb6.0's *On Error Goto:*. We get instead a variation on the Try..Catch..Finally syntax that is supported by C++ and others, that allows us to explicitly control the behaviour of the application under failure conditions. The following code shows the framework for a structured exception handler

```
Try
   ' Starts a structured exception handler.
   ' Place executable statements that may
   ' generate an exception in this block.

Catch [optional filters]
   ' This code runs if the statements listed
   ' in the Try block fail and the filter on
   ' the Catch statement is true.

[Additional Catch blocks]

Finally
   ' This code always runs immediately before
   ' the Try statement exits.
End Try
```

The Try block contains the code that you want to be monitored during execution. If an exception is raised by any of this code, execution passes to the Catch block, which operates in a similar fashion to a Case statement - the filter can specify the type of exception that you are handling. This means that you should always write the Catch blocks in the order of most specific exception, to most general, and always have a Catch block at the end with no filter to cater for exceptions you are not expecting: "No-one expects the Spanish Exception" (Sorry Ed). The Finally block should contain tidy up code that you always want to run, even if an exception has been thrown, for example to release resources. Here is a concrete example from the VB documentation showing some file IO with structured exception handling

```
Function GetStringsFromFile(
   ByVal FileName As String
) As Collection

   Dim Strings As New Collection
   Dim Stream As System.IO.StreamReader =
      System.IO.File.OpenText(FileName)
   'Open the file.

   Try
      While True
      ' Loop terminates with
      ' EndOfStreamException error when end
      ' of stream is reached.

         Strings.Add(Stream.ReadLine())
      End While

   Catch eos As System.IO.EndOfStreamException
      ' No action is necessary ;
      ' end of stream has been reached.

   Catch IOExcep As System.IO.IOException
      ' Some kind of error occurred. Report
      ' error and clear collection.
      MsgBox(IOExcep.Message)
      Strings = Nothing

   Finally
      Stream.Close() 'Close the file.
```

```
    End Try
    Return Strings
End Function
```

If you want to catch an exception to do some error processing, but still forward the error, you should re-throw the original exception (this ensure you don't lose the stack trace information to that point):

```
Try

Catch ex As Exception
    ' Throws original exception, continues
    ' stack trace from original throw point.

    Throw

End Try
```

In this example, whoever receives the exception rethrown in the above example can get access to the full stack trace – right down to the source of the exception.

In the following example, the trace would start in the block shown – earlier stack information (who originally threw the exception) would be lost:

```
Try

Catch ex As Exception
    ' Throws *new* exception,
    ' stacks trace starts from here

    Throw ex

End Try
```

Data Access – ADO.NET

Data access and structures have changed fundamentally under the .NET Framework. Rather than ADO we have ADO.NET. ADO and ADO.NET really only have the letters A, D and O in common.

ADO is a layer on top of the OLEDB COM library to flexible access to a wide range of different data stores including non-relational stores such as Exchange and Active Directory.

Unfortunately, due to its COM heritage, ADO does not handle XML and disconnected recordsets particularly well. The ADO provider cannot read any arbitrary XML that is well-formed, but is instead restricted to the "urn:schemas-microsoft-com:rowset" XML schema. Furthermore, the data remains intimately tied to the data source that originated it: taking data from an Access database and pushing it into a DB2 database was possible - but only just and only with a lot of additional programming. Microsoft therefore had a number of goals when designing ADO.NET:

- Retain the feel of the ADO programming model, after all a lot of people used it successfully.
- Simplify the programming model compared to ADO
- Tight integration with XML
- Loosen the link to relational databases and make it simpler to deal with heterogeneous data sources.

Disconnected Data

ADO.NET is designed around disconnected data access, and this means no more server-side cursors. The two key classes that provide the data are the DataReader and the DataSet.

DataReader is very similar to the fire hose cursor in SQL Server, in that it provides a read-only, forward-only stream of data that you have to handle yourself to put it into a structure. This involves knowing the type of each underlying column so that you can perform the correct casts etc. DataSet on the other hand is much closer ADO's disconnected Recordset with a number of key twists.

First of all, a DataSet has no knowledge whatsoever of what data source produced it, i.e. it is 'pure' data that need bear no relationship to any database structure that you have. Secondly, a DataSet can hold multiple resultsets (DataTables) that can be related internally using its DataRelation objects. Taking these two together it means that it is perfectly possible to have a DataSet where one DataTable was provided by SQL Server, another is based on an Excel workbook and the third is a combination of data from a DB2 database and the Exchange mail server.

