overload 113

FEBRUARY 2013 £3

Secrets of Testing WCF Services

How to create testable
WCF applications

Top Five C++ Cooking Recipes

A number of recipes for
solving C++ problems

Utilising More Than 4GB of Memory in a 32-bit Windows Process

Techniques to access
more memory

On Signs of Trouble

The use of patterns can be
controversial. We evaluate their
cognitive and social value.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 113

February 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 114 should be submitted by 1st March 2013 and for Overload 115 by 1st May 2013.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

The Good, The Bad and The Discordant

There are several signs of bad code. Frances Buontempo considers ugliness, stench and discord in a search for beauty.

First I must apologise. I have failed to write an automatic editorial generator. I was distracted by @TicBot on twitter and after many delightful chats about biscuits, it is clear I should write an @OverloadBot and leave them to figure it out between themselves. I will try harder for the next edition. Meanwhile, I will step back and consider what makes something delightful or indeed beautiful for motivation. Let us start by considering its contrapositive. What makes something repulsive or ugly?

People frequently refer to code smells, indicating that the code in front of their eyes falls somewhere between inducing a nervous tick or the desire to run away screaming and inducing a sense of general, or specific, wrongness. Referring to a smell produced by looking at something is decidedly odd, though possibly traditional. Martin Fowler claims the phrase was first coined by Kent Beck. Furthermore he says, “A smell is by definition something that’s quick to spot” [Fowler]. Perhaps you can spot things with your nose as well as your eyes, though our eyes will show us the usual suspect smells including code duplication, long functions and many similar subclasses. There are other types of code smells that we neither sniff nor see, for example boredom: “Boredom is a smell that you understand the problem well enough to automate a solution” [c2].

It is possible to extend the heuristic of bad smells for code analysis to other senses. This brings to mind some mention of a program called CAITLIN, ‘musical program auralisation’ system for Pascal programs, mapping program data to sound. [Vickers02] Many years ago in an attempt to speed up a program running on an embedded device, a colleague and I inserted beep statements in a loop to see how often a function was called. This was a resounding success, though we were banished to the kitchen since the rest of the team found it deeply annoying. I suspect CAITLIN goes further than simply beeping. It draws on previous research that could be used to represent C programs, and automatically maps loops and structures to sound motifs or short tunes. The authors notice that a musical background made no difference to the results. Regardless of musical expertise the sound motifs seemed to help students detect bugs more quickly. Music would appear to have deep roots in the psyche. Dara Ó Briain’s latest *Science Club* program on BBC2 explored the science behind music, noting that music, specifically the beat, could have a profound impact on people suffering from Parkinson’s disease [Science Club]. The rhythm helped them walk more stably, possibly by allowing a different path through the brain to be formed, short circuiting the broken parts. Music seems to form some kind of universal language and can immediately cause visible delight in young children. Analysing source code aurally therefore seems like a very good idea.

We have considered smells and discord to detect bad and ugly code. We have also noticed that our eyes can tell us much about a code base. Furthermore, there are many code metrics, which will summarise the code space numerically, ranging from test coverage to complexity and beyond. An obvious next step is to graph these, to provide a neat summary. A quick google found NDepend’s Metric View [NDepend], which appears to represent a code base as coloured rectangles. I would need to try this on a code base or two to get a feel for what what counts as good or bad, from this visual perspective.

What makes code good? Certainly it should be fit for purpose, and not noticeably contain bugs. This doesn’t go far enough though. It is possible to automatically generate code to perform tasks, but the generated code is frequently ugly. Of course, we do not yet have a clear definition of ugly though mentions of stinks and discord provide a vague idea. The time has come to flip back our contrapositive and reconsider what is beautiful or delightful. Recall the title of Knuth’s classic algorithms books, the *art* of computer programming. Why does Knuth insist on using the word art, considering so many of us frequently try to suggest we are computer scientists or engineers? “We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty” [Knuth74]. What on earth does it mean to claim a computer program is beautiful? An acquaintance of mine is currently trying to learn python and cannot see why I insist on suggesting different ways of doing something. If it works, it works. Full stop, end of. No discussion necessary. Which makes trying to discuss why I feel this matters somewhat challenging. As a person with a background in pure mathematics I have been trained to appreciate beauty. It is possible to write a long, laborious and difficult to follow proof in analysis. Point proved, QED, full stop, end of. However, it is often possible to prove the same thing formulated in topological terms. Clearly, this is total waste of time if you’re just trying to prove a point. Nonetheless, the topological version will frequently be two or three lines long. The compactness is made possible partly by the level of abstraction introduced. There is a fine balance between terseness and elegance, and beauty sits firmly with elegance and simplicity. Controversially, I would like to suggest spotting beauty is learnt. People use phrase like a ‘trained eye’ or ‘trained ear’. Perhaps experienced programmers develop a trained nose, to spot code smells. Algebra is beautiful, but most school children cry dismay when they first meet the subject, complaining that mathematics should be about numbers rather than letters. The delight and joy that comes from appreciating algebra is not innate; it tends to come from a long hard slog and good mentors. So, to re-ask the question, Victor Norman asks “But, what difference does it make that the code is ugly? If the code works correctly, who should care that it is ‘ugly’?” [Norman]. As with my python

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad’s BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

learning friend complaining that if it just works that's good enough, how can we answer this point? Returning to musical program auralisation, most children will engage with simple music, perhaps in the form of nursery rhymes and will take a long time to appreciate other more complicated musical motifs. Depressingly, many will stick with the simple obvious forms and never progress on from the inane to the beautiful. Does this matter? Almost certainly not, however, they are missing much joy and delight, in the same way that people who never get the hang of algebra are missing out.

Why does beauty matter? Perhaps it doesn't really, though if something is clear, elegant and simple it might be easier to work with. Let us return Knuth again.

> Alas, people these days rarely measure a computer scientist by standards of beauty and interest; they measure us by dollars or by applications rather than by contributions to knowledge, even though contributions to knowledge are the necessary ingredient to make previously unthinkable applications possible. [Knuth interview]

Perhaps we could argue beautiful code is easier to test and refactor and then measure this in dollars. More abstractly, Norman [op cit] suggests that art and computer programs are both about communication. We delved into communication in *Overload 112*, though from the viewpoint of developing software as communications rather than in a search for beauty. An expert musician or artist will notice patterns and structures that a novice may miss thereby failing to appreciate some of the elements of cleverness or joy in the composition. Similarly the structure of a program may follow a design pattern, which a newbie may not spot. Beauty goes beyond communication though. Introducing a level of abstraction, through algebra, repeating sound motifs or developing new data structures and algorithms can open up new possibilities. Recall the surprise at the discovery of the possibilities of template metaprogramming? Being able to read, write and think in a specialist area, be it music, mathematics, code or even a human language is the goal of a skilled artisan. "Fluency goes beyond reading and writing. A fluent speaker of a language begins to be able to think in idiom" [Armitage]. Armitage goes on to say of writing code:

> A great deal of it is much more like sculpture. Data, technology, code, as a slab of clay, to be manipulated, explored, felt between your fingers, and slowly turned into something substantive. It's practically the opposite of engineering. It's an artistic discipline: beginning with sketching and exploring, and then building on those sketches slowly through iteration, watching a final structure emerge.

Perhaps we should avoid the age-old debate about whether what we do is engineering or not, but rather observe that taking pride in creating something beautiful should be possible whichever side of the debate you come down on. He then suggests that computers allow us to think new thoughts. On the face of it, this is a remarkable claim. However, frequently introducing new representations for ideas, in effect new languages and levels of indirection, allows new ideas and new ways of solving problems to emerge. Introduction of the calculus gave ways of expressing both old and new problems in a different way. Would we have 'e' without Newton or Leibniz?

Code can be good, bad or ugly. Indeed, correct code, fulfilling user requirements can be good, bad or ugly. Trying to define 'good' code is difficult, but bad and ugly code is easy to spot. It smells. It makes a discordant racket. It possibly burns your skin or tastes like wallpaper paste, though we haven't yet seen a way of mapping it to touch or taste. Certainly, it looks ugly. It could be long and lumbering functions, with slightly more boolean flags wedged in than you would expect, or using two such flags to describe three possible states and so on. It is easy to list bad and ugly traits shared by bad and ugly code. Instead, let us end by considering good traits shared by beautiful code. I suggest it should be as simple as possible, but no simpler. If it leaves you excited and wanting to play with an idea further it might be a masterpiece. Trying to form a definitive list of what makes something beautiful or even good may be an impossible task, so I'll leave each reader to consider what they find beautiful.

> *Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write grand programs, noble programs, truly magnificent ones!*
> [Knuth74]

References

[Armitage] http://www.bbc.co.uk/news/technology-20764273

[c2] http://www.c2.com/cgi/wiki?BoredomIsaSmell

[Fowler] http://martinfowler.com/bliki/CodeSmell.html

[Knuth interview] http://www.simple-talk.com/opinion/geek-of-the-week/donald-knuth-geek-of-the-week/

[Knuth74] 'Computer Programming as an Art.' *Communications of the ACM* 1974 pp 667–673 http://delivery.acm.org/10.1145/370000/361612/a1974-knuth.pdf?ip=139.149.31.231&acc=OPEN&CFID=243683453&CFTOKEN=93039526&__acm__=1357131190_a70eb5ac6b56861bb9cf8cada5c18e9f

[NDepend] http://www.ndepend.com/Doc_Treemap.aspx

[Norman] http://cs.calvin.edu/documents/christian/BeautyCompProg.pdf

[Science Club] http://www.bbc.co.uk/mediacentre/proginfo/2013/01/dara-o-briain-science-club.html

[Vickers02] Vikers & Alty 'When bugs sing' Paul Vickers, James L. Alty 2002 http://hdl.handle.net/2134/3357

'No Bugs' Top Five C++ Cooking Recipes

Developers often have a few favourite tricks for solving problems. Sergey Ignatchenko shares his five top recipes.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or the *Overload* editor. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented providing an exact translation. In addition, both the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Today, I've decided to recall that I'm a developer (at least at heart), and pause my usual philosophical blabber about the place of developers in the Universe to talk about actual programming a little bit.

Most (if not all) of the stuff in this article is rather well-known, however, such things are frequently overlooked, so I feel that they are worth mentioning once again.

Recipe #5 – set<>, sort(), and 'strict weak ordering'

With **sets**/**maps**/**sorts** there is a well-known headache: how to write a compliant **operator <()** (or a functor). It is known that for classes involved in **map<>**/**set<>**, **operator <()** *must* comply to a so-called 'strict weak ordering'; if this is violated, all kinds of weird things can happen (from identical multiple entries in a supposedly-unique collection, to memory corruption). Unfortunately, looking at an arbitrary **operator <()**, it is usually rather (or very) difficult to tell if it complies with 'strict weak ordering'. Recipe #5 shows how to cook an **operator <()** which does comply with 'strict weak ordering' and therefore can be safely used with **sets**/**maps** (and also with **sorts** etc.) – see Listing 1.

```
class X {
  int i;
  int j;
  string s;
  Y y;
  ZZ zz;

  bool operator <( const X& other ) const {
    if( i != other.i )
      return i < other.i;
    if( s != other.s )
      return s < other.s;
    //it is ok to ignore some fields
    return false;
  }
};
```

Listing 1

operator <() in Listing 1 is guaranteed to comply with 'strict weak ordering' (assuming that nobody has redefined comparison for **strings**).

If necessary, class members can also be included into the comparison (provided that their respective **operators <()** are also compliant with 'strict weak ordering'):

```
  bool yl = y < other.y;
  bool yg = other.y < y;
  if( yl || yg )// this is just a way to write
                // "y != other.y"
    return yl;
```

In addition, functions can also be used (as long as their arguments are constant):

```
int fthis = f( i );
int fother = f( other.i );
if( fthis != fother )
  return fthis < fother;
```

Multi-parameter functions are also possible (as long as all arguments are constant, and as long as they're used exactly as shown below):

```
  int fthis = f( i, s );
  int fother = f( other.i, other.s );
  if( fthis != fother )
    return fthis < fother;
```

It is very important to note that while in any of the above examples, 'strict weak ordering' is guaranteed, any deviations from the forms described above, can be deadly. For example, if you replace **return false;** with innocent-looking **return true;**, it would break the code (as for certain **a** and **b** our **operator <()** would return that **a < b** is true, and simultaneously that **a > b** is true, which is obviously wrong). In another example, comparing **f(i, j)** with **f(other.j, other.i)** would be risky (while comparing **f(i, j)** with **f(other.i, other.j)** is guaranteed to be ok under the conditions stated above).

