Contents

Copy Dates

C Vu 18.1: January 1st 2006

C Vu 18.2: March 1st 2006

Contact Information:

Editorial: Paul Johnson
77 Station Road, Haydock,
St Helens,
Merseyside, WA11 0JL
cvu@accu.org

Advertising: Thaddaeus Frogley
ads@accu.org

Treasurer: Stewart Brodie
29 Campkin Road,
Cambridge, CB4 2NL
treasurer@accu.org

ACCU Chair: Ewan Milne
0117 942 7746
chair@accu.org

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

Membership Secretary: David Hodge
01424 219 807
membership@accu.org

Cover Art:	**Alan Lenton**
Repro:	**Parchment (Oxford) Ltd**
Print:	**Parchment (Oxford) Ltd**
Distribution:	**Able Types (Oxford) Ltd**

Membership fees and how to join:

Basic (C Vu only): £25

Full (C Vu and Overload): £35

Corporate: £120

Students: half normal rate

ISDF fee (optional) to support Standards work: £21

There are 6 issues of each journal produced every year.

Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.

Want to use cheque and post - email membership@accu.org for an application form.

Any questions - just email membership@accu.org

Reports & Opinions

Editorial

Paul Johnson <editor@accu.org>

Being the editor of a magazine is quite often a very enjoyable activity; I have many friends who have been editors for many years and they all say the same in terms of enjoyability and what also really annoys them.

In no real order, the annoyances are:

- Approaching deadlines and promised material has gone missing
- Poor quality material submitted
- Material submitted in an alien format that the editor cannot do anything with
- Having to continually chase new and high quality material
- Problems with copyright on material

The last one has never really been a worry for the ACCU as we never claim the copyright on the work submitted – we just have the right to publish it, non-exclusively, twice with an electronic copy in the private members area in PDF format. It does cause problems for my friends though – and oddly enough given what I've just said, for us.

While an editor does take every precaution possible to ensure that the material is original (or that excessive quoting or verbatim copying has not occurred), it is not always possible – there are thousands upon thousands of books out there and it isn't possible to read every single one, so if something slips past (which in all probability it will some day) then who, ultimately, should be blamed? The author for knowingly copying material they had no right to copy or the editor for not picking up on the infringing material?

The answer is both and both will end up with a sting. In an ideal world though, the honourable thing would be that the author pays the publishers sting – after all, it was the author that put the material there in the first place. The problem is that this is the real world and the sting placed on the publisher is likely to be much higher as publishers are seen to have much deeper pockets than the author and more than that, the publisher is very likely to have some form of indemnity insurance. Best the publisher can hope for is that the author covers the excess.

Now, if we apply this to what the majority of us do for a living and you can begin to see a problem. It was once said that if you give an infinite number of monkeys a typewriter and supply them with an infinite amount of paper, they'll come up with the complete works of Shakespeare (or was it Dickens?). In the software industry, you have the same problem (not that I'm saying you're all monkeys!)

Suppose you wish to implement an algorithm to calculate the shortest distance between two points, but have to go via a third point, you would start out and draw a triangle and use the distance from A to C to B as the basis of an answer. That would be in a perfect world. The problem is that there are humps, turns, roundabouts and the odd T junction to negotiate. You come up with a mix between topography via a helicopter analysis of the road and looking at structural maps and from there, come up with a generalised formula which more-or-less gives the correct answer. Job done.

Well, that's the theory. The code review process has not shown anything wrong and a search for anything similar has drawn a blank. The code is released, is a financial success and then, from out the blue, you're hit (well, the company) with a patent infringement action.

The problem is that this formula is well known to other people in some other field completely unrelated to computing and a patent, is a patent – if you're using the same methods or a derivative method, you're hit.

Sounds somewhat silly that and in an ideal world, just wouldn't happen. Take the following as an example. This is a true patent filing [1].

Process of Relaying a Story Having a Unique Plot

Abstract

A process of relaying a story having a time line and a unique plot involving characters comprises: indicating a character's desire at a first time in the timeline for at least one of the following: a) to remain asleep or unconscious until a particular event occurs; and b) to forget or be substantially unable to recall substantially all events during the time period from the first time until a particular event occurs; indicating the character's substantial inability at a time after the occurrence of the particular event to recall substantially all events during the time period from the first time to the occurrence of the particular event; and indicating that during the time period the character was an active participant in a plurality of events.

I have to say. They have at last invented a way to destroy all cultural development forever more. That's an achievement of a sort.

Okay, that is filed in the USA which has a patent system which is almost completely incomprehensible by us mere humans – but there is nothing to stop it being filed in the EU. The important aspect though is that by filing such a patent, and assuming it is successful, nothing is any more sacred. This is just the tip of the iceberg though. Microsoft attempted to assert a patent on double clicking and there is even one now attempting to derail XML.

Of course, it isn't happening in Europe as while we do have a patent system, it doesn't cover software. It does though cover maths and even genetics. This is why the patent action can be brought against my mythical software company. Surely though that can't happen in reality. It can and has – plenty of times with the psychoacoustic model used in MP3s being a prime example (though this is a disputed patent).

Where was I? Oh yes. Talking about the fun and games editors have...

Poor quality material is not that much of an issue for C Vu and Overload. After all, we are a professional magazine written for and by professionals. The worst I've had to see (other than one somewhat weak article) is the odd `void main()` or some dodgy method used in the middle of a function which doesn't really perform a task, but looks to be more of a cludge than anything. A quick word with the author and the matter is sorted. Other editors I know can relate other stories. The worst example was when a magazine was going to the wall and effectively all of the copy was being written by the companies behind the software.