The DataSet object can load from or save both data and schema information to XML, and may interact closely with XmlDataDocument, which provides a DOM compliant view of the DataSet. Changes to either object are automatically reflected in the other.

Visual Studio .NET provides some very useful capabilities for dealing with all of this.

- Start by adding an XML document to your project and coding some data within it.
- Now right click anywhere inside the XML file and select "Create Schema". VS.NET will then build an associated schema file.
- Open up the schema and fix-up the types of the elements as everything is likely to have been defined as xs:string.
- Lastly switch to the XML source view and add an ID attribute to the xs:schema node.

A command line tool shipped with VS.NET, XSD, uses this to name the class it creates

```
<xs:schema id="mydata"…>
```

You now have a typed XML and an associated data file, which will allow you to create a typed data set using the

.NET Framework. Open a VS.NET command prompt and type

```
xsd /d /l:vb MyData.xsd
```

This will create the MyData.vb class, to see the equivalent C# class use /l:cs instead. You now have a typed DataSet that may be bound onto a Windows form, a Web form or supplied over a Web Service.

Database Connectivity

So the question remains how we get the data from the data store, be it relational or non-relational, to the DataReader or DataSet - the answer is DataAdapter classes.

These comes in a variety of flavours that target specific database platforms: so OleDbDataAdapter will talk to any OLEDB data source; SqlClientDataAdapter is targeted specifically at SQL Server 2000, etc.

Forms and Controls

The forms engine in Visual Studio has also undergone a major revision. Although under VB6 we were told that forms were essentially 'visual' classes, the practicalities of using them in this way made for some very 'interesting' problems!

In VS.NET, forms and controls share all of the features of other classes with the addition of a visual surface that is controlled by a set of components called designers.

The format of forms has also changed for the better with FRX forms banished to that great bit-bucket in the sky. Instead we have a region controlled by the Windows Form Designer that describes the controls and their properties and a .RESX file which contains XML for any additional binary resources.

Events/Delegates

Event handling has also undergone a major revision with the introduction of *delegates*. A delegate is the type-safe object-oriented version of a function pointer, and wiring an event to a delegate can be achieved in two ways:

- Using the designer which creates the control WithEvents and uses the "Handles…" syntax
- Manually using the AddHandler/RemoveHandler methods

Of the two, the Handler methods are the most flexible and the fastest, since you can add and remove handlers at runtime, even on objects that have been created at runtime. The speed issue arises the WithEvent syntax causes an additional de-reference of the underlying object to occur at runtime.

A later article will explore this more fully as the area is extremely rich and allows you to simply write callback routines as well as handle events.

Migrating from VB 6

Microsoft has announced their plans for the future of VB6.0. The good news is that it will be supported commercially until at least 2008. The bad news is that once you have started writing VB.NET you will not want to go back to the restrictions that VB6.0 imposes upon you.

So the question remains; what to do with all the investment you have made in VB6.0 code? In most organisations, starting from scratch will not be a commercially viable option, and there will always be pressures to deliver new functionality to the business rather than spend time on re-inventing features they already have that are cleaner.

The strategy that I would adopt is as follows

- Extend
- Wrapper
- Rewrite

Extend

You want to get started with .NET but you have a large amount of existing VB6.0 code that you cannot afford to redevelop. What you can do is use the ability of .NET to interoperate with COM. You write your new business logic in .NET, but call it from your original VB application.

Wrapper

Then comes the stage when you need a more substantial change, e.g. you had a VB6.0 Windows application containing complex business logic that you want to deploy as a web application.

In this case you use the COM/.NET interoperate ability the other way around, where your business logic is contained in a COM DLL, but is invoked from a ASP.NET application.

This of course implies that you have nicely factored out your business logic from your presentation tier and forms, but we all do that every time don't we ☺

Rewrite

However, there will come a time when you have to consider re-writing your application due to the level of change required, or architecture of your original solution.

First of all if you attempt to port your code rather than rewrite it, do not expect the Project Migration Wizard to solve your problems. The code it produces does take

care of vast majority of syntax issues, but significant areas remain, e.g. data access, security etc. Secondly, if you attempt a port, you will miss a large proportion of the benefits associated with VB.NET, such as use of inheritance. For example by basing all forms in an application against a base class you can introduce application-wide changes very simply. On a recent project we adopted this approach and the code volume in the forms alone dropped by over 70%.