Recipe #4 – Measuring run-time performance

In production code, it is often desirable to gather statistics to measure code performance under real conditions. The problem is how to gather statistics without hurting performance too much. Here are a few tricks to help deal with this issue.

First, for statistical purposes it is usually not necessary to use any kind of thread synchronization. Yes, if you're writing plain **stats.nCalls++** without using mutexes or atomics, you might get an error if two threads try to modify the same **stats.nCalls** simultaneously; however, if all the statements modifying **stats.nCalls**, are incremental ones (like the one above) the error from a single race condition cannot exceed 1. In addition, the chance of race conditions is very slim (even if you try really hard, getting more than 1/100 of assignments to have race conditions would be difficult, and in practice it is usually more like 1e-4 to 1e-5 or

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

if you have lots and lots of calls to your function, it is often ok to take time measurements using a low-precision timer

so). We can usually say (given enough calls) that statistical errors due to race conditions are negligible.

Another issue which often arises in relation to measuring run-time performance is time measurement. The problem here is the following: usually the time frames to be measured are very small (often within a microsecond), and high-precision methods available to measure time are not always readily available. (For example, an x86 RDTSC instruction was observed to provide incorrect results on certain multi-socket hardware due to incorrect implementation of the motherboard.) So, what should you do if you need to measure how long certain function takes (in microseconds), but all you have is a very low-precision timer (like a 15-milliseconds timer)? Apparently, if you have lots and lots of calls to your function, it is often ok to take time measurements using a low-precision timer, and simply add them up (using something like `stats.callTime += deltaTime;`. It might be even better to reduce performance impact, `if( __builtin_expect( deltaTime, 0 ) ) stats.callTime += deltaTime;` – potentially without synchronization, as described above). If you're measuring a time interval of 1.5 microseconds, and have a timer which has precision of 15 milliseconds, you will get `deltaTime == 0` in 99.99% of the cases, but apparently, after averaging a million such calls, results are usually surprisingly precise (the precision I've observed in practice was about ±20%, which is much better than one would expect intuitively given the numbers above). While precision is not guaranteed and your mileage may vary, if you don't have any other options on the table – for example, because RDTSC doesn't work properly on your multi-socket hardware – it is certainly worth a try.

Recipe #3 – _set_se_translator

There is only one thing which I think is fundamentally better with the Microsoft Visual C++ compiler than with GCC, and it is the `_set_se_translator`. In many heavily loaded server-side cases, it really saved me from becoming a rabbit stew. The idea behind `_set_se_translator` is to convert SEH exceptions (like access violations, divisions by zero, etc.) into C++ exceptions, which then are handled in the usual C++ way (with destructors called etc.); details can be found on the MSDN page on `_set_se_translator` [MSDN].

In particular, such a thing is extremely useful if you have a state-machine-based server processing incoming messages. If an access violation (or division by zero) happens during the processing of one message, and it doesn't cause any memory damage (which is very common with access violations and always happens with division by zero), it is usually perfectly ok to ignore such a message and continue to serve the others without crashing.

One should note that using `_set_set_translator` and catching resulting C++ exceptions is very different from trying to catch SEH exceptions with `catch( ... )`. When trying to catch SEH with `catch( ... )`, no C++ destructors are called between the point where SEH is raised and where it is caught; this can cause all kinds of weird effects (and if you're using destructors to remove mutex locks, forget catching SEHs with `catch( ... )`). On the contrary, when using `_set_se_translator`, SEH is converted into a C++ exception right where SEH occurs, so it is a C++ exception which is thrown. All destructors from the point where SEH was raised, to the point where the C++ exception is caught, are properly called, with much better results (unless memory has already been corrupted by the point where SEH has occurred).

Unfortunately, the last time I checked (which was admittedly a few years ago) similar functionality was not available for *nix C++ compilers, including GCC. Theoretically, it might be possible to throw a C++ exception from the signal handler, but this functionality is platform-dependent and in practice, I wasn't able to find a platform where it works :-(.

Recipe #2 – Containers with 'move' semantics

Recipe #2 is admittedly a rather weird one, but every good cookbook should contain at least one weird recipe, so here goes. In high-performance code, there are scenarios, where you need to have a container which stores complex objects (such objects including allocated storage etc.), but those objects are only moved around and are never copied. Providing a copy constructor for such objects can be either difficult (for example, if such an object includes a file handle) or undesirable for performance reasons (if such an object contains allocated memory, copying which would be expensive). One common way to deal with this is to use some kind of reference-counted pointer (or something similar to `auto_ptr<>`); this is a viable option, but it has the associated cost of extra allocation/deallocation, and in really high-performance code, this might be an issue. In such cases, an approach similar to the following could help (rephrasing a proverb, you could say that weird times require weird measures) – see Listing 2.

While other operations on such a container may be added in a similar way, it is extremely important to keep all such operations within this class, and not to expose any operations of an underlying 'shadow' container to outside world without ensuring the integrity of our `StackOfX` container.

Recipe #1 – asserts

And my top place goes to a very simple, but extremely useful, recipe, related to `assert`s. In the C/C++ world, everybody knows about `assert`s; the problem with a standard `assert()` is that it calls `abort()` if assertion fails, which is often a bit too much.

The idea of this recipe is very simple – to create a `MYASSERT` macro which, if violated, throws an exception.

```
#define MYASSERT( cond ) \
  (void)( ( cond ) || ( throw MyAssertException \
  (   #cond, __FILE__, __LINE__ ), 0 ) )
```

What exactly your `MyAssertException` class should be, where to place it in the exception hierarchy, and how to use file name, line number, and the assertion failure is up to you. For example, if you're concerned about revealing information about your code to the customer, you can easily omit `#cond` in the production compile, and track what has happened in

```
//class X is our complicated class

class StackOfX {
  // stack is just one example; any other type of
  // container (including maps, sets, lists, etc.)
  // can be written in a similar manner
  struct ShadowX { char data[ sizeof(X) ]; };
  typedef vector< ShadowX > ShadowV;
  ShadowV v;

  void push( /* move-in */ X& x ) {
    ShadowX& sx = (ShadowX&)x;
    v.insert( v.end(), sx );
  }
  const X& operator[]( int i ) const {
    return (const X&)v[ i ];
  }

  void pop( /* move-out */ X& x ) {
    ShadowV::iterator it = v.end() - 1;
    ShadowX& sx = (ShadowX&)x;
    sx = *it;
    v.erase( it );
  }

  ~StackOfX() {
    for( ShadowV::iterator it = v.begin();
         it != v.end(); ++it ) {
      X& x = (X&)(*it);
      x.X::~X();
    }
  }
};
```

Listing 2

production based only on file/line. (You do have tags in your source control for all the versions released to production, don't you?)

Often, several such **MYASSERT**s are introduced (**MYASSERT**, **MYASSERT2**, **MYASSERT3**, **MYASSERT4**, etc.) to indicate the level at which the check is performed. For example, you can easily create a header which would behave as follows: if you define **MYASSERTLEVEL=2**, then all **MYASSERT**s with **level <= 2** are checked, and all the others are ignored:

```
#if MYASSERTLEVEL >= 2
  #define MYASSERT2( cond )  \
  (void)( ( cond ) || ( throw MyAssertException \
  ( #cond, __FILE__, __LINE__ ), 0 ) )
#else
 #define MYASSERT2() ((void) 0)
#endif
```

Apparently, it is often a good idea to leave at least some of the **MYASSERT**s in the production code (especially if you run it on your own server or have a mechanism to send logs back to you). If you find that occasionally certain **MYASSERT**s in production mode fail, this clearly warrants an investigation.

In addition, if you're using 3rd-party libraries which use the standard **assert()** internally, it is often a good idea to replace the standard **assert()** with **MYASSERT()** to avoid situations when the whole server calls **abort()** because some assertion within a 3rd-party library has failed (while recovery was still perfectly possible). How you do it depends on the specifics of your project, but it usually can be done either via redefining standard **assert()** and recompiling the 3rd-party library, or if **assert()** calls some helper functionon your platform, this helper function can often be replaced to achieve the desired effect. ■

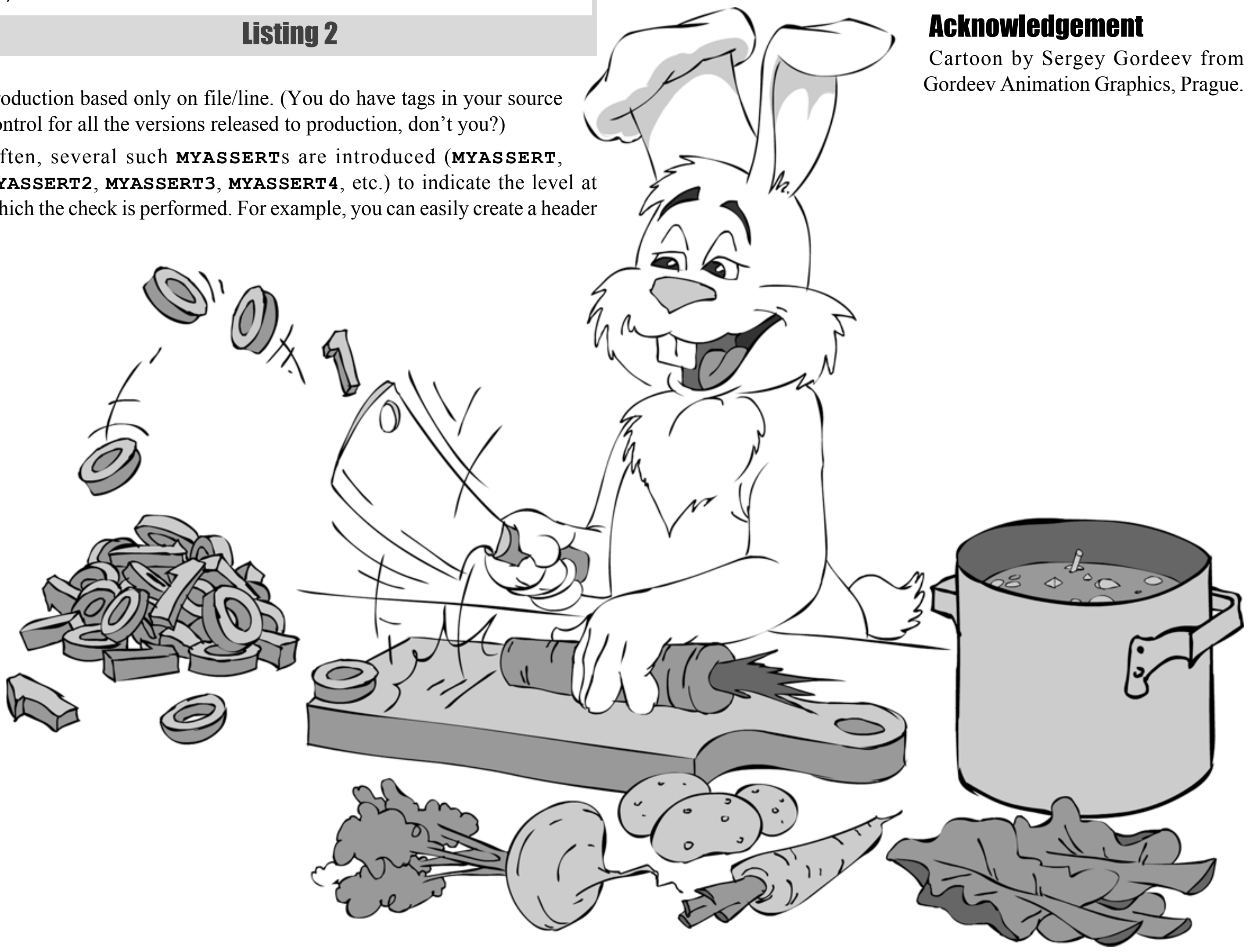
References

[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[MSDN] '_set_se_translator' http://msdn.microsoft.com/en-us/library/5z4bw5h5%28v=vs.110%29.aspx

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Utilising More Than 4GB of Memory in 32-bit Windows Process

Some applications require a vast amount of memory. Chris Oldwood presents techniques to provide extra memory.

