contents

credits & contacts

Overload Editor:
John Merrells
overload@accu.org

Contributing Editors:
Alan Griffiths
alan@octopull.demon.co.uk

Mark Radford
mark@twonine.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial - An Industry that Refuses to Learn

I reckon it's now getting on for two and a half years since I sat in *that* meeting with the rest of the development group. I'd made the mistake of giving up my independence and joining a company owned by someone else – a small and apparently promising company about to get bigger, and with a bunch of development staff that were well above the industry norm in terms of their enthusiasm for their jobs. Then one day and out of the blue, management hired a technical architect, and soon after his arrival, a development group meeting was called to brief the group on his vision for future technical direction. *Components* featured highly on his agenda because, he explained, if we developed components, we would get the benefit of reusability.

Now, most of the development group members had been the industry for two to three years, except one who had been in the industry for about a decade and a half – he and I spontaneously started to reminisce about how this was exactly the rhetoric surrounding objects a decade earlier! We pointed out that this promised reusability hadn't been delivered then, and we were not confident it would be this time around. We also pointed out some other common sense things, such as you need to get past the margin of *usability* first because all too often software is difficult enough just to use, and if it's difficult to use then forget about reusing it. However we couldn't get the architect to understand any of this, and he went on to lead the company in reaping the cost of his dubious wisdom.

So, why am I telling you this story? Well, the above tale is an illustration of the search for the *silver bullet* – that one *thing* that is the solution to all problems. It seems like a poignant way of starting off this editorial, in which I want to talk about what I see as the greatest troubles of our industry. These are the troubles that are uncomfortable – even painful – to talk about. These are the troubles that I encounter far too often, that condemn our industry as nothing short of dysfunctional, and that every time I encounter them make me wish I had a different career (ok, the grass is always greener on the other side, I know). What I have to say may come across as something of a rant, and if it does, *so be it*!

At the time of writing this I'm in my seventeenth year in the industry, and I've seen my share of failed projects. In fact, I've seen more fail than succeed. I say this with some confidence that readers who have a comparable number of years in the industry will, with some sadness, share the sentiment (recently a friend told me of a former colleague who after fifteen years in the industry has yet to work on a successful project). We work in an industry that worships technology. We've all been privy to debates about the merits of one programming language versus another and one technology versus another, and why one is better for some projects and not for others, but let me say this: I have never seen a project fail as a direct result of a particular technology! In fact, I think it's fair to say that most of the projects I've seen fail were doomed to fail, before any technical work was done or before any technology had been put in place. I've just mentioned the term silver bullet, a term coined by Frederick Brooks in his "No Silver Bullet" essays, which can be found included as chapters in the second edition of his book "The Mythical Man-Month". There's much of relevance to discuss in these essays, but I'm going to let that pass because I want to turn my attention to another chapter of the same book: the one entitled "Why Did the Tower of Babel Fail?" In this chapter the author argues compellingly that those building the tower had a clear purpose (even if naïve), and they had adequate technology, plenty of time and all other resources. Yet the project failed. Again Brooks' arguments are compelling: it was lack of *communication* (and consequently a lack of organisation) that was to blame. It has been my experience too, that much of the time projects fail because of lack of communication. I say much of the time, because some of the time there are other reasons too, but I don't want to get into detail because it'll distract from the point I want to make.

Frederick Brooks was project manager for IBM's OS/360, and in "The Mythical Man-Month" he documents the lessons he learned on this and other projects. Now be afraid, be very afraid, because the first edition was first published back in 1975, which means anyone in the industry under twenty-eight years of age (and that's a significant proportion) wasn't even born when the first edition of "The Mythical Man-Month" was first published – and we're still getting it wrong *today*!

Just writing about this makes me shudder, but let's press on anyway, I have another story to tell...

A few weeks ago, I was having a conversation with a web programmer. Now web programming is something I still haven't done, but there was a possibility of me getting into it and helping out on a project with this programmer's company, so he was giving me some background on what's involved before I started pitching into the detail. I'm précising the conversation rather a lot here, but basically, the programmer explained that the most significant issues are to do with the *transactional* nature of programming for the web – the browser collects information, sends it to the web server, and

when the web server replies there is no knowledge in the browser of what went before. As he was telling me this I began to recall a couple of projects I worked on in the early 1990s that involved writing Windows front-ends for mainframe systems. This involved tapping into the 3270 terminal protocol commonly in mainframe systems, where the terminal is not interactive with the mainframe – rather it collects input from a form presented on the screen, sends the whole lot back to the mainframe in a buffer, and springs back into action when the mainframe sends a new buffer full of information back to be presented. I think I started to detect a pattern emerging.

As I mulled over the above conversation I began to think about the two friends I made at the time of my first job as a contractor in 1997, working in a small software house based in Buckinghamshire – a company with a history of supplying mainframe systems. While I was there I became friends with two of the permanent staff who I still regularly visit. Both have been in the industry for nearly two decades longer than I have. Now the scary bit: these two old timers are part of a generation of programmers who solved the problems associated with the transactional programming model over a quarter of a century ago! They solved them when programming mainframes in assembler, the new generation of web programmer solves them using a variety of modern technologies, but the problems and their solutions haven't changed much in a *quarter of a century*.

Let's face it, how could these two old timers pass their knowledge on to the modern generation of programmers anyway? What mechanisms are available for facilitating this kind of communication? Well, in the early 1990s hope appeared in the form of the patterns movement, which offered a framework for capturing problems, their solutions, together with all relevant information such as the tradeoffs involved. Around that time the "Design Patterns" book (by Gamma et al) appeared, and made Patterns visible to the development community at large. However, Patterns were hijacked and became another bandwagon, another buzzword! This is the most damning indictment against the industry: Patterns were hijacked not because of any fault of Erich Gamma or any of his co-authors, but because it is *in the nature* of an industry that is more impressed by fads than by progress.

I fear that my two Bucks based friends are now part of a whole generation of programmer that the technology and silver bullet crazed industry has seen fit to forget about. In recent times the Agile Development movement, and the *Extreme Programming* approach in particular, has brought practices such as unit testing – practices that my two friends regarded, in their youth, as a *way of life* – to the attention of the modern programmer. I fear that such practices will now join the armoury of buzzwords that must be present on a CV to make sure it is found by agency database searches. I fear the same will happen to these practices as happened with Patterns: the industry will pay lip service with enthusiasm, but because of its deep rooted dysfunction it will lose out to the desire to adopt – or rather the desire to be seen to adopt – the latest fad.

The software development industry has *not* made any significant progress towards improving its productivity over the last quarter of a century. It continues to ignore communication and instead looks to fads – in the form of fashionable practices or technologies – to provide a silver bullet that will produce huge leaps in productivity. I'm not saying practices and technologies are not of value, on the contrary their continued development is an essential part of progress, assuming they are used for their merit and not just for their novelty value. Unfortunately, developing something that is an essential *part* of progress is not enough to make progress happen, and *real* progress – the improvement of productivity – is *not* happening. Problems are encountered in the course of any project, and it is the resource – mainly in the form of people's time – expended in solving problems that significantly inflates project costs. This cost can be cut dramatically if known solutions can be applied at the strategic point in the project. This is where the price of looking for the *silver bullet* really takes its toll – it distracts from the process of reusing and cultivating existing knowledge because its promise seduces the industry into concentrating on the search for what can't be found.

Making progress will be a struggle (if it is even possible) because there are too few people who understand that there is a problem. Further, of those who understand that there is a problem, few have any idea how big the problem really is. I believe it is unlikely that things will improve over the next decade, and possibly longer. The software development industry is an industry that refuses to learn.

Mark Radford
mark@twonine.co.uk

Copy Deadlines

All articles intended for publication in *Overload 61* should be submitted to the editor by May 1st 2004, and for *Overload 62* by July 1st 2004.

Letters to the Editor(s)

Comments on Overload 58 article 'A Standard Individual: A Licenced Engineer'

I read with much interest the article in Overload 58 – 'A Standard Individual: A Licenced Engineer'. I thought it was very good and I agreed with most if not all of it.

One small point however – the BCS CPD (Continuing Professional Development) scheme has not disappeared, in fact it is going strong and highly recommended (if not compulsory) for their professional members to partake of. CPD itself is a requirement for continued membership, it is the recording of it on their scheme that is not compulsory. I record my courses, study with the Open University, seminars and so on with their CPD scheme and have found it to have a good balance between being too complex and rigourous (and thus a chore to comply with) and too simple (which is not of much realistic use).

I am also a member of another professional institution - the Insitute of Physics, which used to have a CPD scheme a few years ago but it was withdrawn and has not (yet) reappeared. As I am registered CEng via the IOP I feel I ought to keep up to date with developments in my field(s) even though it is not required, and therefore I use the BCS scheme for all my development.

Yours,

Ray Poynter (aka Dr Ray Poynter CEng CPhys MBCS MInstP)
rayp@tau-re.org

Testing Templated Code

I've just read Mark Radford's editorial in the recent edition of Overload, and felt some unease at the rather sweeping statement that having templated code means you only have to test it once. True, there is only one set of code rather than a lot of similar copies, any one of which could later be independently tweaked.

But, if a template algorithm works fine with one type, there's no guarantee it'll work fine with another type. Assuming it compiles, then the provided type conforms to the required interface, but the implementation of that interface is in the hands of its provider, and clearly affects the result of running the algorithm on that type. Now, a counter argument might be that there are well-defined requirements for some sorts of system, e.g. value types should be default constructable, etc., which would aim to help out, but there's no accounting for problems in the implementation of even relatively simple requirements.

I also thought of another category of potential problem. If the template algorithm uses named functions of a type, then it's going to be pretty clear what functions get called (leaving aside polymorphic behaviour), but if the algorithm is using operators, then it might not be running the same code for different types. For example, suppose there is a global `operator+` in place, which most types are using and the algorithm is adding instances together. That's all fine, but what happens when the supplied type implements its own `operator+`? I don't actually know off the top of my head what'll get precedence (if there isn't an ambiguity), but you can see the problem.

Having used pre-template compilers in the past (a situation like the hypothetical management banning them), we occasionally resorted to macros (with all their inherent problems) to synthesise template behaviour on relatively simple functions which we wanted implemented for lots of types.

Regards,

Simon Sebright
SimonSebright@hotmail.com

Mark's Reply

Thanks to Simon for taking the time to write in. I'm flattered that my first attempt at writing the editorial prompted someone to write in! Now of course, it falls on me to respond.

Let me paraphrase Simon's first point: the fact that a template works with one type is no guarantee it'll work with another, and further, a specialisation may compile ok but the implementation could do anything. Now I agree, a misbehaving implementation would certainly throw a spanner in the works, but let us note in passing that the issue is not specific to templates. For example, a (non-template) function could take advantage of run-time polymorphic behaviour by taking, as a parameter, a pointer/reference to an abstract base class. The same issue applies here with regard to the overriding of a virtual member function.

In the case of implementation supplied by either a template argument or the overriding of a virtual function, there is a requirement for a kind of specification that cannot be enforced at compile-time. The compiler can check syntax, and it can to some extent check semantics, but it can only check the semantics that can be represented in the code. What is at issue here is the specification of run-time behaviour – and when run-time behaviour happens, the compiler is no longer watching.

We clearly need to look elsewhere for help, and it presents itself in the form of *design by contract* – a term coined because of an analogy with legal contracts: a software routine can have *contractual obligations* placed on its input that must be met in order for it to behave correctly. Now, there are static aspects to such contractual obligations in terms of the syntax and semantics of a types interface, but that's not the end of the story, for contractual obligations extend well into run-time behaviour. By way of an example, let's take a look at the sorting algorithms in the C++ standard library, and more specifically, at the comparison operation (which may be in the form of `operator<` or a predicate, depending on the algorithm overload selected) used to determine the sorting order.