Finally, when you do re-write take the time to define the architecture

Conclusion

So finally, the three things I want you to remember are

- The Framework is the important thing, whether you use VB.NET or C# for your day-to-day work will be either a work or personal preference. Howeverit is worth being able to read and write both.
- With VB.Net you can write well-structured, scalable, interoperable, explicitly multi-threaded applications; if your C++ colleague is being paid more than you, now is the time to ask for that raise.
- Have fun! Microsoft has produced a well-featured, and for Microsoft, stable, version 1.0 environment in which you can develop a lot of cool software in a much shorter time than previously.

Author Bio

PAUL HATCHER holds MSCD and MCAD certifications and consults with Ratio Group. Paul is a specialist in writing custom software application development for medium to large size companies based on the Microsoft .NET platform. Paul can be reached via email at paulh@ratio.co.uk.

See back page (40) for details of VB.NET training.

rOOts 2003 - May 5th – 7th, Bergen Conference Center, Norway.
www.roots.dnd.no

rOOts (recent Object Oriented trends symposium) is a forum for presentation, debate and study of the latest object oriented theories and practices. The conference is held in Norway, the country in which OO technology finds its roots. Sponsored by the Norwegian Computer Society (DND), this four-year-old annual event has quickly become a favorite among its target audience: European IT professionals and their managers.

Now firmly rooted in the Scandinavian and European IT community, rOOts 2003 hosts a panel of speakers including (but not limited to) Dave Thomas, Martin Fowler, Angelika Langer, John Sowa and Kevlin Henney. See the rOOts program for details. All the speakers have promised to bring their most recent work, which promises at least a few surprises for our delegates.

The conference concludes with a wide range of experience reports reflecting on current and ongoing work by industry professionals and scientists. 'Birds-Of-a-Feather' (BOF) sessions will give delegates unique opportunities to interact with the experts' through informal discussions and code-ins. For details and registration visit the rOOts website. www.roots.dnd.no

www.roots.dnd.no

Book Review (by Richard Vaughan)

Agile Software Development

Rating: 4 out of 5.
Author: Alistair Cockburn
ISBN: 0201699699
Recommended Target Audience: All developers, technical managers, project managers

Anyone who has experienced the pain of real software development and failed projects is (or should be) interested in what it is that goes wrong and how we can do it better. The subject of development methodologies is therefore of great importance and, in the spirit of objectivity, I should at the outset declare my allegiance to this book's subject matter - I have both feet so deeply in the Agile camp that they took root years ago, long before the formal concept came about. For me, it seems obvious that the source of the trouble is not even *under* our noses but is, in fact, just behind and a little above the nasal apparatus.

For example, Gödel's Incompleteness Theorem implies that the role of human creativity is an integral part of symbolic reasoning. If you accept this then it follows that any formalisation of software-development practices must include a strong element of human psychology. As Gerald Weinberg (whom Cockburn himself quotes) once said: ' That it is *people* who design software is terribly obvious…and ignored.'

So, to the book itself: Weighing in at a trim 278 pages, it is organised into six chapters along with an introduction, three appendices, an extensive set of bibliographic references and an index. Cockburn's style is fluid and readable, and the main text is frequently interspersed with small anecdotes, which he uses to illustrate the points he is making. Importantly, each chapter ends with a 'What should I do Tomorrow?' section that translates the theoretical discussion into concrete steps that one can execute on a real project.

Cockburn's core thesis is that one should maximise face to face communication within teams, and he makes the following point at the beginning of the second chapter: 'If we are going to build systems out of people, we should understand people's operating characteristics.' This sheer realism is what is so agreeable about the book (and the Agile standpoint as a whole). To paraphrase Cockburn, you cannot expect to enjoy an homogenous distribution of aptitude, experience, approach and downright personal-quirkiness in any development team. We cannot therefore view all developers as being largely plug compatible. He complements this by asserting that no single methodology will fit all projects and that one has to tailor the approach taken to account for variations in people, tools, geographical issues and, of course, the nature of the development challenge at hand. Moreover, he is stressing that one should, if necessary, be prepared to adjust the methodology mid stream.