Large scale enterprise services like SQL Server and Exchange Server can be memory hungry beasts. Given the chance, they will devour as much RAM as you can feed them, using it for caching to reduce costly I/O requests. This kind of service is often deployed on some Big Iron hardware with the sole aim of allowing it free rein of the host machine – its job being to serve clients, preferably as many as possible and in the shortest possible time.

This article will outline the various memory constraints that affect 32-bit processes on the Windows platform and the solutions that both Intel and Microsoft provide for overcoming them through hardware, OS configuration or API changes.

32-bit process memory limits

When Windows NT was first being developed back in the early 1990s you were lucky to find hard disks with a capacity over 2GB, let alone that much physical RAM. The initial design decision was to split the 4GB virtual address space that every 32-bit process would be limited to into two halves. That meant 2GB was reserved for the system (or kernel space) and 2GB for the application (or user space). Even today this address space limit of 4GB is still in effect for 32-bit processes. What the Windows engineers have done instead is provide a variety of techniques to either shuffle the kernel/user allocation ratio about or provide other APIs to allow larger memory regions to be allocated and different portions of that to be mapped into the process address space on demand [Russinovich].

Much of the confusion around this particular topic is due to the differences between the following limits: the virtual address space that a process is bound by, the amount of physical RAM that is defined by the hardware and the disk-based virtual memory provided by additional page-files. In some older articles the terms ‘memory’, ‘virtual memory’ and ‘address space’ are used interchangeably, which only compounds the confusion. So, to ensure consistency throughout this article I am going to provide clear definitions of these key constraints.

Process virtual address space

A process lives within a 4GB virtual address space. The limit mirrors that of a 32-bit pointer and is deemed ‘virtual’ because the address pointer does not refer to physical memory but is actually a logical address. Instead, the page belonging to the address can be mapped anywhere within physical RAM or even inside a page-file. This is the classic ‘Level of Indirection’ at play.

Physical memory

Naturally this is the hardware you have within your machine. Desktop editions of Windows have historically only allowed you to access up to 4GB, whereas the Data Centre Edition of Windows Server supports up to 64GB.

Page-files

The hard disk can also act as a temporary store for memory pages that are not currently in use. This is called virtual memory because it can only be used for page storage – the pages still have to be present in physical memory to be accessed.

This total size of all paging files defines the virtual memory limit for the entire machine; this can be smaller or larger than the per-process 4GB limit.

Commit charge (total memory)

If you open the Task Manager and look at the ‘Performance’ tab you will see a number of system memory figures quoted. One of them is labelled ‘Commit Charge’. This represents the sum of both the physical RAM and any space allocated via page-files. It is the total amount of memory available for all processes.

Reserved & committed process pages

Within a process the pages that constitute the virtual address space can be in one of three states: Free, Reserved or Committed. Free pages are exactly that – pages which have yet to be used. Committed pages are those that are in use and count towards the process’s footprint as they must be backed either by the page-file (for data) or the executable image (for code). The intermediate state of Reserved is a half-way house used to put a region of address space to one side without actually forcing the OS to commit any physical resources to maintaining it (except for bookkeeping).

Reserved memory is a particularly tricky beast because it is invisible in the Task Manager due to there being no physical overhead and yet it creates contention and fragmentation that is difficult to observe without inspecting the process directly.

Running out of memory

There are essentially two ways that you can run out of memory. The first is to exhaust your own process’s virtual address space by utilising all the pages within it, or making it impossible for the heap manager to find enough free contiguous pages from which to satisfy a memory allocation request. The second method involves consuming all available system memory (i.e. both physical and virtual) so that the OS cannot allocate a free memory page to any process. The implication of the latter is that a different process is the cause of a memory allocation failure – you might only be the victim.

To diagnose a process breaching its own limits you can monitor it with PERFMON.EXE and watch the Process | Virtual Bytes and Process | Private Bytes counters. The former represents the amount of virtual address space that has ever been allocated for heaps, page-file sections, executable code, etc. The latter is the number of Committed Pages which represents the footprint of the process within the total memory available to the system.

Chris Oldwood started out as a bedroom coder in the 80s, writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is also the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

Determining that the **entire machine has hit the buffers** can be a much simpler affair

The Private Bytes counter can also be seen in Task Manager under the confusingly named column 'VM Size'. Alternatively Process Explorer, via the Properties | Performance tab, provides a single dialog for a process that contains all the important memory statistics. However, it uses the term 'Virtual Size' in place of 'Virtual Bytes'. The following table maps the terms between the various common tools:-

Tool	Working Set	Commit Charge	Address Space
Task Manager	Mem Usage	VM Size	N/A
Perfmon	Working Set	Private Bytes	Virtual Bytes
Process Explorer	Working Set	Private Bytes	Virtual Size

As we shall see later when discussing the mechanisms for breaking the 4GB barrier the column name 'Commit Charge' becomes less meaningful, but it is a good first-order approximation.

The exact cause of the process's exhaustion will likely need much closer examination of the actual page usage, for which WinDbg can be of great assistance. Another more recent tool from the Sysinternals stable, called VMMap, can also be of use. The latter is more graphical in nature than WinDbg so is easier for visualisation.

Determining that the entire machine has hit the buffers can be a much simpler affair. Bring up the Performance tab in Task Manager and compare the Commit Charge 'Peak' to the 'Limit' – if they're the same you've maxed out. Things will likely start going awry before this point though. If for instance you're making very heavy use of file or network I/O, you can drain the number of System Page Table Entries, which is the pool from which everything flows. The likely indicators here are the Win32 error codes 1450 ('Insufficient system resources exist to complete the requested service') and 1453 ('Insufficient quota to complete the requested service'). Perfmon is able to help you visualise the consumption of this vital system resource via the Memory | Free System Page Table Entries counter. If you've hit either of these two conditions then your problem is not going to be solved by any of the solutions below; they may even make it worse!

Memory pressure solutions

There are a number of different options available for remedying a memory bound 32-bit process that range from simple OS level configuration changes to architectural changes via the use of certain Win32 APIs. Porting to 64-bit Windows is mentioned here as well, but only out of completeness.

Configuration based remediation

We start with the OS/process configuration based solutions as they don't require any code changes per-se.

The /3GB or /USERVA boot.ini Flag

One of the simplest ways to increase the amount of memory a process can use on 32-bit Windows is to enable the **`/3GB`** flag in the Windows' `boot.ini` file. This has the effect of adjusting the kernel/user address space split in favour of the application by 1GB, i.e. instead of a 2GB/2GB split you have a 3GB/1GB split. The downside to this is that the kernel address space is halved so there is less space for certain key kernel data structures such as the number of System Page Table Entries mentioned earlier. The **`/USERVA`** flag is an alternative to **`/3GB`** that allows for fine tuning of this ratio.

Unfortunately this magic flag is no good by itself. The increase in application address space means that all of a sudden an application could start dealing with addresses above 0x7FFFFFFF. Signed pointer arithmetic on memory allocated above this threshold could expose latent bugs that may lead to subtle data loss instead of catastrophic failure. Consequently an application or service must declare itself compatible with this larger address space by being marked with the **`/LARGEADDRESSAWARE`** flag in the executable image. This flag is accessible via the Visual C++ linker and is exposed by the later editions of the Visual Studio IDE under System | Linker | Enable Large Addresses. For .Net applications you currently need to use a custom build step that invokes `EDITBIN.EXE` to set the flag.

To aid in testing your application's compatibility with high addresses there is a flag (**`MEM_TOP_DOWN`**) that can be passed to **`VirtualAlloc()`** to force higher addresses to be allocated before lower ones (the default).

As an aside the more recent documentation from Microsoft on this topic now uses the term 4-Gigabyte Tuning (4GT) [MSDN].

Physical address extensions (the /PAE boot.ini flag)

Extending the virtual address space of a single process overcomes one limitation, but there is a second one in play on 32-bit Windows that affects your ability to run many of these 'large address aware' processes, such as in a Grid Computing environment. The maximum amount of physical memory that could be managed by Windows was also originally 4GB. This is still the case for the 32-bit desktop editions of Windows, but the server variants are able to address much more physical RAM – up to 64 GB on the Data Centre Server edition.

This has been achieved by utilising an Intel technology known as Physical Address Extensions (PAE) which was introduced with the Pentium Pro. It adds an extra layer to the page table mechanism and extends entries from 32-bits to 64-bits so that up to 128GB could theoretically be addressed.

The introduction of PAE means that kernel drivers would now also be exposed to physical addresses above the 4GB barrier, something they may not have originally been tested for. Windows tries to keep buffers under the 4GB limit to aid reliability, but once again the enablement of the feature must be a conscious one – this time via the **`/PAE`** switch also in `boot.ini`. If the server hardware supports Hot Add Memory this flag is actually enabled by default.

The danger of /3GB and /PAE

As always there is a cost to enabling this and the halving of the Page Table Index from 10-bits to 9-bits via **`/PAE`** means that there are half as many System Page Table Entries available for use. If you combine this with the **`/3GB`** flag you will have significantly reduced this resource and may see

the risk/reward for porting a line-of-business application that is only in need of a little more headroom may not be sufficient to justify the cost and potential upheaval

the server straining badly under heavy I/O load, i.e. you could start seeing those 1450 and 1453 errors mentioned earlier.

The other major casualty is the video adaptor [Chen], but this is often of little consequence as application servers are not generally renowned for their game playing abilities. Of course the rise in general-purpose graphics processing units (GPGPU) puts a different spin on the use of such hardware in modern servers.

Using 64-bit Windows to run a 32-bit process

Naturally all this **`/3GB`** and **`/PAE`** nonsense goes away under 64-bit Windows as the total system address space is massive by comparison. Although in theory you have 64-bits to play with, implementation limitations mean there are actually only 48-bits to work with. Still, 256 TB should be enough for anyone?

But, 64-bit Windows doesn't just benefit 64-bit processes; the architecture also changes the address space layout for 32-bit processes too. The kernel address space now lives much higher up leaving the entire 4GB region for the application to play with (assuming that your image is marked with the **`/LARGEADDRESSAWARE`** flag as before).

Recompiling for 64-bit

The obvious solution to all these shenanigans might just simply be to recompile your application as a 64-bit process. Better still, if you rewrite it in .Net you have the ability to run as either a 32-bit or 64-bit process as appropriate with no extra work. Only, it's never quite that simple…

There are many issues that make porting to a 64-bit architecture non-trivial, both at the source code level, and due to external dependencies. Ensuring your pointer arithmetic is sound and that any persistence code is size agnostic are two of the main areas most often written about. But you also need to watch those 3rd party libraries and COM components as a 64-bit process cannot host a 32-bit DLL, such as an inproc COM server.

The hardware and operating system will also behave differently. There are plenty of 'gotchas' waiting to catch you out during deployment and operations. In the corporate world 32-bit Windows desktops are still probably the norm with 64-bit Windows becoming the norm in the server space. So, whilst the 64-bit editions of SQL & Exchange Server are well bedded-in, custom applications are still essentially developed on a different platform.

Useable memory

Having a user address space of 2, 3, or even 4 GB does not of course mean that you get to use every last ounce. You'll have thread stacks and executable image sections taking chunks out. Plus, if you use C++ and COM you have at least two heaps competing, both of which will hold to ransom any virtual address descriptors (VADs) that they reserve, irrespective of whether they are in use or not. Throw in 'Virtual Address Space Fragmentation' and you're pretty much guaranteed (unless you've specifically tuned your application's memory requirements) to get less than you bargained for.

The following table describes my experiences of the differences between the maximum and realistic usable memory for a process making general use of both the COM and CRT heaps:-

Max User Address Space	Useable Space
2.0 GB	1.7 GB
3.0 GB	2.6 GB
4.0 GB	3.7 GB

This kind of information is useful if you want to tune the size of any caches, or if you need to do process recycling such as in a grid or web-hosted scenario. To see the amount of virtual address space used by a process you can watch the 'Virtual Bytes' Perfmon counter as described earlier.