The C++ standard states (in section 25.3) that in order for a sorting algorithm to work properly the comparison function must induce a strict weak ordering on the values compared. Note that code using the sorting algorithms will compile without error even if the strict weak ordering requirement is not met. The comparison function could return a hard coded value of true every time and compilation would be unaffected – but this would render all bets off at run-time. With this contract specification in place, sort algorithms can be tested using objects of a specimen type designed to test the algorithm – the order into which these objects should be sorted is known in advance (as part of their design), and they can

Achieving FitNesse in C++

by Alan Griffiths

Sometimes a very simple idea can make a very big difference. FitNesse (and "Fit" on which it builds) are very simple ideas – and when I first encountered them my reaction was "so what". It was only after talking to people using them that I found time to investigate them more seriously. What then, do they offer?

What they offer is feedback on a system under development meeting functional requirements. They encode tests in a format that can both be executed and explained (or even edited) by a customer and can also be executed directly against the system to demonstrate the results. As a perennial problem with software development is a failure of communication, anything that helps narrow the gap between requirements and deliverables is worthy of investigation.

Fit and FitCpp

Fit is a Java component that takes some HTML input, interprets it into tests, exercises the tests against the system being tested, and outputs an updated version of the HTML that incorporates the results. The developer needs to write some lightweight classes that define the mapping between fields in the requirements document and the parameters of the methods to be invoked in the system. These classes are called "Fixtures" and, in the Java implementation must conform to a "fixture" convention that allows Fit to use the reflections API to set values and call methods. (Conventions like this are common in the Java world: Junit, EJBs, JavaBeans and NakedObjects are all technologies that adopt this approach.)

FitCpp is a translation of the Fit functionality into C++. Clearly, C++ lacks a direct analogue of "reflections", but by using a combination of macros and templates it allows the developer to achieve the same effect.

What both of these tools do is to scan the input HTML for tables; these are used to specify the fixture to use, together with the input data and results expected. The input is shown in Table 1.

The reporting is very simple: the results of tests that succeed turn green (light grey in illustration) and the results (both expected and actual are shown) of tests that fail turn red (dark grey). The output is shown in Table 2.

The system must be able to calculate the number of vowels and/or consonants in a sentence.		
sentence_analyser		
Sentence	vowels()	consonants()
Hello world!	3	7
This sentence has four vowels and thirty three consonants.	4	33
"0123456789" Does it handle numbers 0123456789?	7	12
*(@$£)! &^%$£?@# <>/	0	0

Table 1: Input HTML

sentence_analyser		
Sentence	vowels()	consonants()
Hello world!	3	7
This sentence has four vowels and thirty three consonants.	4 *expected* 16 *actual*	33
"0123456789" Does it handle numbers 0123456789?	7	12
*(@$£)! &^%$£?@# <>/	0	0

Table 2: Output Report

[continued on next page]

be sorted from various random orders and the post sort sequence checked to see if it is what is expected. It is important here to note exactly what is being tested: the *template*! It is the logic encoded in the template itself that is tested, to demonstrate that it honours its (behavioural) contractual obligation, *provided* the supplied argument object honours the contractual obligations of the parameter type.

Simon also makes a point about operators: specifically, what happens if the template uses an operator and there are other, unexpected overloads of the same operator in scope? This issue isn't peculiar to operators – especially when argument dependent lookup (ADL) gets in on the act. Anyway, with regard to templates (actually it's not just templates, but here it is templates with which we are concerned), ADL and overloading combine to make up the compile-time polymorphic behaviour of the template parameter's types interface. Now I have already argued that for a template to work properly its argument objects must honour the contractual obligations placed on their respective parameter types – and that's exactly what I'm going to do again here! When I first brought design by contract into this response I mentioned that although contractual obligations extend well into run-time behaviour, they start with compile-time semantics. Compile-time polymorphism is part of the set of semantics that constitute the (compile-time) contractual obligations on template parameter types, and places responsibility for the correct overload selection firmly with the parameter type designer.

As Simon indicates, the correct behaviour of parameterised code is dependent on its argument types also behaving correctly. However, fundamental design principles such as *separation of concerns*, *encapsulation* and *design by contract* all form symmetry – that is, the presence of all of them forms something very robust but things become very fragile in the absence of any. The testing of components is a practice that relies on this symmetry to be intact, but as long as the symmetry is intact, components can be tested individually – the symmetry is comprised of the *rules* governing the interplay between components, and not the interplay between actual components themselves. Therefore, any particular component can be tested to verify it is fulfilling its part of the bargain, and if it is it can do no more and the rest is up to the other participants.

Mark Radford
mark@twonine.co.uk

[continued from previous page]
I have to admit that my initial reaction to this was "so what?" – but what I had missed is that tables of test data and results are very easy to incorporate into requirements documents and are understood by users. Of course, editing the raw HTML that lies behind the above display would put off the typical user: that is where FitNesse comes in...

FitNesse

FitNesse is a Java based Wiki server that includes the facility to passing the HTML on a page to Fit (or to FitCpp) and display the results. It also allows the test pages to be organised into test suites and provides summary reporting on the test results for each page in the suite. (As with the individual tests these are colour coded.)

The idea is that the functional requirements can be captured as a combination of text and tables on the Wiki site. Obviously, when these are first added all the tests will fail, but as the system is developed more and more tests will pass – and as the tests change colour they automatically provide visible feedback on progress for both developers and customers.

All of this is easier to explain and confirm with the customer than a test written in Java, C++, or a test scripting language. (And the text that underlies the Wiki is easier to follow than HTML.)

Doing it yourself

If you've read this far then you'll want to know what you need to do to set this all up. There are basically four things to do:

1. get FitNesse and install it;
2. get fitcpp and install it;
3. capture the requirements as a set of Wiki pages; and,
4. write corresponding fixtures for your system.

Getting FitNesse is pretty trivial (just go to the website given in the references below and follow the links). There are even helpful .bat and .sh scripts for starting it – basically it runs straight out of the archive.

Getting fitcpp isn't quite as simple – the published code I downloaded had a few bugs and is reliant on a number of Microsoft C++ "features" that allow non-standard code to compile. I've updated it with some fixes and to get it to compile with gcc – I've made this updated version available on my website (and also passed the changes back to the author).

Capturing the requirements as a Wiki page is pretty easy. The above example looks like:

```
!define COMMAND_PATTERN {
../c++/fitcpp/overload/FitOverload.exe }

The system must be able to calculate the
number of vowels and/or consonants in a
sentence.

|!-sentence_analyser-!|
|sentence|vowels()|consonants()|
|Hello world!|3|7|
|This sentence has four vowels and thirty
    three consonants.|4|33|
|"0123456789" Does it handle numbers
    0123456789?|7|12|
|*(@$£)! &^%$£?@# <>/|0|0|
```

The first line overrides the default Java "fit" handler for the page and directs FitNesse to invoke an alternative hander – we'll describe creating and using this file below. The paragraph that follows is transcribed without change, and the table is marked out using the vertical bars.

Writing a C++ fixture

The ".exe" file mentioned in the preceding section needs to come from somewhere: obviously it contains any fixtures used for the tests and some code from the FitCpp library. (Clearly, in any real world usage it will also link against the application code.)

Writing a fixture takes a little more explaining, because one needs to understand the role of a fixture and the conventions involved. While there are many types of fixture possible (they simply provide an interface for processing tables) the most useful in writing tests is the "column" fixture used in the illustration above (I didn't choose the name). This fixture is defined as follows:

```
#include "ColumnFixture.h"
#include "MainFixtureMaker.h"

class sentence_analyser
        : public ColumnFixture {
public:
  sentence_analyser();
  string sentence;
  int vowels();
  int consonants();
};
```

The name of the fixture corresponds to the first line of the table – you would typically have multiple fixtures and multiple requirements tables for an application. Looking at the body of the fixture you will see that it exposes public member data[1] and parameter-less member functions that return values. These correspond to the columns of the table. Actually, in C++ (unlike Java) you don't have to make the names correspond to the columns but it is easier to go with this convention.

In Java the reflections API is used to detect these corresponding data and methods, but in C++ we have to write a few more lines:

```
sentence_analyser::sentence_analyser() {
  PUBLISH(sentence_analyser,
          string, sentence);
  PUBLISH(sentence_analyser,
          int, vowels);
  PUBLISH(sentence_analyser,
          int, consonants);
}
```

These macros register the corresponding members with the fitcpp processing engine.

In a real test harness the `vowel()` and `consonant()` member functions would invoke functionality in the application

1 Note: the use of public data in this context isn't an argument for adopting this approach in general. Fixtures have a very specialised design context and are only used within that context.

being tested. However, for the purposes of compiling and running the above demonstration the `vowel()` method is implemented as follows:

```
int sentence_analyser::vowels() {
  return std::count_if(
    sentence.begin(),
    sentence.end(),
    is_vowel());
}
```

Where `is_vowel` is a functor with the obvious functionality.

Joining the dots

In its original distribution FitCpp required some additional "boiler plate" code that provides a "Maker" class to build the Fixtures that are identified by the tables and a driver "main" method that pushes the HTML through the FitCpp Parse engine. I have factored all this out into a MainFixtureMaker class and a generic main method and placed these into the FitCpp core library. The result of this is that only a single line is needed to set up the creation of a Fixture:

```
REGISTER_FIXTURE_MAKER(sentence_analyser)
```

This is based upon the assumption of statically linking the Fixtures with the library and that statically linked objects of global scope are initialised before entering `main()`. (This behaviour isn't strictly required by the ISO standard but, as far as I can tell, it is common across all C++ implementations.)

Running the example

If we compile and link this code we will have an executable that can be invoked from the FitNesse test page described above. What happens when we invoke "test" on this page? And how do we combine tests together?

Pressing "test" causes FitNesse to invoke the program specified in the `COMMAND_PATTERN` directive. (This should be the program containing the fixtures). The program then reads the HTML supplied by Fitnesse from standard input and writes to standard output.

The FitCpp framework scans the input looking for HTML tables. When it encounters one it examines the first cell and constructs a corresponding fixture (the contents of the first cell specifies the type of the fixture). The next row of the table is used to match up the column names with the members of the corresponding Fixture "published" during construction. For column fixtures (like the example we've been following) the remaining rows are processed one cell at a time setting data or calling functions as appropriate. (The example shows how this allows the sentence parameter required by `count_vowels()` to be supplied prior to invoking it). The results of any function calls are compared with the values embedded in the HTML to select the appropriate output (green if it matches, red if not).

There can be (and usually are) several tables within a functional requirements page (which is a single invocation of the program) the plugins corresponding to these tables will be invoked sequentially as part of the same process. Neither FitNesse nor FitCpp provides a mechanism for passing information between these tests but, as program state is maintained, the various fixtures can communicate using global data/objects.

Scaling up

There are several issues of scale that need to be addressed: the principal ones are managing a collection of functional requirements, and allowing developers to apply the tests to their local version of the system (instead of to the integration build on a server somewhere). FitNesse addresses both of these concerns quite neatly by having pages with special properties.

The web pages with FitNesse are organised into a hierarchy like a file system (with the same convention seen in Java package names – using a "." character as a path separator). For example, a page might be called "SystemRequirements" and beneath this would come other pages such as "SystemRequirements.ChapterOne" etc. By setting the "suite" or "virtual" property on a page then the processing of the sub-pages can be tailored.