The introduction therefore examines the nature of and problems with human communication. Subsequent chapters cover the communication games that people can play, an examination of the characteristics of the individual and communication within teams. The book then examines the methodologies that abound before progressing to examine the spectrum of Agile methodologies. This includes a good overview and critique of eXtreme Programming, which relates the principles contained therein to the general points and approach espoused by the rest of the book. Cockburn finally covers his own methodology (or methodology set, more accurately), namely the Crystal methodologies.

The book has a good selection of anecdotes, some amusing, that Cockburn uses to illustrate a given point. For example, with regard to the way that people apply a given methodology and then contravene it by taking short cuts, he tells of a time that a project leader showed him a collection of diagrams that they used in a project. Cockburn ascertained that they used an iterative and incremental approach and that the requirements and the design changed in the second iteration. The following exchange then took place:

Cockburn: 'How did you manage to update all these diagrams in the second iteration?'
Leader: 'Oh, we didn't. We just changed the code...'

The book also contains some excellent insights. For example, Cockburn quotes Jim Highsmith (an IBM Fellow) to point out that Process does not mean Discipline - These are distinct issues and of the two, discipline is the more powerful. This is because 'a person who chooses to act with care and consistency will do better than someone who is just following instructions'. The common mistake that people make is that 'they believe that adherence to a process will somehow engender discipline'. This simply does not follow.

One potential problem does lurk amongst all of this, which is that the very people whose stance this book attempts to change are likely to miss the point and simply repeat the mistakes of the past but in an 'Agile' manner. That is to say that they could leap at the idea of Agile methodologies using Cockburn's and/or others' contributions as the springboard, only then to miss the point by slavishly adhering to the principles and thereby return to square one. Good management is more about the intelligent application of a little psychology than it is about a mass of rules. Thankfully Cockburn reminds us that 'Agile is an attitude not a formula', although for my money, he could have given this point more prominence than it gets.

If I were to identify a real downside, however, I would have liked to have seen the many salient points of the Agile philosophy crystallised out into a rigorous, axiomatically based argument. A lot of the time I felt awash in a soup of very effective but loosely connected assertions and discussions - Being agile (in the traditional sense) and flexible does not preclude the development of clear logic.

Subject matter aside, the only other real criticism that I could level is that the index could be a little more generous. For example, towards the end, Cockburn cites a project that had a distributed team-structure and then shows how it failed the 'second test' (of methodologies, not software). The index, however, does not mention the 'tests' that he is alluding to; indeed testing (in terms of software viability) does not appear in the index at all, even though it is an integral part of every project and is referred to in a number of places in the main text.

Overall, I felt the book to be an interesting and important addition to the literature base. If you already place the individual at the centre of the development process it will tell you a lot of things that you already know. However, this is not a criticism as, for me, it laid previously unread ideas down on paper. Its central idea is one that many should sit up and take notice of and it fills that philosophy out with excellent discussion of the finer points - A welcome addition to the armoury and worth reading.

DOTNET TRAINING AND DEVELOPMENT FROM THE EXPERTS

.NET – A One Day Overview – Public or In-house

• Introduction	• if .NET is the answer, what is the question?
• .NET: The Vision and the Platform	• The .NET object model and type system
• Language interoperability	• Frameworks and applications
• Strategies for adopting .NET	• Summary and perspective

Managed C++ Under .NET – A One Day Overview

• .NET overview for C++ developers	• Managed and unmanaged code overview
• Storage Management	• Inheritance
• Operator overloading	• Delegates and events
• Properties	• The .NET reflection mechanism
• Porting to .NET	• Summary and conclusions

C#.NET in 5 Days – Extensive Hands-on – Public or In-house

• What is .NET?	• The .NET Framework
• Interoperability	• Streams and Files
• Internet Access	• Attributes and Reflection
• Development Technologies	• Collection Classes
• Web Services	• Deployment

VB.NET & ASP.NET in 5 Days – Extensive Hands-on – Public or In-house

• Introduction to .NET	• Visual Studio.NET
• Windows Forms	• DataTables and DataGrids
• Menus and Dialogs	• Overview of the Debugger
• Introduction to OOP	• Classes and Methods
• Overloading Methods	• Understanding Inheritance
• Using the List Box Control	• ADO.NET
• Using XML from VB.NET	• Interacting with COM components
• ASP.NET & WebForms	• Using WebForms
• Using DataGrids	• Using DataList Control
• Web Services	· Asynchronous Web Services

For More Information Contact Ratio on 020 8579 7900
Email info@ratio.co.uk or see www.ratio.co.uk

Talk to us about SME DOTNET development services.