Extending your footprint over 4GB

Those who went through the 16-bit to 32-bit Windows transition will no doubt be overly cautious – the risk/reward for porting a line-of-business application that is only in need of a little more headroom may not be sufficient to justify the cost and potential upheaval straight away.

If it's caching you need, and you don't mind going out-of-process on the same machine (or even making a remote call) then there are any number of off-the-shelf products in the NOSQL space, such as the open source based Memcached. However, if you're looking to do something yourself and you want to avoid additional dependencies, or need performance closer to in-process caching, then there are two options: Address Windowing Extensions and Shared Memory.

What you need to bear in mind though is that it's not possible to overcome the 4 GB address space limit, but what both these mechanisms allow is the ability to store and access more than 4GB memory very quickly – just not all at exactly the same time.

Address Windowing Extensions (AWE)

Windows 2000 saw the addition of a new API targeted specifically at this problem, and it is the one SQL Server uses. The AWE API is designed solely with performance in mind and provides the ability to allocate and map portions of the physical address space into a process. As the name implies you cannot directly access all that memory in one go but need to create 'windows' onto sections of it as and when you need to. The number and size of windows you can have mapped at any one time is still effectively bound by the 4GB per-process limit.

Due to the way the AWE work there are some restrictions on the memory that is allocated:

- The memory is non-paged.
- The application must be granted the 'Lock Pages in Memory Privilege'.

The API functions allow you to allocate memory as raw pages (as indicated by the use of the term Page Frame Numbers) – this is the same structure the kernel itself uses. You then request for a subset of those pages to be you need to **warn your System Administrators** about the **massive rise in page faults** that they will see in the process stats

```
const size_t size = 1024 * 1024;

HANDLE mapping =
CreateFileMapping(INVALID_HANDLE_VALUE,
                  NULL, PAGE_READWRITE,
                  0u, size, NULL);

if (mapping == NULL)
  throw std::runtime_error
     ("Failed to create segment");
```

Listing 1

mapped into a region of the process's restricted virtual address space to gain access to it, using the previously returned Page Frame Numbers.

For services such as SQL Server and Exchange Server, which are often given an entire host, this API allows them to make the most optimal use of the available resources on the proviso that the memory will never be paged out.

Page-file backed shared memory

There is another way to access all that extra memory using the existing Windows APIs in a manner similar to the AWE mechanism, but without many of its limitations: Shared Memory. Apart from not needing any extra privileges the memory allocated can also be paged which is useful for overcoming transient spikes or exploiting the paging algorithm already provided by the OS.

Allocating shared memory under Windows is the job of the same API used for Memory Mapped Files. In essence what you are mapping is a portion of a file, though not an application defined file but a part of the system's page-file. This is achieved by passing `INVALID_HANDLE_VALUE` instead of a real file handle to `CreateFileMapping()`. Listing 1 creates a shared segment of 1MB.

At this point we have allocated a chunk of memory from the system, but we can't access it. More importantly though we haven't consumed any of our address space either. To read and write to it we need to map a portion (or all) of it into our address space, which we do with `MapViewOfFile()`. When we're done we can free up the address space again with `UnmapViewOfFile()`. Continuing our example we require the code in Listing 2 to access the shared segment.

Every time we need to access the segment we just map a view, access it and un-map the view again. When we're completely done with it, we can free up the system's memory with the usual call to `CloseHandle()`.

Limitations of shared memory segments

This approach is not without its own constraints, as anyone who has used `VirtualAlloc()` will know. Just as with any normal heap allocation the actual size will be rounded up to some extent to match the underlying page size. What is more restrictive though is that the 'window' you map to

```
const size_t offset = 0;
const size_t length =  1024 * 1024;

void* region = MapViewOfFile(mapping,
   FILE_MAP_ALL_ACCESS, 0u, 0u, length);

if (region == NULL)
  throw std::runtime_error
     ("Failed to map segment");

// read & write to the region...

UnmapViewOfFile(region);
```

Listing 2

access the segment (via `MapViewOfFile`) must start on an offset which is a multiple of the 'allocation granularity'. This is commonly 64K and can be obtained by calling `GetSystemInfo()`. The length can be any size and will be rounded up to the nearest page boundary. This pretty much guarantees it's only useful with larger chunks of data.

A more subtle problem can arise if you fail to match the calls to `MapViewOfFile` with those to `UnmapViewOfFile`. Each call to `MapViewOfFile` bumps the reference count on the underlying segment handle and so calling `CloseHandle` will not free the segment if any views are still mapped. If left unchecked, this could create one almighty memory leak that would be interesting to track down.

Apart from the API limitations there is also the problem of not being able to cache or store raw pointers to the data either outside or inside the memory block – you must use or store offsets instead. The base address of each view is only valid for as long as the view is mapped so care needs to be taken to avoid dangling pointers.

One other operational side-effect of this technique that you need to warn your System Administrators about is the massive rise in page faults that they will see in the process stats. What they need to understand is that these are probably just 'soft faults' where a physical page is mapped into a process and not a 'hard fault' where a disk access also occurs. Although the segment is officially backed by the system page-file if enough physical RAM exists the page should never be written out to disk and so provides excellent performance.

Real-world use

I have previously used shared memory segments very successfully in two 32-bit COM heavy services that ran alongside other services on a 64-bit Windows 2003 server. One of them cached up to 16 GB of data without any undue side effects, even when transient loads pushed it over the physical RAM limit and some paging occurred for short periods.

I'm currently working on a .Net based system that is dependent on a 32-bit native library and have earmarked the technique again as one method of overcoming out-of-memory problems caused by needing to temporarily cache large intermediate blobs of data.

Research project – service-less caching

The ability to cache data in shared memory, which is effectively reference counted by the OS, provided the basis for a prototype mechanism that would allow multiple 'engine' processes running on the same host to cache common data without needing a separate service process to act as a gateway. This would avoid the massive duplication of cached data for each process, which, as the number of CPUs (and therefore engine processes) increased, would afford more efficient use of the entire pool of system RAM.

The mechanism was fairly simple. Instead of each engine process storing its large blobs of common data in private memory, it would be stored in a shared segment (backed by a deterministic object name) and mapped on demand. The use of similarly (deterministically) named synchronisation objects ensures that only one engine needed to request the data from upstream and the existence of a locally cached blob could be detected easily too. This was done by exploiting the fact that creation of an object with the same name as another succeeds and returns the special error **`ERROR_ALREADY_EXISTS`**.

The idea was prototyped but never used in production as far as I know.

Summary

This article provided a number of techniques to illustrate how a 32-bit Windows process can access more memory that the 2GB default. These ranged from configuration tweaks involving the **`/3GB`** and **`/PAE`** flags through to the **`AWE`** and Shared Memory APIs. Along the way it helped explain some of the terminology and showed how to help diagnose memory exhaustion problems. ■

Credits

Thanks to Matthew Wilson for reciprocating and commenting on my first draft, and to Frances Buontempo for her valuable feedback and encouragement too.

References

[Chen] Raymond Chen, 'Kernel address space consequences of the /3GB switch', http://blogs.msdn.com/b/oldnewthing/archive/2004/08/06/209840.aspx.

[MSDN] 'Memory Limits for Windows Releases', http://msdn.microsoft.com/en-gb/library/windows/desktop/aa366778(v=vs.85).aspx

[Russinovich] Mark Russinovich and David Solomon, *Windows Internals* 4th edition

The Signs of Trouble: On Patterns, Humbleness and Lisp

Patterns can be a controversial topic. Adam Petersen considers their cognitive and social value.

Our challenge of humbleness

In his classic talk at the Turing awards, Dijkstra remarked that computer programming is an "intellectual challenge which will be without precedent in the cultural history of mankind" [Dijkstra72]. What is it that makes software development so hard? Dijkstra himself gave the answer by concluding that a "competent programmer is fully aware of the strictly limited size of his own skull" [Dijkstra72]. It's a reference made to the great cognitive challenges of software development. Programming stretches our cognitive abilities to a maximum and we need to counter with effective design strategies to handle the complexity inherent in software.

One such strategy is to share knowledge and base our solutions on what has been known to work well in the past. Since few designs are really novel we often find that previous solutions, at least on a conceptual level, apply to our new problem too. Another strategy recognizes the capacity limits of our brain. Identifying ways to break those limits provide for more efficient usage of our precious grayware.

Patterns incorporate both of these strategies. As software developer and author of a technical book on patterns I obviously find value in the pattern format. And as a psychologist I see the links to our cognitive capabilities and the social value of patterns.

In this article I will detail my view on patterns and the value I see in them. Since patterns are a controversial topic, I will build the article around the criticism against patterns. Let the critics have the first word.

Misunderstood, misused or a sign of trouble?

A common view, expressed here by Jeff Atwood of Coding Horror fame, is that "design patterns are a form of complexity" [Atwood07]. As such, patterns should be avoided when simpler solutions would do.

Another view that tends to pop-up in discussions are patterns as workarounds for missing language features. I will ignore for a moment that patterns aren't limited to the programming task itself – patterns have been harvested in fields as diverse as testing, team organization, databases, etc – and keep the discussion on the subset of patterns related to design.

Proponents of this view refer to Peter Norvig's presentation on design patterns in dynamic programming languages [Norvig98]. In his presentation, Norvig classifies 16 of the 23 design patterns in the seminal *Design Patterns* book [Gamma94] as simpler or even invisible in higher-level languages. It's an interesting read and a true testimony to the power of the Lisp family of languages.

Paul Graham adds an interesting twist to the subject: "When I see patterns in my programs, I consider it a sign of trouble" [Graham04]. Graham continues to express that "a program should reflect only the problem it need to solve" [Graham04] which brings us to the very essence of design. Since I'm a Lisp programmer myself, I understand Graham's view. I also think he misses the most valuable part of what patterns really are.

Adam Petersen Combining degrees in engineering and psychology, Adam tries to unite these two worlds by making his technical solutions fit the human element. While he gets paid to code in C++, C#, Java and Python, he's more likely to hack Lisp or Erlang in his spare time. Other interests include modern history, music and martial arts.

The complexity of simplicity

Let me start by addressing Atwood's claim since it's more superficial than Graham's and provides a better starting point. I will work my way back through the mists of misconceptions and valid criticisms of patterns to finally take on Graham's critique from the view of a fellow Lisp programmer. But first we need to consider a more fundamental aspect: the use and possible abuse of patterns.

Atwood has a point; patterns are indeed used in situations where they don't apply. Sometimes that's a good sign. At least in a supportive environment. A willingness to try something new is a sign of intrinsic motivation. It's an attempt to improve. We need to make those errors and learn from them. The challenge is to provide an environment where the consequences are controlled. A mix of mentoring and peer reviews has proven to be a successful approach. Direct feedback is an important learning tool.

A worse problem than a motivated individual on a learning trail is over-engineering. Patterns aren't necessarily good. Used in the wrong context where the forces are unbalanced they make the resulting context worse. Patterns, like any other design choice, imply a trade-off. We buy flexibility in one area traded for some other consequences. Unless the flexibility is required we travel into the obscurity of speculative generality. A simpler solution would probably serve both the business and the brain of the maintenance programmer better. Or, as I put it in my book: "maintaining an AbstractFactorySingletonDecoratorBuilderPrototypeFlyweight isn't why I went into programming" [4]. Believe me, I've been there. There's just no supplement to good taste.

But the story is not as simple as Jeff Atwood implies. A pattern doesn't equal some automatic increase in complexity. I tend to use patterns as targets for refactorings. The reason I refactor is to get rid of accidental complexity and adapt the design to an increased understanding of both problem and solution space.

When used as targets of refactoring, patterns do shine. One of the reasons is that the resulting context is well documented. It's possible to reason about the changes up-front and contrast them to the existing implementation. A refactoring is an investment that we want to pay-off. Any good pattern description acknowledges the weak sides of the pattern, its trade-offs and hints at scenarios where the solution doesn't apply.

Knowing the common patterns in your domain gives you a powerful cognitive tool for large-scale refactorings.