If the top-level page is given the "suite" property then it acquires a "suite" option that can be used to run all the tests in pages that lie beneath it. This results in a summary of the results of these tests with links to the results of individual tests so that the user can "drill down" if desired. Setting up a "suite" page that covers all the functional requirements allows progress to be demonstrated in an immediate and visible manner (as more functionality is delivered more of the tests show "green").

The "virtual" attribute is used in a local instance of the FitNesse Wiki server running on a developer's machine. This causes FitNesse to fetch the content of all the subpages from another URL (the remote Wiki server – in the configuration I have set up this is the integration "build" machine). The local Wiki server will run tests (and suites) locally against the developer's version of the system but the content of the functional tests is maintained on the central server. To avoid confusion the pages retrieved from the remote server are given a blue background before being shown by the local server.

Miscellaneous Notes

As with any test framework it is normally appropriate to consider beginning the test with a "StartUp" fixture and ending with a "TearDown" fixture. (There is more on patterns of test design in the FitNesse documentation.)

Row Fixtures

In the discussion above I've only discussed Column Fixtures, there is a second type of Fixture supported by FitNesse/FitCpp – RowFixtures. These are used for describing collections of rows that may be returned by the application (think "result set") and allow missing and/or extra rows to be reported. The way they are implemented is directly analogous to the ColumnFixtures discussed above, and their use is adequately covered in the FitNesse documentation.

Alan Griffiths
`alan@octopull.demon.co.uk`

References

FitNesse: `http://sourceforge.net/projects/fitnesse/`
FitCpp: `http://fitnesse.org/`
(Updated version referred to in this article:
`http://www.octopull.demon.co.uk/download/fitcpp.tar.gz`)

Transforming XML with XSLT

by Fazl Rahman

David Nash's article in C Vu of October 2003 covered reading XML data into a program. Here, I hope to introduce newcomers to manipulating XML with XSLT scripts, using an example drawn from that article.

What Is It?

Simply stated, XML is portable, self-describing data structured as a tree of named nodes, each possibly containing named attributes, text, and sub-nodes. It may already be the most popular way to bridge systems built using disparate technologies. XSLT stands for eXtensible Stylesheet Language for Transformations. The *stylesheet* part concerns the presentation of XML ('pure content'), but that's about all I have to say about that. The fact that *transformations* are part of XSLT shows the W3C's recognition of the need for manipulation, as well as presentation, of XML content.

If you ever accidentally opened an XML file in a Microsoft environment, you might have been surprised to see a collapsible tree view like that in Figure 1. That is produced by the web browser using a built in XSL transform which converts the data to HTML, and is perhaps the most widespread example of using XSLT to adapt input for a pre-existing parser. (It's also handy way for looking at non-trivial XML files.)

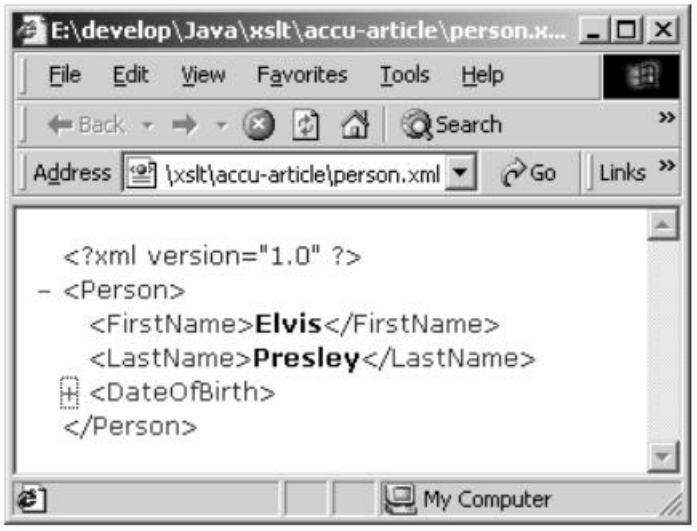

Figure 1 – XSLT used to render XML as HTML

An XSL transform is itself valid XML, usually stored in a file with the extension "`.xsl`" and commonly called an XSL *script*. Programs that execute XSL scripts against XML data are called XSLT processors (see 'Tools' below). Modern integrated development environments (IDEs) often contain a built in XSLT processor to support rapid prototyping of transforms. For example in IBM's Websphere IDE you can view two XML files side by side and point & click to specify how the source data should map to the destination – and the IDE will create a transform for you.

What Can I Do With It?

Although XML is rapidly becoming the preferred mechanism for data sharing, the XML that one system produces might not be structured precisely the way all of its consumers expect, despite containing the needed information[1]. For example, David's article shows a C++ program that can read personal data like `person.xml`, shown below.

```
<?xml version="1.0"?>
<Person>
<FirstName>Elvis</FirstName>
<LastName>Presley</LastName>
<DateOfBirth>
<Year>1935</Year>
<Month>01</Month>
<Day>08</Day>
</DateOfBirth>
</Person>
```

Assume today you build and debug a parser to read `person.xml`. What if tomorrow you're confronted with the need to read personal information from a new source which structures its data slightly differently? For example, consider flatperson.xml, which contains the same information as `person.xml` but more concisely[2]:

```
<?xml version="1.0"?>
<flatperson firstname="Elvis"
      surname="Presley" dob="19350108" />
```

Often, as I hope to show here, an XSLT script can save you having to touch your parser code by adapting the new data:

```
$ xsltproc  expand.xsl flatperson.xml
<?xml version="1.0"?>
<Person>
  <FirstName>Elvis</FirstName>
  <LastName>Presley</LastName>
  <DateOfBirth>
    <Year>1935</Year>
    <Month>01</Month>
    <Day>08</Day>
  </DateOfBirth>
</Person>
```

This article guides the novice in stages to construct scripts like `expand.xsl` (and its converse, `flatten.xsl`) which can interconvert between a `flatperson` and a `Person`.

Naturally, if your system produces output for others, you can be sure that the XML structure it uses internally will not perfectly match the format expected by each potential consumer. Here, a set of XSLT scripts can adapt that internal representation for each consumer. The classic example of this is a web portal that transforms XML into browser- or device- specific markup as a final stage of request processing. (Despite the ECMAScript standard, browsers still have their quirks in the way they handle web content.)

Tools

XSLT processors are available in many flavours just like XML parsers. They typically provide an API allowing you to incorporate a transformation capability into a program, as well as a command line wrapper for experimenting with. By default, a processor will interpret scripts (for rapid prototyping), but can

1 See `http://www.oasis-open.org` for an effort underway to change this situation.

2 Ivan Kiselev's soother for people (like me) who find XML configuration files overly verbose for name-value pairs: "..any design decision is a compromise, some like it hot and nobody's perfect".

also be told to precompile them for performance. I often use the Cygwin environment which includes Gnome's `libxslt` and its `xsltproc` command (which we have seen in action above). My JDK installation includes the Apache `Xalan`[3] processor, so I could achieve the same effect thus:

```
$ java org.apache.xalan.xslt.Process
      -XSL expand.xsl -IN flatperson.xml
```

See Resources at end for pointers where you can download these. I use `xsltproc` in this article for brevity.

How Does a Transform Work?

An XSL script has a top-level `xsl:transform` element typically containing a number of 'functions' that you write, having the job of operating on some part of the input XML to produce some part of the output data. These 'functions' are embodied in `xsl:template` elements and can be written declaratively and/or imperatively, depending on how you want them to be invoked. By default a template will just copy any text it contains to the output when called; however templates also have at their disposal rich data manipulation and control flow constructs similar to those lurking in your favourite programming languages.

The XSLT processor treats the input XML data as a tree of nodes, each of which has an associated path (a route, by name, down the tree to that node). The processor starts at the top of this tree (path: "/") then searches the transform script for a template matching the current path (via the template's `match="..."` attribute).

If the processor finds a matching template, it invokes it. What happens next depends on how the template is coded. E.g. it could imperatively call other templates as part of its processing, just like a C function can call other functions.

If no user-supplied template matches the root node, a "default template" is invoked, which prints out the text of the current node then recursively traverses its descendants. This is the effect of transforming `person.xml` using an 'empty' XSL transform (one containing no user templates):

```
$ xsltproc empty.xsl person.xml
  Elvis
  Presley
    1935
    1
    8
```

Hello...

Time for a concrete example[4]. This is the `hello.xsl` script, containing one simple template.

```
<?xml version="1.0"?>
<xsl:transform
       xmlns:xsl="http://www.w3.org/1999/
       XSL/Transform" version="1.0">
<xsl:output method="text"/>
<xsl:template match="/"> Hello World
</xsl:template>
</xsl:transform>
```

3 In JDK 1.4 the Xalan classes live in `<JAVA_HOME>\jre\lib\rt.jar`. In case of classpath woes, try adding this jar to your classpath. If your JDK precedes 1.4 you can download the Xalan classes (see Resources).

4 I recommend installing one of the free XSLT processors and trying out the examples.

Don't worry too much about the packaging – focus on the single `xsl:template` element. You'll often see scripts with a template declared to `match "/"` whose job is simply to invoke other templates in the desired order, a role akin to a `main()` function. Our template is less ambitious, doing its work with no help.

Two things to note are: (1) The `match` attribute declares its intention to be called in the context of the root input node and (2) It just ignores the input data and outputs the greeting we've all grown to love, with a nod to Kernighan & Ritchie.

This shows `hello.xsl` being applied to `person.xml` on the command line.

```
$ xsltproc hello.xsl person.xml
Hello World
```

As you can see, the contents of `person.xml` don't figure in the output – all we see is the greeting.

So, we've seen one script blindly copy all text from its input to the output and another ignoring its input altogether. Time for something with a bit more intent behind it.

Personal Hello

When manipulating XML you generally want your templates to read parts of the input structure and create output with a structure more suitable for your purposes. Let me introduce the `xsl:value-of` tag, which is a bit like the SQL `select` statement, as it returns the value of some specified aspect of the data. You specify what you want to retrieve, in its `select` attribute, with a path (i.e. a particular route to a particular node in the input tree).

For example, in our `hello` template we could query the `Person`'s first name with the path `/Person/FirstName`. To upgrade `hello.xsl` so it greets the right person on first name terms, replace the text `World` with the tag `<xsl:value-of select="/Person/FirstName"/>`.

This shows the resulting script `personal.xsl` and its effect on our `Person` data.

```
$ cat personal.xsl
<?xml version="1.0"?>
<xsl:transform
       xmlns:xsl="http://www.w3.org/1999/
       XSL/Transform" version="1.0" >
  <xsl:output method="text"/>
  <xsl:template match="/" >
    Hello <xsl:value-of select=
                  "/Person/FirstName"/>
  </xsl:template>
</xsl:transform>

$ xsltproc  personal.xsl person.xml
    Hello Elvis
```

Expand an Attribute

We're nearing the original goal of converting `flatperson` data to `Person` data (for which, hypothetically, we have a pre-existing parser). New aspects to this are: A `flatperson` holds its information in attributes (like `firstname`) which the 'expanding' transform must read; and the output must contain XML elements (like `FirstName`) encapsulating this information, rather than plain text like "`Hello ...`".

We can reference an attribute (in `xsl:value-of` and elsewhere) using `@<AttributeName>` in a path. For example a `flatperson`'s first name has the path "`/flatperson/@firstname`". (This has a modicum of charm, I confess.)

Creating XML structure in the output is achieved by simply embedding XML tags in the template body.