Patterns as communication tools

Patterns have social value too. The format arose to enable collaborative construction using a shared vocabulary. In *Patterns in C* I write on the groundbreaking work of architect Christopher Alexander:

> The patterns found in Alexander's books are simple and elegant formulations on how to solve recurring problems in different

> contexts. [...] His work is a praise of collaborative construction guided by a shared language – a pattern language. To Alexander, such a language is a generative, non-mechanical construct. It's a language with the power to evolve and grow. As such, patterns are more of a communication tool than technical solutions. [Petersen12]

Patterns in the original sense are context-dependent and do not by some work of magic provide universally 'good' designs. You can't take the human out of the design loop. This is why I get disappointed every time I see someone offering a 'patterns code library' or expensive case tools that come stacked with 'UML pattern templates'. Reducing patterns to mechanisms doesn't tell the full story. The most interesting part of a pattern is rarely the implementation.

I understand the marketing perspective but those ideas completely miss the very essence and purpose of patterns. Sure, in software we are free from the physics that constrained Christopher Alexander. We may take a natural departure from Alexander's idea and provide general parameterizable implementations of specific patterns. If we do our job right that abstraction may well be usable, yes, sometimes even re-usable. But the ROI is diminishing fast.

Jeff Atwood falls into this trap as he labels the whole idea "a complex recipe of design patterns" [Atwood07]. Design patterns are not a 'recipe' either and were never intended as such. But Jeff's view is an understandable outcome of skimming through *Design Patterns* by the Gang of Four [Gamma06]. One of the main problems I have with the Gang of Four book is its prominent use of class diagrams. Open any pattern in the book. First thing we see is a class diagram.

I addressed this in *Patterns in C*:

> This has lead many developers to confuse the diagram with the actual pattern. It's not. At best, it's one possible way to implement the pattern in a certain language. Nothing more. A pattern is a dynamic, generative entity. Depending on context, the applications of a pattern may look radically different each time. It's my firm belief that much of the harm done to the design patterns movement could have been avoided had the Gang of Four just excluded the section on structure. [Petersen12]

Patterns for cognition

Patterns are primarily about communication. But the value of a shared vocabulary goes beyond communication. Patterns are powerful reasoning tools. Instead of reasoning about individual design elements and coding constructs, patterns provide a way to group these concepts into a larger unit. This has implications on our problem solving abilities.

In the introduction I quoted Dijkstra famously referring to the limits of our skulls. One of the main factors behind the limitation is working memory. Working memory is understood as the system that allows us to hold information in our mind, integrate different parts, reason about them and manipulate them. Working memory is what we use when we try to decipher a macro in Lisp, understand the relationship between two Java objects, or find a way to express a certain domain rule in Haskell.

Working memory is vital to our reasoning, problem solving, and decision making. It's also strictly limited in its capacity. Back in 1956, George Miller made the first quantification of our working memory capacity. Miller arrived at the now well-known heuristic of seven items, plus minus two. Sub-sequent research has refined Miller's number and distinguishes between verbal and visual information. The latter is even more limited with a mere four simultaneous items.

Given the few items we can hold in working memory simultaneously, it's no wonder that programming is hard; any interesting programming problem has a multitude of fine-grained parameters and possible alternatives. One way around this limitation is a process known as chunking. Chunking is an encoding strategy where individual elements are grouped into higher-level groups, chunks. While the limit on the number of units still apply, each unit now holds more information.

Patterns are a sophisticated form of chunking. Their names serve as handles to the vast knowledge stored in our long-term memory. Reading the name of a known pattern activates the associated network of knowledge and brings the ideas to conscious attention with economic usage of our working memory.

Program close to the domain

Given all the benefits of patterns, why would Paul Graham consider them "a sign of trouble" [Graham04]? Well, Graham doesn't actually discuss patterns. He mentions them in a more general discussion of the relative power of different programming languages: "regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough" [Graham04].

So Graham discusses patterns in the repetitive semantic sense of the word. And I completely agree. Repetitive patterns as in subtly duplicated code that cannot be refactored away is a sure sign that the language lacks expressive power. I often see that in Java (consider its dysfunctional **`try-catch-finally`** pattern in versions prior to Java 7).

Patterns as a medium for sharing knowledge, ideas and used as reasoning tools are a different story though. It has nothing to do with repetitive code. The two cases are orthogonal.

In this sense, the interesting thing is not patterns within a program; it's the conceptual patterns between programs. Such patterns are a sign of knowledge, learning and propagation of experience.

The patterns found in different languages will often differ. Design patterns are a result of the transition from problem to solution domain. This is never a perfect fit. If we can evolve and grow our language, like we do in Lisp, we get closer to the problem domain. Using Lisp as building material for domain-specific languages is a powerful strategy. It gives a much smother transition now that the solution domain has been designed to express precisely the specific problem at hand. At least on the surface. Our domain-specific language, however high-level it may be, will have to bridge the gap for us. The deeper we dive into the layers of the domain-specific language, the further away we get from the problem domain. It is here that documented patterns help. In case of Lisp, the patterns would focus on language creation, capture common approaches and document the trade-offs.

Level of scale

Patterns don't have to be large and complex. They exist at all level of scales. High-level languages grow their idioms too. One challenge of learning a new language is to get accustomed with its idiomatic ways of problem solving. By documenting the idioms we can provide guidance and ease the learning curve for newcomers to our technologies. Experts may benefit from the nuances and documented consequences present in all well-written patterns. This is what I tried to explore in *Patterns in C* [Petersen12].

Finally, it's worth pointing out that the GoF book [Gamma94] was a starting point, not the final word on patterns. The book is almost two decades old and in desperate need of a second edition. It's deliberately focused on a limited sub-set of the design space available to programmers (single-dispatch object-oriented design). Harvesting patterns in different paradigms would certainly result in a different catalogue with different implementation mechanisms. This journey is well worth embarking on. ■

References

[Atwood07] Jeff Atwood (2007). Rethinking Design Patterns

[Dijkstra72] Dijkstra (1972). The Humble Programmer

[Gamma94] Gamma et al (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*

[Graham04] Paul Graham (2004). Hackers and Painters

[Norvig98] Peter Norvig (1998). Design Patterns in Dynamic Languages.

[Petersen12] Adam Petersen (2012). *Patterns in C: Patterns, Idioms and Design Principles.*

The Open–Closed Principle (OCP)

Changing requirements and environments can require cascading changes through software. Nan Wang demonstrates how the Open-Closed principle can minimise changes

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. [APPP]

When the requirements of an application change, if the application confirms to OCP, we can extend the existing modules with new behaviours to satisfy the changes (Open for extension). Extending the behaviour of the existing modules does not result in changes to the source code of the existing modules (Closed for modification). Other modules that depend on the extended modules are not affected by the extension. Therefore we don't need to recompile and retest them after the change. The scope of the change is localised and much easier to implement.

The key of OCP is to place useful abstractions (abstract classes/interfaces) in the code for future extensions. However, it is not always obvious which abstractions are necessary. It can lead to over complicated software if we add abstractions blindly. I found Robert C Martin's 'Fool me once' attitude very useful [APPP]. I start my code with a minimal number of abstractions. When a change of requirements takes place, I modify the code to add an abstraction and protect myself from future changes of a similar kind.

I recently implemented a simple module that sends messages and made a series of changes to it afterward. I feel it is a good example of OCP to share.

At the beginning, I created a `MessageSender` that is responsible for converting an object message to a byte array and send it through a transport.

```
package com.thinkinginobjects;
public class MessageSender {
  private Transport transport;
  public synchronized void send(Message message)
     throws IOException{
    byte[] bytes = message.toBytes();
    transport.sendBytes(bytes);
  }
}
```

After the code was deployed to production, we found out that we sent messages too fast for the transport to handle. However, the transport was optimised for handling large messages, so I modified the `MessageSender` to send messages in batches of size of ten (Listing 1).

```
package com.thinkinginobjects;
public class MessageSenderWithBatch {
  private static final int BATCH_SIZE = 10;
  private Transport transport;
  private List buffer = new ArrayList();
  private ByteArrayOutputStream byteStream
     = new ByteArrayOutputStream();

  public
    MessageSenderWithBatch(Transport transport) {
      this.transport = transport;
    }

  public synchronized void
    send(Message message) throws IOException {
    buffer.add(message);
    if (buffer.size() == BATCH_SIZE) {
      sendBuffer();
    }
  }

  private void sendBuffer() throws IOException {
    for (Message each : buffer) {
      byte[] bytes = each.toBytes();
      byteStream.write(bytes);
    }
    byteStream.flush();
    transport.sendBytes(byteStream.toByteArray());
    byteStream.reset();
  }
}
```

Listing 1

The solution was simple but I hesitated to commit to it. There were two reasons:

1. The `MessageSender` class needs to be modified if we change how messages are batched in the future. It violated the Open-Closed Principle.
2. The `MessageSender` had a secondary responsibility to batch messages in addition to the responsibility of converting/delegating messages. It violated the Single Responsibility Principle.

Therefore I created a `BatchingStrategy` abstraction, who was solely responsible for deciding how message are batched together. It can be extended by different implementations if the batch strategy changes in the future. In a word, the module was open for extensions of different batch strategy. The `MessageSender` kept its single responsibility of converting/delegating messages, which means it does not get modified if similar changes happen in the future. The module was closed for modification (see Listing 2).

The patch was successful, but two weeks later we figured out that we can batch the messages together in time slices and overwrite outdated messages with newer versions in the same time slice. The solution was specific to our business domain of publishing market data.

More importantly, the OCP showed its benefits when we implemented the change. We only needed to extend the existing `BatchStrategy` interface

Nan Wang works as a software engineer in the financial sector. He has designed and built various systems, from ultra low latency exchange platforms to algorithmic trading and optimisation frameworks. He is enthusiastic about object oriented design and domain modelling. His blog is at http://www.ThinkingInObjects.com and he can be contacted at nwang0@gmail.com

The key of OCP is to **place useful abstractions** (abstract classes/interfaces) in the code **for future extensions**

```
package com.thinkinginobjects;
public class MessageSenderWithStrategy {
  private Transport transport;
  private BatchStrategy strategy;
  private ByteArrayOutputStream byteStream
     = new ByteArrayOutputStream();
  public synchronized void send(Message message)
     throws IOException {
    strategy.newMessage(message);
    List buffered = strategy.getMessagesToSend();
    sendBuffer(buffered);
    strategy.sent();
  }
  private void sendBuffer(List buffer)
     throws IOException {
    for (Message each : buffer) {
      byte[] bytes = each.toBytes();
      byteStream.write(bytes);
    }
    byteStream.flush();
    transport.sendBytes(byteStream.toByteArray());
    byteStream.reset();
  }
}
package com.thinkinginobjects;
public class FixSizeBatchStrategy
   implements BatchStrategy {
  private static final int BATCH_SIZE = 0;
  private List buffer = new ArrayList();
  @Override
    public void newMessage(Message message) {
      buffer.add(message);
    }
  @Override
    public List getMessagesToSend() {
      if (buffer.size() == BATCH_SIZE) {
        return buffer;
      } else {
        return Collections.emptyList();
      }
    }
  @Override
    public void sent() {
      buffer.clear();
    }
}
```

Listing 2

with an different implementation. We didn't change a single line of code but just the spring configuration file. (Listing 3)

* For the sake of simplicity, I have left the message coalescing logic out of the example.

```
package com.thinkinginobjects;
public class FixIntervalBatchStrategy
   implements BatchStrategy {
  private static final long INTERVAL = 5000;
  private List buffer = new ArrayList();
  private volatile boolean readyToSend;
  public FixIntervalBatchStrategy() {
    ScheduledExecutorService executorService
       = Executors.newScheduledThreadPool(1);
    executorService.scheduleAtFixedRate
       (new Runnable() {
      @Override
      public void run() {
        readyToSend = true;
      }
    }, 0, INTERVAL, TimeUnit.MILLISECONDS);
  }
  @Override
  public void newMessage(Message message) {
    buffer.add(message);
  }
  @Override
  public List getMessagesToSend() {
    if (readyToSend) {
      List toBeSent = buffer;
      buffer = new ArrayList();
      return toBeSent;
    } else {
      return Collections.emptyList();
    }
  }
  @Override
  public void sent() {
    readyToSend = false;
    buffer.clear();
  }
}
```

Listing 3

Conclusion

The Open-Closed Principle serves as an useful guidance for writing a good quality module that is easy to change and maintain. We need to be careful not to create too many abstractions prematurely. It is worth deferring the creation of abstractions to the time when the change of requirement happens. However, when the changes strike, don't hesitate to create an abstraction and make the module to confirm OCP. There is a great chance that a similar change of the same kind is at your door step. ■

References:

[APPP] *Agile Software Development, Principles, Patterns, and Practices,* Robert C Martin

Secrets of Testing WCF Services

WCF services provide middleware for applications but can be hard to test. Steve Love describes a way to develop a testable app.

The Windows Communication Foundation, or WCF, is part of a loosely related family of frameworks from Microsoft for developing robust and reliable systems. I say loosely because WCF and the other libraries can be used independently or in concert. The other main frameworks are WPF for presentation, and WF for workflow. WCF is for distributed and inter-process communications. Specifically, it is the middleware that applications can use to talk to each other, whether they are on the same machine, distributed over a LAN or even on the Internet.

There is a wealth of information about WCF on the MSDN [WCF, MSDN], and in a whole range of books [Resnick08] which presents the details of writing and configuring services and client applications. From reading this documentation, one would be forgiven for thinking that the 'W' in WCF was 'Web' rather than 'Windows' because much of it covers setting up a web-service, and using SOAP to establish communications and discover services over HTTP. WCF is, however, the official replacement for .Net Remoting Services, which itself superceded Distributed COM. It's my experience that WCF is actually really quite good in this application space, even if the documentation (official and otherwise) wants you, the developer, to think about web services and SOAP instead of RPC.

In this article, I want to explore some of those almost-hidden secrets of configuring and running WCF service- and client-applications. By going off the beaten path blazed by the example code that is available on MSDN, I hope to demonstrate how the resulting code can be more flexible and less intrusive in your applications.

The full source code for this is available on github [Love]. It's a Microsoft Visual Studio 2010 solution in C#, but should work in VS2012. A wiki page on the github site explains the different projects in the code.

A standing start

There are two main starting points from where discussion of using WCF for applications might commence. The first is to begin with the premise that the application will be distributed, that WCF has been chosen as the facilitator for that, and so we would start with defining some service contracts and definitions. The second scenario – which is probably much more common – is to begin with the premise that there is some application already in existence, for which WCF has been chosen as the technology to turn it into a distributed application. Most of the MSDN introductions, and many of the books on WCF follow the first scenario, but I'm now going to introduce a third possibility.

I want to start with a new application that doesn't use WCF in any way, and turn it into a distributed application.[1] The main reason for this is about testing. Distributed applications that use technologies like WCF are notoriously hard to test in an automated fashion. Testing the client usually has to assume that a corresponding server is available, and the client must be configured correctly to use it. Testing the server is notionally simpler if you separate the communications code from the 'server' code (as it were) internally, so you can at least automate testing the code that provides the logic for your service, but it's all rather messy. Which is probably why discussion of this kind is conspicuously absent from MSDN examples and books about WCF.

The example code for this article is a recipe book application for cocktails. It isn't terribly sophisticated, and the outline of the design ought to be clear from Listing 1.

```
public interface RecipeBook : IDisposable
{
  IEnumerable< Drink > AllDrinks { get; }
  IEnumerable< string > AllIngredients { get; }
  void Add( params Drink [] newDrinks );
  IEnumerable< Drink > WithIngredients
     ( params string [] selected );
}

public interface Drink : IEquatable< Drink >
{
  string Name { get; }
  string Method { get; }
  IEnumerable< Ingredient > Ingredients { get; }
}

public enum Measurement
{
  Fill, Measure, Drop, Tsp,
}

public sealed class Ingredient
   : IEquatable< Ingredient >
{
  public Ingredient( string name,
                     Measurement mmt, int qty )
  {
    if( string.IsNullOrEmpty( name ) )
      throw new ArgumentNullException( "name" );
    Name = name;
    Amount = mmt;
    Qty = qty;
  }

  public string Name { get; private set; }
  public Measurement Amount { get; private set; }
  public int Qty { get; private set; }
}
```

Listing 1

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

1 Yes, it's cheating a bit, because it's similar to doing scenario 1 and then 2 in sequence.

one would be **forgiven for thinking** that the 'W' in WCF was **'Web' rather than 'Windows'** because much of it covers **setting up a web-service**

```
public class DrinksCabinet
{
  public DrinksCabinet( RecipeBook recipes )
  {
    this.recipes = recipes;
  }
  public Drink Find( string name )
  {
    return recipes.AllDrinks.Single
       ( d => d.Name == name );
  }
  public IEnumerable< string > Ingredients
  {
    get { return recipes.AllIngredients; }
  }
  public IEnumerable< Drink > NotContaining
      ( params string [] selected )
  {
    var remain = Ingredients.Except( selected );
    return recipes.WithIngredients
       ( remain.ToArray() );
  }
  private readonly RecipeBook recipes;
}
```

Listing 2

One thing of note here, for the sake of brevity later, is that the `Drink` type is an interface instead of a value-type like `Ingredient`. Consider it a foresight of things to come in the design when we start considering distributed communications.

In any case, there is nothing here, or in the implementing types, to do with WCF or any kind of remoting. To make the example more interesting, let's introduce a new class that uses the recipe book, and adds some value. Listing 2 uses the `RecipeBook` interface, and provides some simple queries over those in `RecipeBook`. Lastly listing 3 has some unit tests for the `DrinksCabinet` queries. In this test class you can see that the concrete `LocalDrink` type is used; this is a value-like implementation of the `Drink` interface mentioned previously. The `DrinksCabinet` constructor takes a `RecipeBook` reference, and the test creates a `LocalRecipeBook` which implements that interface using simple collection types to store the data.

Clearly, we could use any implementation of `RecipeBook` to provide to the `DrinksCabinet` object, which brings us neatly to the next section.

Pace yourself

Would it be possible to use the `RecipeBook` interface as the contract for a remote service? If it is, then the service can implement it in some way, and the client can use a proxy implementation to communicate. Well, we can't use the interface directly, but it would certainly be possible to modify it in simple ways to operate as a WCF contract. However, let's step back

```
[ TestFixture ]
public class DrinksCabinetTests
{
  private RecipeBook recipes;
  [SetUp]
  public void Start()
  {
    recipes = new LocalRecipeBook();
  }
  [TearDown]
  public void End()
  {
    recipes.Dispose();
  }
  [Test]
  public void EmptyCabinetHasNoIngredients()
  {
    var cabinet = new DrinksCabinet( recipes );
    var results = cabinet.Ingredients;
    Assert.IsFalse( results.Any() );
  }
  [ Test ]
  public void CanLocateSpecificDrinkByName()
  {
    var cabinet = new DrinksCabinet( recipes );
    var expected = new LocalDrink( "a", "",
      new[] { new Ingredient( "1",
      Measurement.Tsp, 1 ) } );
    var error = new LocalDrink( "b", "", new[] {
      new Ingredient( "2", Measurement.Tsp, 1 )
    } );
    recipes.Add( expected, error );
    var result = cabinet.Find( expected.Name );
    Assert.AreEqual( expected, result );
  }
  [ Test ]
  public void CanFilterDrinksOnNotSpecified()
  {
    var cabinet = new DrinksCabinet( recipes );
    var expected = new LocalDrink( "a", "",
       new[] {
      new Ingredient( "1", Measurement.Tsp, 1 )
      } );
    var error = new LocalDrink( "b", "", new[] {
      new Ingredient( "2", Measurement.Tsp, 1 )
    } );
    recipes.Add( expected, error );
    var results = cabinet.NotContaining( "2" );
    Assert.AreEqual( expected, results.Single() );
  }
}
```

Listing 3

WCF imposes some restrictions on the way you compose contracts. WCF requires those contracts to explicitly attribute the interfaces, methods and types used in the contract

```
[ ServiceContract ]
public interface RecipeBookContract
{
  [ OperationContract ]
  IEnumerable< DrinkDto > AllDrinks();

  [ OperationContract ]
  IEnumerable< string > AllIngredients();

  [ OperationContract ]
  IEnumerable< IngredientDto > IngredientsOf
     ( string drink );

  [ OperationContract ]
  void Add( DrinkDto drink,
    IEnumerable< IngredientDto > newIngredients );

  [ OperationContract ]
  IEnumerable< DrinkDto > WithIngredients
     ( IEnumerable< string > ingredients );
}
```

Listing 4

for a moment. WCF can be used to provide network communications, as well as inter-process, and should be less 'chatty' than a native interface. A direct port may not be at all appropriate if it would introduce the need for unnecessary communications between client and server. Adapting a native interface to reduce the communications requires an intermediate abstraction.

Additionally, WCF imposes some restrictions on the way you compose contracts. WCF requires those contracts to explicitly attribute the interfaces, methods and types used in the contract. Ideally, keeping these WCF specific warts out of the way of application code would be a Good Idea.

Listing 4 shows what a WCF contract version of the **RecipeBook** interface might look like. It's not a direct port from **RecipeBook**, partly because I want to demonstrate adapting one interface to the other, but there are other differences that are due to it being part of the WCF subsystem. Also note the use of names such as **IngredientDto** (**Dto** indicates it's a DataTransferObject, see [Fowler, Fowler02]) instead of just **Ingredient**. We could easily use the same names for the component types as in the original **RecipeBook** interface that we're adapting, and namespaces would take care of clashes. However, the adapting code would necessarily contain lots of explicit namespace qualification to distinguish between the native and DTO versions, and it can get difficult to keep track of which side of the WCF boundary you are on.

The **AllIngredients** and **AllDrinks** properties from **RecipeBook** are methods in the contract. A **ServiceContract** is essentially a collection of **OperationContracts**, which are required to be methods (as opposed to properties). Another restriction is on the use of **params**

```
[ DataContract ]
public class DrinkDto
{
  [ DataMember ]
  public string Name { get; set; }

  [ DataMember ]
  public string Method { get; set; }
}

[ DataContract ]
public class IngredientDto
{
  public enum Measurement { Fill, Measure,
     Drop, Tsp, }

  [ DataMember ]
  public string Name { get; set; }

  [ DataMember ]
  public Measurement Amount { get; set; }

  [ DataMember ]
  public int Qty { get; set; }
}
```

Listing 5

arrays, so these have become **IEnumerable** objects instead, which is only a minor inconvenience really in any case.

Listing 5 shows the participating classes. Note that **DrinksDto** is *not* an interface here, because it's not a **ServiceContract**, it's a **DataContract**. One other difference to note is that a **DrinksDto** doesn't directly expose its ingredients. The facility to associate ingredients with drinks has moved up to the contract interface, in order to keep all the **OperationContract** methods in one place. It also permits me to illustrate some interface adapting between the contract and the native interfaces [2].

Planning a route

Adapting between the **RecipeBookContract** and **RecipeBook** interfaces is quite straightforward, but before getting to that, I want to briefly show the implementations of the WCF service and corresponding client applications, to illustrate some of the challenges with separating out the needs of the application from the requirements of WCF.

There are three main components to any WCF service, known as the ABCs:

- A is the address
- B is the binding
- C is the contract

2. It's not entirely specious; the new interface allows ingredients to be lazy-loaded on demand.

```
<system.serviceModel>
  <services>
    <service
       name="Shaker.Service.RecipeBookService">
      <endpoint binding="basicHttpBinding"
         contract=
         "Recipes.Contract.RecipeBookContract" />
    </service>
  </services>
</system.serviceModel>
```

Listing 6

```
class ShakerServiceProgram
{
  static int Main()
  {
    const string
       address = "http://localhost:5110";
    var recipes = new RecipeBookService();
    using( var host = new ServiceHost( recipes,
       new Uri( address ) ) )
    {
      host.Open();
      Console.WriteLine
         ( "Press [Enter] to close" );
      Console.ReadLine();
    }
    return 0;
  }
}
```

Listing 7

We have already dealt with C, and the most flexible way of associating the contract with the binding is through a configuration file, such as that in listing 6.