This shows the script `expandfirst.xsl` and how it partly reconstitutes a `flatperson` to a `Person` (for brevity, it only reconstitutes the first name).

```
$ cat expandfirst.xsl
<?xml version="1.0"?>
<xsl:transform
       xmlns:xsl="http://www.w3.org/1999/
       XSL/Transform" version="1.0" >
  <xsl:output indent="yes" />
  <xsl:template match="/" >
    <Person>
      <FirstName>
        <xsl:value-of select=
              "/flatperson/@firstname"/>
      </FirstName>
    </Person>
  </xsl:template>
</xsl:transform>

$ xsltproc expandfirst.xsl flatperson.xml
<?xml version="1.0"?>
<Person>
  <FirstName>Elvis</FirstName>
</Person>
```

(Note the `xsl:output` tag tells the processor what kind of output is expected. Without the `method="text"` attribute, the processor defaults to emitting an `<?xml..?>` header . Without the indent attribute, redundant whitespace like newlines would not be added, leading to slightly leaner output at the expense of readability.)

Flatten an Element

For completeness let's look at the reverse direction. The `xsl:attribute` tag can inject an attribute into an output XML element. To illustrate, here is `flattenfirst.xsl`, the 'inverse' of `expandfirst.xsl`. It takes a `Person` on input and outputs a minimal `flatperson` containing just the first name:

```
$ cat flattenfirst.xsl
<?xml version="1.0"?>
<xsl:transform
       xmlns:xsl="http://www.w3.org/1999/
       XSL/Transform" version="1.0" >
  <xsl:template match="/" >
    <flatperson>
      <xsl:attribute name="firstname">
        <xsl:value-of select=
                  "/Person/FirstName"/>
      </xsl:attribute>
    </flatperson>
  </xsl:template>
</xsl:transform>

$ xsltproc flattenfirst.xsl person.xml
<?xml version="1.0"?>
<flatperson firstname="Elvis"/>
```

(Note that all `xsl:attribute` tags must precede other content for an element. You might see why if you consider where attributes would end up in the output in relation to, say sub-elements.)

Completing the Scripts

Within the context of a `flatperson`, the path "`/flatperson/@dob`" refers to the `dob` attribute whose value (for Elvis) is "`19350108`". The function `substring()` can pick out individual parts so we can populate the `Year`, `Month` and `Day` elements of a reconstituted `Person`. For example this tag extracts the YYYY digits of a `flatperson`'s `dob` attribute:

```
<xsl:value-of select=
      "substring(/flatperson/@dob,1,4)"/>
```

Armed with this, the hands-on reader is encouraged to upgrade `expandfirst.xsl` into the script `expand.xsl` which reconstitutes the *whole* `Person` from a `flatperson`.

XSLT additionally provides access to a rich set of string-related functions, including regular expressions. For details see the Xquery and Xpath specifications (Resources).

Now, by default, the XSL processor merges a sequence of text items into a single text item when creating an output node, so the following would be one way to concatenate a `Person`'s `DateOfBirth` sub-elements, for readers wishing to complete `flattenfirst.xsl`:

```
<xsl:attribute name="dob">
<xsl:value-of select=
            "/Person/DateOfBirth/Year"/>
<xsl:value-of select=
           "/Person/DateOfBirth/Month"/>
<xsl:value-of select=
            "/Person/DateOfBirth/Day"/>
</xsl:attribute>
```

Conclusion

Of course you can parse XML and navigate/manipulate the resulting DOM tree using various languages. However XSLT was specifically designed to transform XML so it supports working at a higher level than SAX or DOM. Though transforming then parsing can be slower than one-step parsing with a new parser, building and debugging that new parser will often be overkill as a first port of call when a simple XSLT script lets you reuse an existing parser. Once you're happy with a script, you would typically dispense with the command line interpreter in favour of programmatically invoking a precompiled version of your script from your application.

I hope I have helped curious readers in their first few steps with XSLT, with simple but self-contained examples, and shown how relatively painlessly it can adapt XML data for a pre-existing parser.

Fazl Rahman
Fazl.Rahman@web.de

[concluding sections at foot of next page]

A Little Detail

by Alexander Nasonov

Some time ago I wrote a simple mixin class template. A week later I found a little problem with it. Although I found a solution in a second I decided to analyse it more deeply. It's worth analysing further because it concerns some fundamental features of C++.

Here is the problematic code:

```
template<class T>
struct Mixin : T {
  ~Mixin();
};
```

I guess I know your feelings. The class template looks like an example taken from a C++ book. You might have been taught with code like this. Your feelings about it most likely are based on unchallenged assumptions about simple C++ language constructs. Despite its basic nature the code has one little problem.

Why does this code look nice at first glance? Well, if it was an ordinary class you could just compile it and see that everything is fine. But the "just compile it" idea doesn't work in the case of class templates. Actually, writing the code is only half the job. The second half is instantiating the template. This will be done by the user unless you think of all possible cases and instantiate them in your tests.

This is a different way of thinking. If you deal with templates you should imagine how different instantiations could be compiled. You can tell me "Hey, what's the problem, I can write tests and instantiate the template there". Yes, you can. But first you have to find the right classes for instantiations. As an example, can you find an instantiation of `Mixin<X>` that breaks the code above?

Don't think too much, I have an answer. Here it is:

```
struct X {
    virtual ~X() throw();
};
```

Once the right class is found you can try to compile it. My compiler (g++ 3.2.2) complains:

```
1.cpp: In instantiation of 'Mixin<X>':
1.cpp:12:   instantiated from here
1.cpp:3: looser throw specifier for 'void
         Mixin<T>::Mixin() [with T = X]'
1.cpp:7:   overriding 'virtual X::~X()
           throw ()'
```

According to our best friend, the C++ standard [1], paragraph 15.4, bullet 3:

> *If a virtual function has an exception-specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the exception-specification of the base class virtual function.*

None is allowed in a destructor of base class `X`. Therefore, none should be allowed in a destructor of the derived class `Mixin<X>`:

```
template<class T>
struct Mixin : T {
  ~Mixin() throw();
};
```

Well, we found a quick solution to the problem. Does it have some drawbacks? Can it break other instantiations? For example, what if `T`'s destructor may occasionally throw? `Mixin<X>` has an empty exception specification list, therefore, `std::unexpected` will be called. This function will call `std::terminate` and program execution will be aborted. This is definitely not what a user wants.

Luckily, many C++ gurus recommend not throwing exceptions in the destructor at all. It's enough to mention in the documentation of `Mixin` that the destructor of `T` must meet the `Nothrow` requirement.

It seems that the problem is solved. Indeed, if you're a bug hunter who has just ended up with code like that above you can stop reading here. I'd rather analyze it a little bit more.

What is annoying me in a destructor with an empty exception specification is the fact that a compiler may put the destructor's code into a try-catch block. It protects your application against "exception leaks". The try-catch block can be omitted only if the destructor's body is available and the compiler can deduce that the destructor never throws. Otherwise, unnecessary try-catch blocks make the code bigger and execution slower.

Another inconvenience of the code was suggested by Phil Bass while reviewing this article. His concern is a design flaw rather than implementation details. Phil suggested that, if

Acknowledgements

LOTS of people kindly read drafts! My thanks in particular to Frederek Althoff, Phil Bass, Dr Islam Choudhury, Dr Trevor Hopkins and Dirk Laessig for helpful feedback.

References

David Nash, "Combining the STL with SAX and Xpath for Effective XML Parsing", *C Vu Volume 15 No 5* (October 2003, pp.18-20)

Ivan Kiselev, *Aspect Oriented Programming with AspectJ*, SAMS Publishing.

Kernighan & Richie, *The C Programming Language*, Prentice-Hall (The archetypal use of "Hello World" to introduce a programming language.)

Resources

XSLT specification on the W3C site:
`http://tinyurl.com/2ewsm`

Xpath/Xquery functions/operators: `http://tinyurl.com/2puva`

Gnome project's `libxslt`: `http://xmlsoft.org/XSLT`
Windows binary distribution: `http://tinyurl.com/2p9yt`

Xalan-c at Apache website: `http://xml.apache.org/xalan-c`

Sun have a Web Services tutorial with a good intro to XML and XSLT: `http://tinyurl.com/2572z`

The newsgroup `comp.text.xml` is full of helpful stuff.

Check `http://cocoon.apache.org` for an approach that heavily uses XSLT for multi channel user interfaces.

Mixin is part of a general-purpose library, it would be great if Mixin were to follow a project-specific exception specification policy.

There are two major exception specification policies used in destructors:

- No exception specification at all
- Empty exception specification

Probably, the first policy is used more widely than the second. I would say both are used in C++ projects. For example, the C++ standard library uses both.

Needless to say, a Mixin<T> destructor that is neutral to the exception specification policy of T is preferred rather than a destructor that forces using either choice.

I recommend that you stop reading for a moment and try to find a best-of-all-worlds solution. A solution that is free from the limitation of the first version of Mixin and that doesn't dictate a particular exception specification policy.

Although you have little freedom in defining the destructor the solution may surprise you. It is no destructor at all, that is, an implicitly defined destructor:

```
template<class T>
struct Mixin : T {
};
```

Why is this better? To explain why, let me refer you to [1], paragraph 15.4, bullet 13. Apart from an explanation of our case it contains an example with multiple inheritance, which we'll analyze later. In my informal interpretation, an implicitly defined destructor "inherits" its exception specification from the base destructor. Whatever exception specification T's destructor has so has an implicitly defined destructor of Mixin<T>. Perfect, exactly what we need!

You may ask how to keep it implicitly defined in real class templates. I recommend that you use RAII wrappers, smart pointers, C++ strings and containers wherever you can. This reduces the need for explicitly defined destructors to very unusual cases.

More complexity

Now it's time to solve the problem I faced. It's almost the same as our original problem with one difference – Mixin has an additional base:

```
struct Base {
  // ...
};

template<class T>
struct Mixin : Base, T {
  // ...
};
```

It's clear that we can always use a nothrow destructor in Mixin. I'd like you to analyze the case of an implicitly defined destructor. Just remember that, on the one hand, an implicitly defined destructor inherits exception specifications from all its bases, and on the other hand, if any of the base destructors is virtual, ~Mixin() can't have a less restrictive exception specification. The analysis is a kind of combinatorial puzzle. You can combine the virtuality and exception specifications of all the destructors. Fortunately, there are only a few combinations.

The first case is a non-virtual destructor ~Base(). The analysis shows that the destructor of Base has to have an empty exception specification in order to define ~Mixin() implicitly.

```
struct Base {
  ~Base() throw();
};

template<class T>
struct Mixin : Base, T {
};
```

Although this solution dictates the exception specification policy of the Base destructor, it's still of interest because the resulting Mixin class template is neutral to the user's exception specification policies.

The second case doesn't have a solution. If Base's destructor is virtual we can always find a type T that breaks the compilation regardless of the exception specification of ~Base().

This was my case. I could take the destructor's virtuality out of the base into another class responsible for polymorphic cloning and destruction (let's say, storage management). Although it would better fit the one class, one responsibility principle I decided to use a quick fix solution:

```
struct Base {
  virtual ~Base();
};

template<class T>
struct Mixin : Base, T {
  ~Mixin() throw();
};
```

Conclusion

I'd like to draw two conclusions. First, a summary of what has been done.

Mixin classes often come with general-purpose libraries or libraries that make no assumptions about the projects that will use them. It's important to follow the project's rules and policies even when a set of projects is unknown to the library author. In this article I showed how to solve one particular problem with respect to possible uses of your code.

The second conclusion is rather philosophical. Although you can rarely find code simpler than that discussed in this article it's worth analyzing it. I dare say there is no such thing as a little detail in C++. Everything is important in the C++ world. If you find an interesting note on a C++ feature or some side effect, try to play with it. Many C++ tricks and modern techniques were discovered this way. Keep trying! Together we'll make a better language.