The service name in the `system.serviceModel` element is the fully-qualified type name of the implementing type of the contract (shown later in listing 8). The contract attribute of the endpoint element is the fully-qualified type name of the contract interface. Lastly for the configuration, this example uses the `basicHttpBinding` type which defines the communication method being exposed by the server.

One of the features of WCF is the ability to discover a service interface. The tool for this is `svcutil.exe`, which is also exposed inside Visual Studio as 'Add Service Reference' in the project tree. This tool is there to allow you to reference not only WCF services, but SOAP services on other platforms. Likewise, with certain restrictions, WCF services can be exposed to other platforms to consume. This is achieved by setting up a meta-data exchange (mex) endpoint in the service configuration, which uses SOAP standards to 'explain' to the client the details of the contract. svcutil generates the client-side configuration and proxy code to handle the communication layer.

However, this is much more than this simple application needs, and where all the communicating parts of an application are in .Net, a shared library referenced by both client and server with the interface for the contract is sufficient, and that's what the example here uses. Besides which, I have a phobia about generated code.

You can also add the base address for the server in the same configuration, but for reasons of brevity – and because it'll become useful later – the address for this server is directly in the server application code. This is a simple console application, shown in listing 7.

The heart of the application is the `ServiceHost` object, which uses the application configuration to determine the binding (and any other attributes which are defined in the configuration file), and to host the service. The default behaviour of a WCF service host is to create a single-threaded instance of the contract implementation (`RecipeBookService`, listing 8) for each call to the service. Other hosting options are available, including single-instance and per-session. Since this service is just using local collections to manage the objects to be stored and retrieved, it's important that successive calls to the service communicate with the same instance. To achieve this, two things are needed. First, the implementing class is attributed with the `Single` instance mode, and secondly an actual instance of the type is passed to the `ServiceHost` object, instead of a type for it to instantiate as it sees fit.

```
[ ServiceBehavior( InstanceContextMode
   = InstanceContextMode.Single ) ]
public class RecipeBookService
   : RecipeBookContract
{
  public RecipeBookService()
  {
    drinks = new List< DrinkDto >();
    ingredients = new Dictionary< string,
       List< IngredientDto > >();
  }
  public IEnumerable< DrinkDto > AllDrinks()
  {
    return drinks;
  }
  public IEnumerable< string > AllIngredients()
  {
    return drinks.SelectMany( d => IngredientsOf
       ( d.Name ) )
      .Select( i => i.Name ).Distinct();
  }
  public IEnumerable< IngredientDto >
     IngredientsOf( string drink )
  {
    return ingredients[ drink ];
  }
  public void Add( DrinkDto drink,
     IEnumerable< IngredientDto > i )
  {
    if( drinks.Any( d => d.Name == drink.Name ) )
    {
      throw new FaultException
         ( "DuplicateDrink" );
    }
    drinks.Add( drink );
    ingredients.Add( drink.Name, i.ToList() );
  }
  public IEnumerable< DrinkDto > WithIngredients
     ( IEnumerable< string > selectedIngredients )
  {
    var result = drinks.Where
       ( drink => ingredients[ drink.Name ]
       .Any( dto => selectedIngredients.Contains
       ( dto.Name ) ) );
    return result;
  }
  private readonly List< DrinkDto > drinks;
  private readonly Dictionary< string,
     List< IngredientDto > > ingredients;
}
```

Listing 8

Finally, it's time to show the contract implementation. Listing 8 shows an example implementation of the `RecipeBookContract` service contract interface in listing 4. Note that the `IngredientDto` and `DrinkDto` types are already full classes – not interfaces – and can so be used directly.

The client application uses the same interface and data contract types to communicate with the server. A configuration is still required to set up the client-side WCF parts. The client configuration that corresponds to the server configuration in listing 6 is shown in listing 9.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="ShakerService"
        binding="basicHttpBinding"
        contract=
          "Recipes.Contract.RecipeBookContract" />
    </client>
  </system.serviceModel>
</configuration>
```

Listing 9

The differences are subtle; instead of a services collection, there is a single client section, and the endpoint definition has a name. This name is used in the client program to identify the endpoint to which to connect.

Lastly, of course, we need code in the client application to set up the channel, talk to the service and consume the results, such as that in listing listing 10.

This code sets up a connection to the endpoint on which the server is listening using the name of the client-side endpoint definition in the configuration file, and uses the service contract. The **`ChannelFactory`** is parameterised with the contract interface type, associates it with the endpoint address specified, and uses the application config shown in listing 9 to configure the channel.

Client and server are separate programs, using WCF to communicate with each other. In this instance, they are console applications (for the server, the correct terminology is 'self-hosting'), but could be Windows Forms, WPF or (for the server) an IIS-hosted web app.

```
static class ShakerClientProgram
{
  static int Main()
  {
    const string name = "ShakerService";
    const string address
       = "http://localhost:5110";

    using( var channel = new ChannelFactory
       < RecipeBookContract >( name ) )
    {
      var recipes = channel.CreateChannel
         ( new EndpointAddress( address ) );
      recipes.Add(
        new DrinkDto{ Name = "G&T",
           Method = "Mix with ice and lime" },
        new[] {
          new IngredientDto{ Name = "Gin",
          Amount =
          IngredientDto.Measurement.Measure,
          Qty = 2 },
          new IngredientDto{ Name = "Tonic Water",
          Amount = IngredientDto.Measurement.Fill,
          Qty = 1 }
      } );
      foreach( var d in recipes.AllDrinks() )
      {
        Console.WriteLine( d.Name );
      }
      Console.WriteLine("Press [Enter] to exit");
      Console.ReadLine();
      return 0;
    }
  }
}
```

Listing 10

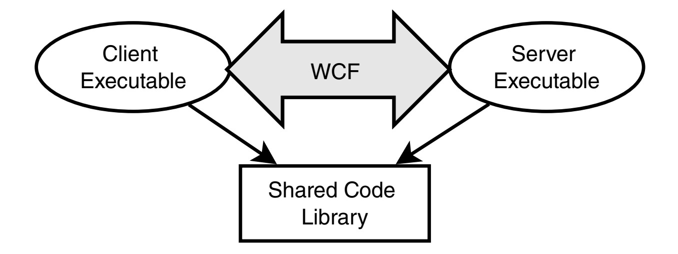


Figure 1

Since they share the service contract definition in the **`RecipeBookContract`** interface, it makes sense to create a shared library which both console applications can reference. It would be possible to have the contract definition as a class in the server application, and have the client application reference the server assembly directly, but there are many reasons why this would be a bad idea; having one executable directly reference another is rarely a sign of good design.

Figure 1 shows the basic architecture of this application.

Back on track

Clearly, having the client side of the WCF application running as a process is less than ideal with respect to using it from other parts of your codebase. It would be better to make the client operate from a library. The problem with that is the channel configuration. The **`ChannelFactory`** shown in listing 10 is hard-wired to use the application configuration file. This is a convenience for the supposed default case (where the client is the applicaton), but is here exposed as a liability. The superficial answer to that is to put all the WCF configuration into the app.config file of whatever application is hosting the client, and this works. However, it pollutes the application's configuration, and means that the client *must* be hosted in an executable.

An alternative to that would be to remove all configuration from the client-side of the WCF channel. You can create a bare-bones channel, and set properties on it within the code, thus removing the need for a configuration file. This solution has some attractions, of course, but sometimes being able to fiddle with the settings without having to recompile the code is useful, perhaps even necessary.

Fortunately, Microsoft provide a solution. The **`ConfigurationChannelFactory`** is very similar to

```
public class RecipeBookClient : Recipes.RecipeBook
{
  public RecipeBookClient( string address )
  {
    const string name = "ShakerService";
    var cfgMap = new ExeConfigurationFileMap
      { ExeConfigFilename
        = "RecipeBook.Client.Config" };
    var config = ConfigurationManager
         .OpenMappedExeConfiguration
       ( cfgMap, ConfigurationUserLevel.None );
    channel = new ConfigurationChannelFactory
       < RecipeBookContract >
      ( name, config,
        new EndpointAddress( address ) );
    recipes = channel.CreateChannel();
  }
  private readonly ConfigurationChannelFactory
     < RecipeBookContract > channel;
  private readonly RecipeBookContract recipes;
  }
}
```

Listing 11

```
public IEnumerable< Drink > AllDrinks
{
  get { return Call( () => recipes.AllDrinks()
    .Select( d => new DrinkAdapter
       ( recipes, d ) ) ); }
}

public IEnumerable< string > AllIngredients
{
  get { return Call( () =>
    recipes.AllIngredients() ); }
}

public void Add( params Drink[] newDrinks )
{
  foreach( var drink in newDrinks )
  {
    Call( () =>
      recipes.Add( new DrinkDto {
        Name = drink.Name,
        Method = drink.Method },
      drink.Ingredients.Select
         ( i => new IngredientDto {
        Name = i.Name,
        Amount =
           ( IngredientDto.Measurement )i.Amount,
        Qty = i.Qty
      } ) ) );
  }
}

public IEnumerable< Drink > WithIngredients
   ( params string[] s )
{
  return Call( () => recipes.WithIngredients( s )
    .Select( d =>
      new DrinkAdapter( recipes, d ) ) );
}

public ResultType Call< ResultType >
   ( Func< ResultType > method )
{
  var result = default( ResultType );
  Call( () => { result = method(); } );
  return result;
}

public void Call( Action method )
{
  try
  {
    method();
  }
  catch( FaultException x )
  {
    switch( x.Message )
    {
      case "DuplicateDrink":
        throw new DuplicateDrinkException
           ( x.Message );
    }
    throw;
  }
}
```

Listing 12

```
public class DrinkAdapter : Drink
{
  public DrinkAdapter( RecipeBookContract recipes,
     DrinkDto drinkDto )
  {
    this.recipes = recipes;
    Name = drinkDto.Name;
    Method = drinkDto.Method;
  }

  public string Name { get; private set; }
  public string Method { get; private set; }
  public IEnumerable< Ingredient > Ingredients
  {
    get { return recipes.IngredientsOf( Name )
      .Select( i => new Ingredient(
        i.Name,
        ( Measurement )i.Amount,
        i.Qty ) ); }
  }

  private readonly RecipeBookContract recipes;
}
```

Listing 13

`ChannelFactory` except that it accepts a `System.Configuration` object in its constructor[3]. In conjunction with the almost impenetrable interfaces to the .Net application configuration system – with which details I shall not bore you – you can load any configuration file of the correct format at runtime. Armed with this, a library assembly can have its own configuration file, and load it internally to configure the WCF subsystem for an implementation of the client-facing interface.

3. It is not hard to argue that making this the default, or an overloaded constructor on ChannelFactory, might be better than a separate type.

The constructor shown in listing 11 shows how the configuration is loaded and associated with the channel. Setting the `ExeConfigFileName` property of the `filemap` object instructs the system to look for the file in the working directory. Other properties exist to tell it to look in various profile folders. The remainder of the class, in listing 12, adapts the WCF contract interface's methods and objects into the client interface. All calls to the WCF service are wrapped in a delegate call to a private method that captures the common error handling.

Of more interest at this point is the implementation of the `Drink` interface first shown in listing 1 used by the `RecipeBookClient` in listing 12 to adapt between the native and contract interfaces. The implementation is in listing 13. Precisely why this was an interface should now be clear. Being able to implement it independently to communicate with a WCF service has great benefits for efficiently and effectively exposing the `ServiceContract` interface and its collaborators. Figure 2 shows the new architecture.

It's now possible to use the client code from anywhere – a hosting executable, part of a library of shared code, or a test case in a DLL. It still depends on having a running server process, so such a test isn't really stand-alone, but it does allow us to provoke the code that uses the server. One of the benefits of this is that it is a test of the client configuration – a reasonably novel concept.

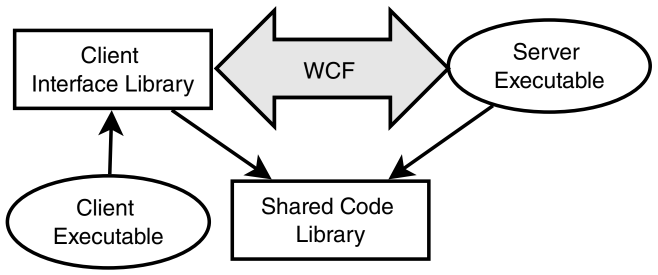


Figure 2