Alexander Nasonov
alnsn@yandex.ru

Reference

[1] ISO/IEC 14882

From Mechanism to Method: Generic Decoupling

by Kevlin Henney

Simplification of code is often equated with the elimination of options. At best, this turns out to be a false correlation; at worst, it hampers the long-term code quality and development. The side effects of premature generalization and over-abstraction [1] are as much a problem in software as the consequences of premature optimization: complexity, unmaintainability, brittleness, bloatware, strengthened coupling, weakened cohesion, loss of flexibility – in short, a lot of criticisms that we would prefer not to have leveled at our own code.

It is true that in many cases of simplification options will be eliminated, but more often than not the eliminated options are the ones that tended to complicate the code or were of little practical use in the first place – dead code waiting for a garbage collector.

For example, in the C++ Standard library, the only noticeable role that traits and allocator parameters of the `std::basic_string` serve is to complicate the usage and error reporting on, typically, `std::string`. Their role is so constrained as to make them almost completely useless. The few people that take advantage of them are often attempting to solve the wrong problem or are employing the wrong solution. There is in fact a great deal of scope for increased simplicity and useful parameterization in string types [2, 3]; it's just a shame that `std::basic_string` and its moribund parameters are already parked in that space.

It is possible to simplify the structure of software without losing effective options. It is even possible to do so and increase your options. Now, that sounds worthwhile: simpler *and* more flexible.

Decoupling in General

Although we cannot predict the future with any certainty, it is still possible to write code that is graceful and accommodating – rather than troublesome and resistant – in the face of change. Software development is concerned with the development of structure – partitioning and connection, separation and composition – so any conscious and conscientious approach to software development should have, as one of its prominent manifesto promises, a clear focus on structure management.

The aggressive pursuit of LCHC (low coupling and high cohesion) can ensure that the effect of change is simplified and isolated, rather than traumatic and global. LCHC also simplifies testing, building, versioning, experimentation, optimization, team organization, and pretty much any other development activity you can think of that absorbs more time, effort, or grief than you had originally anticipated. Sadly, few approaches can genuinely boast LCHC as one of their main pledges, preferring instead the active pursuit of more obviously crowd-pleasing headlines such as *reuse*.

The trick to achieving generality is, somewhat counter-intuitively, to make the code specific enough to be fit for purpose. A fit to the task in hand must be targeted with one eye; the other should be seeking opportunities to keep options open, but without attempting to pursue all of the choices. It is tempting to try to enumerate all the possible ways in which something could change and be adapted and then incorporate all the necessary hooks and extra parameterization into your design. Unfortunately this style tends to make your code more complex to understand. In fact, your code can become so full of conveniences that it's almost impossible to use either simply or correctly. Over-guessing may narrow rather than widen your options. You – and your users – may end up with a lot of unused code and many workarounds.

By contrast, a concerted focus on dependency management will deliver you some tangible benefits in the short term – development times, build times, lunch times – and reduce the cost of change in the long term. The loose coupling keeps the code supple and more stable as, over time, the genuine sources of variation, and therefore parameterization, become apparent and needed.

So what are the sources of coupling in C++ code? We can classify two basic forms of coupling:

- *Physical coupling* requires that for the compilation of one piece of code the compiler must see another piece of code. In practical terms this means that the code depended upon appears in the same source file or is pulled in by `#include`.
- *Conceptual coupling* [4] implies that for a piece of code to work there is a dependency on another concept, which may exist either tacitly outside the code or explicitly within it. For instance, a template parameter can be described by a set of requirements outside the code, whereas a class definition is known to the compiler.

One does not necessarily imply the other:

- An inheritance relationship represents both a conceptual and a physical relationship. A derived class is conceptually dependent on its base class(es) because it may use or override features. The compiler must also be able to see the definition of any base classes to compile a derived class.
- A class or function template conceptually depends on its template parameters, but the use of the parameter does not require any `#include` support. A dependency on an actual parameter type occurs at the point of use of instantiation, but not at the point of definition.
- The use of inline functions or template code written in headers may introduce a physical dependency, but not necessarily a conceptual dependency. Use of an inline function or a class template also pulls in any of the dependencies that are used in implementation, but are not relevant to the usage interface.

There are four complementary approaches for decoupling a C++ system.

Dynamic Typing

Why does that compile-link cycle take so long? Static type checking is the root cause of the delay; the design of the preprocessor merely exacerbates the issue. Want to know that your code makes at least basic sense? Let the compiler check your types and how you're using them and then let the linker tie all the loose ends in your program together. Efficiency and confidence in execution is your reward; extended surfing breaks and water cooler conversations are your punishment. Hmm, OK, perhaps we need a different spin on this: long build times put the *irritation* and *detrimental* into iterative and incremental development, frustration and time wasting that are only temporarily relieved by a machine upgrade.

If you weaken the type system you reduce the physical dependencies. This may conjure up images of `void *` in your mind, but banish those thoughts immediately: I want to loosen coupling, but the kind of unsafe promiscuity that `void *` often encourages is not quite what I had in mind. A more dynamically checked type system lies at the heart of many interpreted languages, from LISP to Smalltalk, Awk to Ruby. Good support for reflection

allows you to get at the soft underbelly of other statically typed systems, such as Java or the meta-information available in many component middleware architectures. It is a matter of balance: you loosen the checking at compile time to increase flexibility, but you increase scope for failure at run time. You pays your money; you makes your choices; you takes your chances. That's the essence of design.

C++ does not currently have good standard support for reflection: the existing RTTI mechanism is a foot in the door, but no more. In spite of the half-open door, C++ programs often make effective use of dynamic typing:

- Variant types, such as `boost::any` [5, 6] or CORBA's `any` type, can hold values of arbitrary type. Depending on your application you can choose to leave the type fully uninterpreted, as in the case of `any`, or you can impose constraints on the contained types that are reflected in the interface of the variant (e.g., comparison or arithmetic operations).
- Work in terms of strings, interpreting them as necessary and with respect to the context. In the Age of the Internet, strings are the new integers: everyone's using them for everything. Whether we are talking about internal command languages or data exchange, strings are remarkably versatile – given the right functions and classes, they support the Three Rs. You can take some of the guesswork out of how to structure your data and work with it by adopting a data definition language or meta-language, of which XML is certainly the most fashionable.

However, remember that these techniques reduce only the physical dependencies not the conceptual ones. Those are as strong as ever and will be lining up to bite you at run time should you disrespect them. You still need to know how to use them. Their correct usage is now implicit rather than explicit, and semantic drift between versions or developers is all too easy.

Flexibility has a price... and a number. This was recently brought home to me when I was entering a particularly long order number into a spreadsheet cell: the spreadsheet abbreviated the many significant digits of the reference number using scientific notation. Aha, yes, it *is* a number, just not *that* kind of number.

Interface Classes

Inheritance in its most common employment seems to be used more for subclassing (with a focus on inheritance of code) than for subtyping (with a focus on classification and substitutability). Hierarchies that accumulate implementation, often with concrete classes inheriting from concrete classes, lead to classes that are hard to understand.

But *common* is not the same as *recommended*: such usage is in direct contrast to much of the advice on practice that is available and held in some regard. For instance, only the leaves of a hierarchy should be concrete; its roots should be fully abstract. virtual functions should be introduced into a hierarchy as pure virtuals rather than with default implementations that must be guessed, and delegation and non-public derivation should be used to acquire implementation when there is no intent to hold a reference or pointer to a base class.

Is this just so much theory? No, it's better than either just theory or just practice: it's both. In practice it can be demonstrated that the failure to use inheritance in a controlled manner can be much worse than not using it at all [7]. The use of deep hierarchies, with implementation scattered, defaulted, accumulated, and overridden over a derived trail of concrete classes, actively ambushes our ability as humans to grasp all the features of a concept within a single embrace. This kind of inheritance often sabotages the localization benefits of encapsulation.

All this may sound harsh and idealistic, but it is typically less harsh and far less idealistic than believing in the timely development, and appropriate quality, of a project that takes the common but unrecommended path. Of course there is wriggle room for pragmatism, for compromise. But remember that *to compromise* has two different meanings – make sure you choose the one that means *to settle or resolve by making concessions* rather than the one that means *to expose to suspicion, disrepute, or mischief*.

Inheritance is the strongest form of logical coupling you can have. The need for physical coupling follows in its wake: base classes must be directly visible or included in the source above their derived classes. But derivation is a blade with two edges: you can also use it to reduce coupling in a system.

An *interface class* [8, 9] (also known as a *protocol class* [10]) refocuses a class hierarchy's clients on the conceptual interface, away from the physical baggage and variability of its descendents. The absence of code in an interface class contributes to its stability [11] and comprehensibility – although a code-free class sometimes clashes with a programmer's instincts for producing executable code. The Observer pattern [12] is an example of a larger pattern that includes this smaller interface-decoupling pattern:

```
class subject;

class observer {
public:
  virtual ~observer();
  virtual void update(subject *) = 0;
protected:
  observer() {}
private:
  observer(const observer &);
  observer &operator=(const observer &);
};
```

The use of virtual functions in interface classes is distinctly public. Such a recommendation is clearly in tension with the alternative recommendation that class hierarchies should always have non-virtual public interfaces [13]. A number of practices, such as the Template Method pattern [12] and the corresponding Form Template Method refactoring [14], tend to give rise to non-virtual public interfaces in C++. Such interfaces have some useful properties, but they typically arise as a consequence of specific practices rather than being a necessary and general virtue in their own right. They are by no means the only tool in the box. Design should be considered a dialogue with a situation rather than a monologue; there is often more than one reasonable route that such a conversation may take.

Hidden Delegation

Wherever there is a recommendation concerning inheritance, you can be sure that not far behind it is a contrasting recommendation framed in terms of delegation. The root of delegation-based decoupling is the forward declaration. It can be used both to resolve the problem of tail-chasing cyclic dependencies and to reduce the exposed physical dependency of using an `#include`, reducing the essential surface area between class definitions:

```
class observer;

class subject {
public:
  virtual ~subject();
  virtual void attach(observer *) = 0;
  virtual void detach(observer *) = 0;
  ...
protected:
  subject() {}
private:
  subject(const subject &);
  subject &operator=(const subject &);
};
```

For classes that are, by nature, concrete and not part of a class hierarchy, interface decoupling through interface classes has relatively little to offer. Value objects [5], for instance, are best manipulated directly in terms of their concrete type. Interface classes are primarily a means for decoupling class hierarchies. Another practice is required for specifically concrete classes.

The common idiom goes by various names, of which the most evocative is also the name originally coined for it in the late 1980s: the Cheshire Cat idiom [15]. The name, taken from Lewis Carroll's surreal cat whose ability to disappear except for its grin quite bemused Alice, is apt:

```
class cat {
public:
  ...
private:
  class body;
  body *self;
};
```

Here the representation disappears entirely from the class definition in the header, leaving behind only the discreet smile of a pointer. The details of the body are elaborated in the corresponding source file:

```
class cat::body {
public:
  body();
  ~body();
  ... // representation details
};
```

This technique also goes by the name of the Pimpl idiom [16] or, very descriptively, as the Fully Insulating Concrete Class [10]. Naturally, all idioms have consequences that must be considered: the additional level of indirection, extra memory management, and restriction on inlined functions are the price of the afforded creature comforts in this case. The introduction of this separation also allows representation sharing, although this is not a path one should tread either necessarily or lightly [3].

Cheshire Cats can be introduced to complement the use of interface classes, ensuring that class hierarchy users are as insulated from representation details as possible. However, they are less effective with class templates. Compiler portability constraints mean that it is common to require the definition of class template members in header files. In such situations, having to include the full definition of the nested body in the header rather takes the smile off the technique.