```
public class RecipeBookHost : ServiceHost
{
  public RecipeBookHost( string address )
  {
    var cfgMap = new ExeConfigurationFileMap
      { ExeConfigFilename =
        "RecipeBook.Server.config" };
    config = ConfigurationManager
      .OpenMappedExeConfiguration
      ( cfgMap, ConfigurationUserLevel.None );
    var service = new RecipeBookService();
      InitializeDescription( service,
      new UriSchemeKeyedCollection( new Uri
         ( address ) ) );
      Open();
  }

  protected override void ApplyConfiguration()
  {
    var section =
       ServiceModelSectionGroup.GetSectionGroup
       ( config );
    if( section == null )
      throw new ConfigurationErrorsException
        ( "Failed to find service model
          configuration" );
    foreach( ServiceElement service in
       section.Services.Services )
    {
      if( service.Name ==
         Description.ConfigurationName )
        base.LoadConfigurationSection
          ( service );
      else
        throw new ConfigurationErrorsException
          ( "No match for description in
          Service model config" );
    }
  }
  private readonly Configuration config;
}
```

Listing 14

Going off-road

The next step is to try and isolate the server code into a shared library. The benefits here are largely about testing, but having the service code in a shared library allows you to defer the decision on whether to host it in a console application, a Windows Service or some other solution.

Moving the implementation of the **`RecipeBookContract`** from listing 8 into a shared library is easy enough: it can be copied as it stands, since it doesn't use any WCF objects directly.

As with the **`ChannelFactory`** in the original client code (listing 10), the **`ServiceHost`** object in the server code (listing 7) is hard-wired to use the `app.config` for the WCF settings. There isn't a corresponding **`ConfigurationServiceHost`** class which can be used in place of it, but we can derive from it and override it's default behaviour. Listing 14 shows the new class.

The constructor sets up the configuration in much the same way as the **`RecipeBook-Client`** from listing 11.

The important method is the overridden **`ApplyConfiguration`**. This pulls the required WCF settings out of the configuration object and calls the base class's **`LoadConfigurationSection`** to apply those settings.

```
static class ShakerServiceProgram
{
  static int Main()
  {
    const string address =
       "http://localhost:5110";
    using( var host =
       new RecipeBookHost( address ) )
    {
        Console.WriteLine( "v1 Shaker Service
           running. Press [Enter] to close" );
        Console.ReadLine();
    }
    return 0;
  }
}
```

Listing 15

`ApplyConfiguration` is called from the **`InitializeDescription`** invocation in the constructor. If this hasn't been done before **`Open`** is called, then the `app.config` is automatically loaded, so the order of operations is crucial. This class can be used in a hosting application such as that shown in listing 15, or (for example) a test case.

Figure 3 shows the architecture of the client/server portion now.

```
[TestFixture, Explicit, Category("Service")]
public class ClientServiceTest
{
  [Test]
  public void
    ClientAndServiceStartAndCanCommunicate()
  {
    const string address
       = "http://localhost:5110";
    using( var host = new RecipeBookHost
       ( address ) )
    using( var recipes = new RecipeBookClient
       ( address ) )
    {
      Assert.IsNotNull( host );
      Assert.IsNotNull( recipes );
      Assert.DoesNotThrow( () => {
        var x = recipes.AllDrinks; } );
    }
  }
}
```

Listing 16

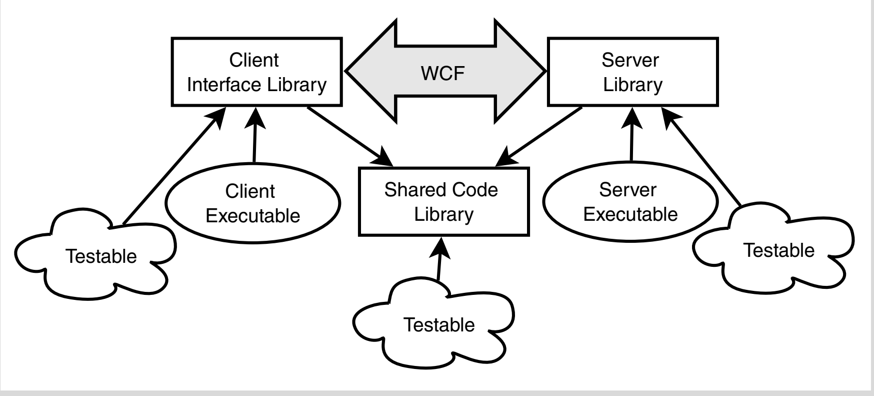


Figure 3

Tests

With the new server and client library code we've now developed, it's possible to write a truly automated test that has no requirements on external running code. Just as importantly, we can test them independently of any other code in an application (but not independently of each other - the client code *must* have a server to talk to). listing 16 shows a very simple test that creates an instance of the server, an instance of the client, and asserts that they can communicate. This is effectively testing the configurations of both client and server (an important thing, to be sure), but a much more interesting test would be to reproduce the tests we performed at the very beginning - by using the `DrinksCabinet` object.

Recall from listing 2 that `DrinksCabinet` accepts an instance of the `RecipeBook` interface. Our new client code implements that interface, and adapts it to the WCF contract. It's now possible to reproduce the `DrinksCabinetTests` from listing 3 but instead of using an instance of `LocalRecipeBook`, use an instance of `RecipeBook-Client`, as shown in listings 11 and 12.

Listing 17 shows the new tests using a locally running instance of the server, and the WCF implementation of the `RecipeBook` interface. Of course, the service implementation of the `RecipeBook` interface, and its client-side counterpart, can be tested in a similar way.

Conclusion

In this article, I've described a simple application, beginning from a basic native (and testable) interface and implementation, then showing how that interface and its collaborators could be transposed into a WCF service application. The flaws in the direct translation using a process to represent the server and client sides were that the implementation code wasn't shareable easily with other applications, and that it wasn't easily (and automatically) testable. The solution to both of those problems involved investigating WCF and by necessity, the .Net Configuration management systems to allow both server implementation and the client proxy adapter code to be exposed from their own shared assembly using their own configurations for the WCF subsystem. Being able to automatically test the WCF portions of your application should give you greater confidence that it works correctly – including the fact that the configuration is correct and sufficient – without having to run your application end-to-end to provoke it. Of course, that shouldn't stop you from performing end-to-end testing! ■

Acknowledgements

Many thanks to Frances Buontempo for reading and commenting on initial drafts of this, and to Roger Orr and Chris Oldwood for valuable feedback on it.

References and source

[Fowler] Martin Fowler. Data transfer object. Technical report, http://martinfowler.com/eaaCatalog/dataTransferObject.html.

[Fowler02] Martin Fowler. *Patterns of Enterprise Application Architecture*. AddisonWesley, 2002.

[Love] Source code at: https://github.com/essennell/WcfTestingSecrets

[MSDN] MSDN(WCF). Windows communication foundation reference. Technical report, Microsoft, http://msdn.microsoft.com/en-us/library/dd456779.aspx.

[Resnick08] Bowen Resnick, Crane. *Essential Windows Communication Foundation*. Microsoft .Net Development Series. Addison Wesley, 2008. .Net 3.5.

[WCF] Microsoft(WCF). Windows communication foundation msdn articles. Technical report, Microsoft, http://msdn.microsoft.com/en-us/library/dd560536.aspx.

```
[ TestFixture, Explicit, Category( "Service" ) ]
public class ServiceDrinksCabinetTests
{
  private RecipeBookHost host;
  private Recipes.RecipeBook recipes;
  [ SetUp ]
  public void Start()
  {
    const string address
       = "http://localhost:5110";
    host = new RecipeBookHost( address );
    recipes = new RecipeBookClient( address );
  }

  [ TearDown ]
  public void End()
  {
    recipes.Dispose();
    host.Close();
  }

  [Test]
  public void EmptyCabinetHasNoIngredients()
  {
    var cabinet = new DrinksCabinet(recipes);
    var results = cabinet.Ingredients;
    Assert.IsFalse(results.Any());
  }

  [Test]
  public void CanLocateSpecificDrinkByName()
  {
    var cabinet = new DrinksCabinet(recipes);
    var expected = new LocalDrink("a", "", new[] {
      new Ingredient( "1", Measurement.Tsp, 1 )
    });
    var error = new LocalDrink("b", "", new[] {
      new Ingredient( "2", Measurement.Tsp, 1 )
    });
    recipes.Add(expected, error);
    var result = cabinet.Find(expected.Name);
    Assert.AreEqual(expected, result);
  }

  [Test]
  public void CanFilterDrinksOnNotSpecified()
  {
    var cabinet = new DrinksCabinet(recipes);
    var expected = new LocalDrink("a", "", new[] {
      new Ingredient( "1", Measurement.Tsp, 1 )
    });
    var error = new LocalDrink("b", "", new[] {
      new Ingredient( "2", Measurement.Tsp, 1 )
    });
    recipes.Add(expected, error);
    var results = cabinet.NotContaining("2");
    Assert.AreEqual(expected, results.Single());
  }
}
```

Listing 17

Letter to the Editor

Dear Editor,

I really enjoyed Cassio Neri's article on 'Complex Logic in the Member Initialiser List' as it's a problem I've had to deal with several times. I was surprised to learn that Listing 6 was valid, because I didn't realise callers of that constructor didn't need access to the private 'storage' type to create the default argument at the call site. On reflection I realised the caller doesn't have to name the type and that access checking in C++ is done on names. I love an *Overload* article that makes me stop halfway through reading to reach for the compiler and learn something new!

Despite discussing how some C++11 features help solve the problems being discussed, I was surprised to notice the article didn't mention one of the new C++11 features that I find very useful for solving exactly those sort of problems. The 'delegating constructors' feature allows one constructor to invoke a different constructor, in order to avoid duplicating logic in constructor bodies. This feature is useful when one constructor (the delegating constructor) wants to process or munge its arguments somehow and then pass them on to another constructor (the target constructor.) Using delegating constructors allows Cassio's Listing 6 to be rewritten like so:

```
class bar : public base {
  struct storage {
    storage(double d) : b(d * d) { }
    double b;
  };
  ...
  bar(double d, foo& r1, foo& r2, storage tmp);

public:
  bar(double d, foo& r1, foo& r2);
};

bar::bar(double d, foo& r1, foo& r2)
: bar(d, r1, r2, storage(d))
{ }

bar::bar(double d, foo& r1, foo& r2, storage tmp)
: base(tmp.b),
  x_(cos(tmp.b)), y_(sin(tmp.b)), ...
{ }
```

In this version of the code the user-accessible constructor doesn't mention the private 'storage' type, which separates the interface meant for users from the implementation details of the complex constructor logic. Additionally, I believe it solves several of the problems mentioned in the article and makes the techniques shown in Listing 7 and Listing 8 unnecessary, and ultimately avoids the need to use a discriminated union for this scenario.

Because the temporary 'storage' object is initialized in the member initializer list, not the parameter list, there is no restriction on referring to the function parameters, so 'storage' can be constructed with 'd' and can have an arbitrarily complex constructor that can do any necessary calculations. Because the delegating constructor doesn't initialize any base classes (that's done by the target constructor) the 'storage' object is guaranteed to be initialized before the target constructor is invoked so it avoids any problems with order of initialization of base classes.

I think using delegating constructors for this problem is almost the ideal solution. The 'storage' constructor allows the complex logic to be placed in a separate function where it can be written more naturally (rather than in a contrived function such as `bar::init_base`) and that function is guaranteed to be executed at exactly the right time: after the user calls the (delegating) constructor but before the invocation of the target constructor that actually performs initialization of the base classes and members.

The only downside, which might be why they weren't considered in the original article, is that delegating constructors were not widely supported until quite recently. Clang has supported them since version 3.0, GCC since 4.7 and MSVC now supports them too as of the November 2012 CTP.

Yours,

Jonathan Wakely

Dear Jonathan,

Thank you for bringing this to our attention. Feel free to write in with any further comments. Cassio tells me Jeff Snyder (who can be found online as je4d) contacted him with similar remarks about delegating constructors.

Frances – Overload editor