Template Parameters

Templates are not normally associated with loosening physical coupling. Quite the opposite. The inclusion of source code in headers imposes a significant burden on the size of headers and the patience of the programmer. However, the conceptual loosening that arises from defining function and class templates independently of their actual template parameter types has a knock-on physical decoupling effect. The point at which the physical dependency on the actual parameter type is needed is deferred to the point of use in the code.

Generic decoupling forms the basis of generic programming and the STL: templated iterator ranges for algorithm-based functions and container constructors, and templated value types to allow any appropriate convertible value to be used in a function, member or non-member. The following function (inlined for brevity) shows how the implementation of an Observer's `subject` class might use existing STL features to automate observer updates:

```
class subjected : public subject {
  ...
  void notify() {
    std::for_each(
        observers.begin(),
        observers.end(),
        std::bind2nd(
            std::mem_fun(
                &observer::update),
                this));
  }
  ...
  std::list<observer *> observers;
};
```

An alternative approach perhaps demonstrates a number of generic-decoupling techniques a little more explicitly:

```
template<typename argument_type>
class update {
public:
  explicit update(argument_type argument)
     : argument(argument) {}
  template<typename updateable>
  void operator()(updateable *target)
              const {
    target->update(argument);
  }
private:
  argument_type argument;
};

template<typename argument_type>
update<argument_type>
updater(argument_type argument) {
  return update<argument_type>(argument);
}
```

This generalized code leads to the following crisp usage:

```
class subjected : public subject {
  ...
  void notify() {
    std::for_each(
       observers.begin(),
       observers.end(),
       updater(this));
  }
  ...
  std::list<observer *> observers;
};
```

The obvious trade off with using templates to decouple is that implementation detail typically migrates to header files. This is particularly noticeable when introducing member function templates in place of ordinary member functions. Another consequence of the decision to template member functions is that they cannot be declared virtual. A more dynamically typed, variant-based approach can counterbalance this [5, 17].

What is also apparent with generic decoupling is that the code tends to become more flexible and more precise as an immediate consequence. For instance, a different take on the needs of an observer dispenses with the need for any forward declarations:

```
template<typename subject>
class observer {
public:
  virtual ~observer();
  virtual void update(subject *) = 0;
protected:
  observer() {}
private:
  observer(const observer &);
  observer &operator=(const observer &);
};
```

And consequently allows more flexible and varied observing:

```
class data;
class events;

class watcher : public observer<data>,
public observer<events> {
public:
  virtual void update(data *);
  virtual void update(events *);
  ...
};
```

Noosely Coupled Exceptions

As another worked example of generic decoupling, it is possible to loosen the noose of cyclic dependencies. Consider the standard exception classes defined in `<stdexcept>`. Each exception takes a `std::string` for construction. Note that `std::string` is mentioned only in the single argument constructor: There is no requirement that it is used for implementation, and the only query function offered by the standard exceptions, `<stdexcept>`, returns a `const char *`. Given this asymmetry in construction versus query types, and the role of exceptions in a program, it is certainly open to question whether `std::string` should be used at all in the interface.

However, the issue is not so much with the choice of type dependencies in the library in general, but with the nature of the dependencies: The `<string>` header defines `std::basic_string`, some of whose functions throw `std::out_of_range`. There is therefore a cyclic dependency between the types defined in `<stdexcept>` and those in `<string>`; this logical dependency is made more physical when inlined implementations are used – the norm for template implementations. The absence of a standard `<stringfwd>` header or a more general concept of strings means that each vendor is invited to break the cycle in their own way, some of which meet users expectations and some of which do not (e.g., `char *` may or may not convert implicitly for the exception constructor argument).

As an aside, it can be considered surprising that exceptions are granted the privilege to use string given that I/O and file handling, which are more obviously and intimately connected with string handling, have no such honor. Although file streams depend on `char_traits`, as found in `<string>`, `const char *` is used as the type for naming files and the type for predefined string insertion and extraction. The `<string>` header itself depends on I/O streams, representing another dependency noose.

Loosening the Noose

Returning to the `<stdexcept>` and `<string>` cycle, a decoupling can be arrived at by considering sufficiency and substitutability: the exception classes in `<stdexcept>` are conceptually more primitive than `std::string` and should not have the imposition and dependency on such a specific string type. The dependency should be narrower and more accommodating. The diversity of string-user needs means that such users cannot be characterized collectively as a community. Likewise, their needs cannot be met by a single type such as `std::basic_string` – a class template that attempts to be all things to all people, but manages only a few in each case.

So what if we don't depend on a specific string type at all? The following is an alternative version of `std::logic_error`, which uses a dynamically allocated `char *` internal representation and has no dependency on `<string>`:

```
class logic_error : public exception {
public:
  explicit logic_error(const char *detail)
     : detail(duplicate(detail,
                        strchr(detail,
                               '\0'))) {}
  template<typename string>
  explicit logic_error(const
                       string &detail)
     : detail(duplicate(detail.begin(),
                        detail.end())) {}
  logic_error(const logic_error &other)
     : detail(duplicate(
           other.detail,
           strchr(other.detail,
                  '\0'))) {}
```

```
  logic_error &operator=(const
                    logic_error &rhs) {
    char *new_detail =
         duplicate(rhs.detail,
                   strchr(rhs.detail,
                          '\0'));
    delete[] detail;
    detail = new_detail;
    return *this;
  }
  virtual ~logic_error() {
    delete[] detail;
  }
  virtual const char *what()
                        const throw() {
    return detail;
  }
private:
  template<typename iterator>
  static char *duplicate(iterator begin,
                         iterator end) {
    char *result =
               new char[end - begin + 1];
    copy(begin, end, result);
    result[end - begin] = '\0';
    return result;
  }
  char *detail;
};
```

Lightly Strung

The most commonly used string initializer for exceptions is a vanilla null-terminated character sequence. In the revised `logic_error` shown, this maps directly to a constructor without requiring conversions and the creation of temporary string objects:

```
throw std::logic_error("illogical");
```

The templated constructor caters to the standard string type, and indeed any other character container that satisfies the minimal requirements for begin and end members that return random-access iterators – SGI's `rope` [18], `std::vector<char>`, or a suitable string type of your own devising. So with a few obvious drawbacks, not only has the cyclic dependency been removed, the generality of the code has been increased:

```
std::vector<char> message;
...
throw std::logic_error(message);
```

I said *few* drawbacks. That is not to say that there are none. However, the most obvious and significant limitation may not be considered that great a disadvantage: a string type that has a user-defined conversion to `char *`, but does not sport `begin` and `end` functions, can no longer be used to directly initialize a `logic_error`. The success of such a conversion is not guaranteed in the existing Standard, but the arrangement of types in the headers often supports it. The suggested redesign is forward rather than backward looking: string classes that support such user-defined conversions are unsafe and the absence of support for container operations is nonstandard. So if you were to rework your own existing classes to support this style of string decoupling, existing code that worked in terms of legacy string classes would need to be modified – either with explicit casts or, taking the hint, with more standard-conforming types.

Conclusion

Code should be supple, not subtle. For code there is such a thing as being too well connected and too eager to please. Generality and reuse are often better served by paying attention to necessity and to the core activities of software development – comprehension, change, and confirmation – than to whimsy and speculation.

Refactoring code to reduce its coupling often has the effect of increasing its cohesion. In the exception example, physical and conceptual decoupling improved the precision of the requirement on the string type: only specific features were required, not the whole interface. This LCHC strategy suggests a design path that is as applicable to domain-specific libraries as it is to the liberalization of string types.

Kevlin Henney
kevlin@curbralan.com

This article was originally published on the C/C++ Users Journal C++ Experts Forum in November 2001 at
http://www.cuj.com/experts/documents/s=7989/cujcexp1911Henney/
Thanks to Kevlin for allowing us to reprint it.

References and Notes

[1] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community* (Oxford, 1996).

[2] Andrei Alexandrescu. "Generic<Programming>: A Policy-Based basic_string Implementation," *C/C++ Users Journal C++ Experts Forum*, June 2001, www.cuj.com/experts/1906/alexandr.htm

[3] Kevlin Henney. "From Mechanism to Method: Distinctly Qualified," *C/C++ Users Journal C++ Experts Forum*, May 2001, www.cuj.com/experts/1905/henney.htm

[4] Conceptual dependencies are sometimes referred to as logical dependencies. The distinction between – and separation of – *logical* from *physical* has been handed down to us from structured analysis and design. However, the bias inherent in the use of the word *logical* tends to cast all physical concerns into the shade as impure and irrational. Such Puritanism is of little practical use. The natural complement of *physical* is *conceptual* rather than *logical*, whose antonym is *illogical.* C++'s reliance on the preprocessor may not be elegant, but, given its rules, it is entirely logical that a piece of code requiring a declaration in a header file should also have a physical dependency on it.

[5] Kevlin Henney. "From Mechanism to Method: Valued Conversions," *C++ Report*, July-August 2000, www.curbralan.com

[6] Boost C++ Libraries, www.boost.org

[7] Les Hatton. "Does OO Sync with the Way We Think?", *IEEE Software*, 1998, www.oakcomp.co.uk

[references concluded at foot of next page]

All Heap No Leaks

by Paul Grenyer

The use of `new` and `delete` in C++ can cause problems for both novice and expert developers alike. Using `new` without a corresponding `delete` results in a memory leak. Some languages such as C#, Java and Python provide managed objects that can be created on the managed heap leaving deletion to the garbage collector.

Smart pointers allow similar behaviour to be achieved in C++, with the added bonus that they provide deterministic destruction of the object to which they point. However, the user still has to know that they should be using smart pointers and many don't. In this article I am going to look at a way of writing C++ objects that can only be created on the heap, and that must be managed by (memory management) smart pointers.

Creating Objects on the Heap

There are a number of situations in which it makes sense to force objects to be created on the heap. I recently developed a thin database access layer for use with my cross-platform test framework [1]. The sole purpose of the layer is to allow databases to be created and dropped before and after tests. Although Open Database Connectivity (ODBC) is available on both Windows and Linux (my initial target platforms), I decided to allow the use of ActiveX Data Objects (ADO) on Windows as it sometimes provides better performance.

ODBC and ADO are very different. ODBC is implemented as a C based API and ADO is implemented as a family of COM objects. However, both use a connection string to connect to a database and allow the execution of strings containing SQL statements. Therefore both types of database connection can be used via a common abstract base class.

Like many other types of resources a database connection is expensive and the most expensive operation is generally the initial connection to the database. In an ideal situation an application would have a single database connection object that is shared by all of the objects in the application that require database access. Connection to the database would be put off until absolutely necessary (lazy evaluation), and maintained for as long as possible. This results in the connection being created in one part of the application and (potentially) destroyed in another.

A common interface to one or more concrete objects, and the lifetime requirements of a database connection object, make it an ideal candidate for heap creation. One of the most likely scenarios is that the appropriate database connection object would be created at the highest possible level via a Factory Method [2] that takes the connection string as a parameter. The database connection object is then passed to objects in the application that need to use it [3], via a reference counted smart pointer [4]). The first object to attempt to execute a SQL statement causes a connection to the database to be made, and then the connection is maintained throughout the lifetime of the database connection object.

As the database connection object is created via a factory method, it follows that it should be destroyed via a Disposal Method [5]. The disposal method can be passed to the reference counted smart pointer and used to destroy the database connection object instead of `delete`. This is an example of the Resource Acquisition Is Initialisation [6] idiom provided by smart pointers.

The use of factory and disposal methods gives the writer of the database connection object complete control over how and where the object is created.

Background

In Item 27 of More Effective C++ [7] Scott Meyers discusses ways of restricting objects so that they can only be created on the heap. Meyers explains that sometimes a developer wants to create objects that can destroy themselves by calling `delete this`. If an attempt was made to create such an object on the stack the effects could be catastrophic, so it makes sense to enforce heap creation.

Equally there are times when a developer wants to prevent an object from being created on the heap. For example local (or automatic) variables are unlikely to require heap creation. Meyers also discusses preventing objects from being created on the heap. This can be achieved by declaring `operator new` and `operator new[]` private. Meyers also suggests that unless there is a compelling reason not to, `operator delete` and `operator delete[]` should also be made private.

```
namespace Meyers {
  class NotOnHeap {
  private:
    static void* operator new(size_t);
    static void  operator delete(void*);
    static void* operator new[](size_t);
    static void  operator delete[](void*);
  };
}
```

[continued from previous page]

[8] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++* (Addison-Wesley, 1995).

[9] Kevlin Henney. "From Mechanism to Method: Total Ellipse," *C/C++ Users Journal C++ Experts Forum*, March 2001, www.cuj.com/experts/1903/henney.htm

[10] John Lakos. *Large-Scale C++ Software Design* (Addison-Wesley, 1996).

[11] Robert C. Martin. "Object-Oriented Design Quality Metrics: An Analysis of Dependencies," *ROAD*, September-October 1995, www.objectmentor.com

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

[13] Herb Sutter. "Sutter's Mill: Virtuality," *C/C++ Users Journal*, September 2001.

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).

[15] Robert B. Murray. *C++ Strategies and Tactics* (Addison-Wesley, 1993).

[16] Herb Sutter. *Exceptional C++* (Addison-Wesley, 2000).

[17] Kevlin Henney. "From Mechanism to Method: Function Follows Form," *C/C++ Users Journal C++ Experts Forum*, November 2000, www.cuj.com/experts/1811/henney.htm

[18] SGI Standard Template Library Programmer's Guide, www.sgi.com/tech/stl/

Meyers forces heap-only creation by making the object's destructor private (or protected if you want to be able to inherit from the object) and implementing a disposal method. For example:

```
namespace Meyers {
  class HeapOnly {
  private:
    ~HeapOnly() {}
  public:
    // Disposal Method
    void Destroy() { delete this; }
  };
}
```

If an attempt is made to create this object on the stack:

```
int main() {
  using namespace Meyers;
  HeapOnly heapOnly;
}
```

the compiler will issue an error. However, if the object is created on the heap:

```
int main() {
  using namespace Meyers;
  HeapOnly* pHeapOnly = new HeapOnly;
  pHeapOnly->Destroy();
}
```

the compiler is perfectly happy (gcc 3.2.3 [8] issues a warning about `HeapOnly` only having a private destructor and no friends).

In terms of forcing objects to be created on the heap, Meyers's example does exactly what is needed. However, it does have one drawback. The developer using the heap-only object must explicitly call `Destroy` in order to prevent a memory leak. Wouldn't it be better if when the pointer pointing to the heap-only object went out of scope it destroyed the object automatically? After all that is what this article is about! Enter smart pointers!

Managing Heap-Only Objects with Smart Pointers

With the exception of developing this managed heap-only object, I don't remember the last time I explicitly used the `delete` keyword. Every time I create an object on the heap I use a smart pointer to prevent a memory leak. Generally, I use Boost's [9] `scoped_ptr` and `shared_ptr`, although there are other smart pointers available, including the reference counted smart pointer described by Meyers at the end of More Effective C++ [7].

So what happens if `shared_ptr` is used with Meyers's heap-only object?

```
int main() {
  using namespace Meyers;
  boost::shared_ptr<HeapOnly>
            pHeapOnly(new HeapOnly);
}
```

The compiler reports an error because `shared_ptr` cannot call `HeapOnly`'s private destructor. Herb Sutter discusses some ways of getting around this in his CUJ [10] article Befriending Templates [13]. However, I prefer a different approach suggested by an `accu-general` regular, James Slaughter, which requires some changes to Meyers's `HeapOnly` object.

So, let's start with a new object, a managed object rather than a heap-only object:

```
class Managed {
private:
  ~Managed() {}
public:
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
};
```

`shared_ptr` can take a second constructor argument, which is a function pointer used to destroy the object. To accommodate this, the managed object's `Destroy` member function is now static and has a single parameter which is a pointer to the managed object. The pointer is used to delete the object instead of `delete this`. The lifetime of the managed object can now be fully managed by the shared pointer like this:

```
int main() {
  boost::shared_ptr< Managed >
      pManaged(new Managed, Managed::Destroy);
}
```

The above syntax isn't as nice as it could be and the developer using the managed object must still make a conscious decision to use the smart pointer and must also know that `shared_ptr`'s constructor can take a second parameter. Wouldn't it be better if the developer could only use the smart pointer? Some would argue that restricting a developer is bad. I agree in a lot of instances, but not this one, when the aim is to have a heap-based managed object that cannot leak memory.

So, how can a developer be forced to use a smart pointer? The answer is to make all the managed object's constructors, including the copy-constructor, private (or protected if the you want to inherit from the object) and provide a static factory method that returns a smart pointer (obviously if the managed object has a number of different constructors then a factory method that takes the appropriate parameters must be added for each):

```
class Managed {
public:
  typedef boost::shared_ptr< Managed > SmartPtr;
  // Factory Method
  static SmartPtr Create() {
    return SmartPtr(new Managed, Destroy);
  }
private:
  Managed() {}
  ~Managed() {}
  // Disposal Method
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
  Managed(const Managed&);
};
```

An instance of the managed object can then be created in the following way:

```
int main() {
 Managed::SmartPtr pManaged = Managed::Create();
}
```

Although we now have a managed C++ object, the solution still isn't as versatile as it could be. What if the developer using the managed object is not able to use Boost (possibly for some commercial reason) or would like to use a custom smart pointer or a smart pointer that offers thread safety? Any type of memory management smart pointer can be used with the managed object, as long as it supports assignment and can take a disposal method as a second constructor parameter. The introduction of a template template parameter [12] allows smart pointers to be interchanged easily.

```
template< template< class > class SmartPtrT >
class Managed {
public:
  typedef SmartPtrT< Managed > SmartPtr;
  static SmartPtr Create() {
    return SmartPtr(new Managed, Destroy);
  }
private:
  Managed() {}
  ~Managed() {}
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
  Managed(const Managed&);
};
```

The syntax used to create the parameterised managed object can be simplified by introducing a typedef:

```
int main() {
  typedef Managed< boost::shared_ptr >
                                    Managed;
  Managed::SmartPtr pManaged =
                          Managed::Create();
}
```

Ok, so now we have a managed object that can be used with different smart pointers. However, this solution requires a lot of extra code to be added to each new managed class written. Let's have a look at some boilerplate code that can be used to simplify the conversion of regular objects into managed objects.

Managed Object Boilerplate Code

First we need a regular object to convert to a managed object. An abstract base class pointer usually points to an object created on the heap. Therefore in this example I am going to use the simple class hierarchy that the thin database access layer described above provides, as it consists of an interface class with a single pure virtual member function, and a concrete class that implements the base class's member function:

```
class IConnection {
public:
  virtual ~IConnection() {}
public:
  virtual void ExecuteSQL(const std::string&
                                    sql) = 0;
};
```

```
class Connection : public IConnection {
public:
  explicit Connection(const std::string&
                            connectionString) {
    std::cout << "Connection String: "
              << connectionString << "\n";
  }
  virtual ~Connection() {
    std::cout << "Disconnecting\n";
  }
  virtual void ExecuteSQL(const std::string&
                                       sql) {
    std::cout << "Execute: " << sql << "\n";
  }
};
```

`IConnection` cannot be created on the stack or the heap, as it has a pure virtual member function. `Connection` can be created on the stack or the heap, as it has the compiler generated default public constructor and destructor (see item 45 of Scott Meyers' Effective C++ [14]), and no pure virtual member functions.

For `Connection` to be fully managed it must be prohibited from being created on the stack or on the heap, other than via a factory method. Preventing heap creation is easy, as discussed above; all that needs to be done is to define private or protected implementations of `operator new` and `operator new[]`. The factory method can then use these implementations to create an instance of `Connection` on the heap.

There are two ways of preventing `Connection` from being created on the stack: As discussed above, it can be given private or protected constructors and destructors or a pure virtual member function.

As the intention is to create boilerplate code `Connection` itself should be modified as little as possible. Inheriting from a managed base class is a simple way of adding (and documenting) managed behaviour. The following `ManagedBase` class defines a protected `operator new` and `operator new[]` (with their corresponding `delete` operators), disposal method, virtual destructor to ensure proper deletion, and pure virtual member function:

```
class ManagedBase {
private:
  template< typename ConcreteT,
            typename InterfaceT,
            template< class > class SmartPtrT >
  friend class Managed;
  class OnlyAvailableToManaged {};
  virtual void ForceUseOfManaged (const
            OnlyAvailableToManaged&) const = 0;
  template< template< class > class SmartPtrT,
             class InterfacePtrT,
             class ConcretePtrT,
             class ConcreteTypeT >
  friend class SmartPtrPolicyTraits;
protected:
  virtual ~ManagedBase() {}
  static void Destroy(const ManagedBase*
                                pManagedBase) {
    delete pManagedBase;
  }
```

```
  static void* operator new(size_t size) {
    return ::operator new(size);
  }
  static void operator delete(void *pObject) {
    ::operator delete(pObject);
  }
  static void* operator new[](size_t size) {
    return ::operator new[](size);
  }
  static void operator delete[](void
                                  *pObject) {
    ::operator delete[](pObject);
  }
};
```

In practice there is nothing to stop the creator of `Connection` from declaring its constructors and destructors public, and therefore negating that method of preventing stack creation. If a pure virtual member function is used, which in its simplest form would have a return value of void and exactly zero parameters, the creator of `Connection` could allow stack creation by overriding it and giving it a body.

One solution to the problem presented by using a pure virtual member function to prevent stack creation was suggested in a private email from Mikael Kilpelainen. If the pure virtual member function (`ForceUseOfManaged`) has a parameter (`OnlyAvailableToManaged`) of a type that is private to the class in which the pure virtual member function is declared, it is possible to restrict the classes that can override the pure virtual member function to friends of the class in which it is declared.

Therefore, the only way that the creator of `Connection` can override the pure virtual member function declared in `ManagedBase` is to make `Connection` a friend of `ManagedBase`. This cannot be done without modifying `ManagedBase`. The knock-on effect is that another class must be provided that derives from `Connection` and is a friend of `ManagedBase`. This class is discussed next.

`ManagedBase` lacks the necessary factory method. Although it is possible for a base class to create an instance of its subclass [15] a factory method is not present in `ManagedBase`. The factory method is a member of another class, `Managed`:

```
template< typename ConcreteT,
          typename InterfaceT,
          template< class > class SmartPtrT >
class Managed : private ConcreteT {
public:
  typedef SmartPtrT<InterfaceT> InterfacePtr;
  typedef SmartPtrT<ConcreteT>  ConcretePtr;
private:
  ~Managed() {}
public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
                  ::Initialise(new Managed);
  }
};
```

`Managed` has a private destructor to prevent stack creation. As `Managed`'s factory method will create an instance of `Connection` (`ConcreteT`) and return a smart pointer (`SmartPtrT`) of type `IConnection` (`InterfaceT`), all three must be specified as template parameters.

`Managed` inherits from `ConcreteT` so that it can override and define a body for the pure virtual function (`ForceUseOfManaged`) defined in `ManagedBase`. The smart pointers for the types `ConcreteT` and `InterfaceT` are typedef'd for convenience.

The factory method (`Create`) uses traits [11] & [12] to enable initialisation code for different smart pointers to be added easily. The factory method creates an instance of `Managed` on the heap, and passes a pointer to it to the trait `Initialise` function. The `Initialise` function returns a copy of the initialised smart pointer, which is then returned by the factory method.

When calling the trait `Initialise` function the smart pointer type must be specified so that the compiler knows which trait to use. The interface type and concrete smart pointer type and the concrete type must also be specified as these are used by the `Initialise` function. The traits template is defined as follows:

```
template< template< class > class SmartPtrT,
          class InterfacePtrT,
          class ConcretePtrT,
          class ConcreteTypeT >
class SmartPtrPolicyTraits;
```

A partial specialisation [12] is used to define the way in which Boost's `smart_ptr` is initialised:

```
template< class InterfacePtrT,
          class ConcretePtrT,
          class ConcreteTypeT >
class SmartPtrPolicyTraits< boost::shared_ptr,
                            InterfacePtrT,
                            ConcretePtrT,
                            ConcreteTypeT > {
public:
  static InterfacePtrT
       Initialise(ConcreteTypeT* pConcrete) {
    return ConcretePtrT(pConcrete,
                        &ManagedBase::Destroy);
  }
};
```

`Initialise` needs to be able to access `Destroy`, the protected static member function defined in `ManagedBase`. One way to allow this would be to make `Destroy` public, but then it could be called from anywhere. `Destroy` should only be called by the specified smart pointer. The alternative solution is to declare `SmartPtrPolicyTraits` a friend of `ManagedBase`. The required friend template is shown in the definition of `ManagedBase` above, but here it is again:

```
class ManagedBase {
protected:
  template< template< class > class SmartPtrT,
            class InterfacePtrT,
```

```
            class ConcretePtrT,
            class ConcreteTypeT >
  friend class SmartPtrPolicyTraits;
  ...
};
```

All the boilerplate code is now in place. In order to make `Connection` a managed object it must be modified to inherit from `ManagedBase`:

```
class Connection : public ManagedBase,
                   public IConnection {
public:
  explicit Connection(const std::string&
                      connectionString) {
    std::cout << "Connection String: "
              << connectionString << "\n";
  }
  ...
};
```

There is still one outstanding problem. `Connection`'s constructor takes a single argument and `Managed` does not have the appropriate constructor or factory method. There are at least two possible solutions to this problem; each has its advantages and disadvantages.

The first is to create a partial template specialisation of `Managed` specifically for `Connection`, which has the appropriate constructor and factory method:

```
template< typename InterfaceT,
          template< class > class SmartPtrT >
class Managed< Connection,
               InterfaceT,
               SmartPtrT >
         : private Connection {
public:
  typedef SmartPtrT<InterfaceT> InterfacePtr;
  typedef SmartPtrT<Connection>  ConcretePtr;
private:
  explicit Managed(const std::string&
                           connectionString)
        : Connection(connectionString) {}
  ~Managed() {}
public:
  static InterfacePtr Create(const
            std::string& connectionString) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 Connection >
             ::Initialise(new
                  Managed(connectionString));
  }
};
```

The advantage with this solution is that you can control how the constructor parameter is passed to `Create` and on to `Managed`'s constructor. For example, if for some reason `Connection` modified `connectionString`, the partial template specialisation allows the parameter to be specified as and passed by (non-const) reference. The disadvantage of this solution is that it is a lot of extra code, as it is effectively a reimplementation of `Managed`.

The second solution is to give `Managed` a number of templated constructors and `create` function pairs. For example:

```
template< typename ConcreteT,
          typename InterfaceT,
          template< class > class SmartPtrT >
class Managed : private ConcreteT {
  ...
private:
  explicit Managed() : ConcreteT() {}

  template< typename A >
  explicit Managed(const A& a)
        : ConcreteT(a) {}

  template< typename A, typename B >
  explicit Managed(const A& a, const B& b)
        : ConcreteT(a, b) {}
  ...
public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
             ::Initialise(new Managed);
  }

  template< typename A >
  static InterfacePtr Create(const A& a) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
             ::Initialise(new Managed(a));
  }

  template< typename A, typename B >
  static InterfacePtr Create(const A& a,
                             const B& b) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
             ::Initialise(new Managed(a, b));
  }
};
```

The advantage is that extra code is not needed for objects that have different numbers of constructor parameters, assuming that none of the objects have a number of constructor parameters greater than the constructor and `Create` pair with the highest number of parameters. The disadvantages are that you cannot control how the parameters are passed to `Create` and onto the constructor, as you can by using the partial template specialisation and that you cannot use an object that has more

constructor parameters than the constructor and `Create` pair with the highest number of parameters unless you modify `Managed`. There is, however, a workaround for the second disadvantage using the boost [9] `PreProcessor` library, shown to me by Paul Mensonides:

```
#include <boost/preprocessor/arithmetic/inc.hpp>
#include <boost/preprocessor/repetition/
                        enum_binary_params.hpp>
#include <boost/preprocessor/repetition/
                        enum_params.hpp>
#include <boost/preprocessor/repetition/
                        repeat.hpp>

#ifndef MAX_ARITY
#define MAX_ARITY 1
#endif

template< typename ConcreteT,
          typename InterfaceT,
          template< class > class SmartPtrT >
class Managed : private ConcreteT {
public:
  typedef SmartPtrT<InterfaceT> InterfacePtr;
  typedef SmartPtrT<ConcreteT>  ConcretePtr;
private:
  explicit Managed() : ConcreteT() {}

#define CTOR( z, n, _) \
  template< BOOST_PP_ENUM_PARAMS( \
           BOOST_PP_INC(n), typename A) > \
  explicit Managed(BOOST_PP_ENUM_BINARY_PARAMS \
                  (BOOST_PP_INC(n), \
                  const A, & p)) \
        : ConcreteT(BOOST_PP_ENUM_PARAMS( \
                   BOOST_PP_INC(n), p)) {} \
  /**/
  BOOST_PP_REPEAT(MAX_ARITY, CTOR, ~)
#undef CTOR

  virtual ~Managed() {}
  virtual void ForceUseOfManaged(const
       ManagedBase::OnlyAvailableToManaged&)
       const {}
public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
               ::Initialise(new Managed);
  }

#define CREATE(z, n,) \
  template< BOOST_PP_ENUM_PARAMS( \
            BOOST_PP_INC(n), typename A) > \
  static InterfacePtr \
    Create(BOOST_PP_ENUM_BINARY_PARAMS( \
                          BOOST_PP_INC(n), \
                          const A, &p)) { \
      return SmartPtrPolicyTraits< SmartPtrT, \
                                   InterfacePtr, \
                                   ConcretePtr, \
                                   ConcreteT > \
             ::Initialise(new \
               Managed(BOOST_PP_ENUM_PARAMS( \
                     BOOST_PP_INC(n), p))); \
    } \
    /**/
    BOOST_PP_REPEAT(MAX_ARITY, CREATE, ~)
  #undef CREATE
  };
```

The above code tells the C++ pre-processor to generate `MAX_ARITY` constructor and `Create` function pairs. For more information on exactly how it does this, consult the boost `PreProcessor` library documentation on the Boost [9] website.

There is no need to choose between either the constructor and `Create` function pairs solution or the partial template specialisation solution. Both solutions coexist quite happily together. The compiler will choose the partial template specialisation solution over the templated constructor and `Create` function pairs solution. Therefore you should use the templated constructor and `Create` function pairs solution by default, when the constructor parameters of your object are passed by const reference and write a partial template specialisation when your object requires constructor parameters to be passed by something other than const.

Finally we are at the stage where instances of our managed object can be created. The syntax used to create a managed object can be greatly simplified using another typedef:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
          ManagedConnection;
  ManagedConnection::InterfacePtr
      pConnection = ManagedConnection::Create(
                      "Connection String");
  pConnection->ExecuteSQL(
                   "SELECT * FROM MyTable");
}
```

Now that we have our managed object, we should check that it behaves in the way we expect. We have already seen that it can be created on the heap via the factory method; let's see if it can be created on the heap using `new`:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
          ManagedConnection;
  ManagedConnection::InterfacePtr
      pConnection
          = new ManagedConnection(
                      "Connection String");
  ...
}
```

No. The compiler gives an error stating that it cannot access `operator new` and `operator delete` in `ManagedConnection` as they are private. So far so good; What about creating the managed object on the stack:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
                         ManagedConnection;

  ManagedConnection connection(
                        "Connection String");
}
```

No, this doesn't work either. The compiler gives an error, stating that it cannot access the private constructor and destructor in `ManagedConnection`. Ok, what about creating the `Connection` object directly on the heap:

```
int main() {
  IConnection* pIConnection =
         new Connection("Connection String");
}
```

No, this doesn't work. The compiler complains that it cannot instantiate `Connection` as it is an abstract class and that it cannot access `Connection`'s `new` and `delete` operators as they are private. What about creating `Connection` directly on the stack?

```
int main() {
  Connection
         connection("Connection String");
}
```

This final test doesn't work either. The compiler complains that it cannot create `Connection` as it is an abstract class. Making `Connection` static or a class member does not work either, for the same reason.

These five tests demonstrate that our managed object works as expected.

Finally

In this article I have discussed how to use smart pointers to prevent memory leaks and ways of forcing objects to be created on the heap only, while also preventing the possibility of a memory leak.

I expect this technique to be useful not only to library writers but also to general application writers who are concerned about possible misuse of their objects by other people.

I also hope this will appeal as a viable alternative to people thinking about Microsoft's Managed C++ as a solution to their memory leak problems.

Paul Grenyer
paul@paulgrenyer.co.uk

References:

[1] Aeryn: http://www.paulgrenyer.co.uk/aeryn/

[2] *Design Patterns: elements of reusable object-oriented software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, ISBN 0201633612

[3] The method of creating objects at the highest level and then passing some form of reference around has come to be known in some circles as Parameterise from Above and is a much talked about work in progress by Kevlin Henney:
http://www.java.no/web/moter/javazone03/presentations/KevlinHenney/Programmer_s%20Dozen.pdf

[4] *Reference Counting*, Kevlin Henney:
http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ReferenceAccounting.pdf

[5] *Factory and Disposal Methods*, Kevlin Henney:
http://www.two-sdg.demon.co.uk/curbralan/papers/vikingplop/FactoryAndDisposalMethods.pdf

[6] *Executing Around Sequences*, Kevlin Henney:
http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf

[7] *More Effective C++*, Scott Meyers, Addison Wesley, ISBN 020163371X

[8] gcc 3.2.3: http://gcc.gnu.org/

[9] Boost: www.boost.org/

[10] CUJ: http://www.cuj.com/

[11] *Traits: a new and useful template technique*, Nathan C. Myers:
http://www.cantrip.org/traits.html

[12] *C++ Templates: The Complete Guide*, David Vandervoorde, Nicolai M. Josuttis, Addison Wesley, ISBN 0201734842

[13] *Befriending Templates*, Herb Sutter,
http://www.cuj.com/documents/s=8244/cujcexp2101sutter/

[14] *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, Addison Wesley, ISBN 0201924889

[15] *Data Abstraction and Heterarchy*, Kevlin Henney,
http://www.cuj.com/documents/s=7992/cujcexp1908henney/

Acknowledgments

Thanks to: James Slaughter, Adrian Fagg, Phil Nash, Mark Radford, Kevlin Henney, Allan Kelly, Mikael Kilpelainen, Jennifer Hart, Paul Mensonides, Tim Pushman