There isn't anything wrong in doing that, as long as the editor either removes either unjustified or unverifiable claims, anything which changes the article from being a true representation of the subject and into an unpaid for advert or material which unfairly produced an imbalance with competitors software. The problem was though two fold.

We need you!

Pete Goodliffe

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

- What do you have to contribute?
- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

If seeing your name in print wasn't enough, every year we award prizes for the best published article in C Vu, in Overload, and by a newcomer.

1. The editor was that rushed off his feet that a lot went through which really shouldn't have. The value of the magazine dropped hugely in both terms of the readership appreciation and standing for impartiality.
2. The companies submitting the material didn't need to take out adverts – after all, they've just been given free reign for 3 pages of copy!

The magazine was eventually given away (literally – the company, in order to drop the title, paid for the title to go!). Not a good way to end a title.

Alien Formats. The Biggest Bane There Is and That's No Exageration!

In the past (and including other publications I've worked on), I have had material submitted in Wordstar, WordPerfect 4.2 and 5, Word 4 (Mac), Serif Publisher, EasiWriter, 1stWord+, Impression Publisher, Acorn DTP, PDF, MS Publisher and quite a few others. The problem is though that sometimes the material promised is so amazingly good, that you can't just reject it because of the format. PDF is not that bad, at least you can copy a PDF word for word (or print and use OCR to get the majority of the text). It is the other formats which causes the problems. Other than Serif and WordPerfect, the other formats are gobbledy-gook with plain text in the middle – it takes time to extract the text, but if there are any images in the file, they cannot usually be extracted. It's a pain!

Deadlinus Proximita– or "Where did the time go-itis"

There is a void space in every month where undefined behaviour exists, multiplies, breeds and does strange things to the barriers holding it. Sometimes, the forces become too large and manage to invade real space. It infects computers (which is why as a deadline for code approaches, more and more goes wrong as more items are added to the list), it infects the day (ever noticed that you go to bed later and get up earlier, but nothing more is done?), weeks just vanish as if they had never existed – they must have as the pain you felt in the tooth has now gone - and worse than that, it removes or corrupts files on the hard drive of either the contributor or the editor.

As a deadline approaches, the suck of real time becomes harder until, BANG! The deadline has been and gone and you're left chasing your tail, trying to find what you need, putting it into a publishable order and sending it over to the production editor for the finalisation magic to be performed.

Ah well, such is the life of the editor. It's a hard job, but incredibly fun!

New to C Vu!

We at the ACCU encourage a full and frank discourse between the membership and enjoy nothing more than promoting applications which are written by ACCU members that will benefit everyone. They don't have to be open source applications, they don't have to be free and they don't have to cater to many different platforms – all they have to do is benefit other members.

Define Benefit

Given that just about all those who take C Vu and Overload are professionals within the software industry, this needs to be covered. Why? To benefit me (as a Linux bod), the software would have to be free or have something which is essential and provides something which the open source alternative does not possess. I'm not that worried if it isn't open source (others may though), as long as there which benefits.

What We're Offering

If you have a piece of software which fits the above criteria, all you need to do is submit a small piece about the software and we'll do the rest.

To kick off with, `undodb` – a debugger for Linux comes under the spotlight. If you want your software promoted in this way, please just drop me an email (`cvu@accu.org`).

Things That Cause Delays

Plenty of things can delay a magazine. Waiting for a killer article, waiting for copy, waiting for the computer to be repaired –you name it, it can be delayed.

This issue is running slightly behind as it was delayed by the imminent birth of my daughter, Ashleigh Elizabeth. Well, that's happened now and she's as cute as a button.

Mother, baby and brother Richard are fine. Daddy is tired, but very happy.

Paul F. Johnson

Bibliography

[1]Jones, P. *USPTO filing* (2005) `http://www.groklaw.net/article.php?story=20051103183218268`

And So Ends Another Year

Yep, the year is at an end again. We've had another successful conference, OpenOffice2 is out, software has carried on moving at an incredible pace. Who knows what 2006 will bring! I won't predict anything as it's a pretty pointless activity, but I do know that C Vu will still be here with a whole platform of stimulating and interesting articles.

Until the next issue, can I wish all of our readers a restful and happy festive period and a prosperous New Year.

See you all in 2006!

View from the Chair

Ewan Milnes `<chair@accu.org>`

Those of you who attended the AGM earlier this year will recall that a change to the constitution was proposed, aimed at allowing greater flexibility in offering different types of membership. This was mainly motivated by the committee's desire to create enhanced corporate membership schemes. You may remember the high level of debate it triggered, and the fact that the proposal was rejected when it was put to a vote.As both the proposer of the motion and the chair of the meeting, I have to admit that the concerns raised from the floor took me by surprise, but it was quite clear on reflection that the proposal was both ill-specified and poorly explained. The blame for this lies with me, however my aim now is not to dwell on this, but to return to the issue and put it right at next year's AGM.In preparation for this, I would like to present a draft of the revised changes I plan to propose, and explain the rationale behind them. I would very much welcome any comments or suggestions. It must be granted that constitutional reform is hardly a subject to quicken the pulse, but this year's experience tells me that it is worth getting preciselyright.There are two sections of the constitution concerned with membership classes:

4.1 There shall be three classes of membership of the Association.
These shall be Basic, Full and Corporate. Individuals may hold Basic or Full membership. Companies and Institutes may hold Corporate membership for a single site. Individuals providing proof of full-time education are eligible for a discount on the Basic or Full membership fees.

7.8 At a General Meeting, each Individual Member present shall have one vote. Voting by Corporate bodies is limited to a maximum of four individuals from that body. The identities of Corporate voting and non-voting individuals must be made known to the Chair before commencing the business of the Meeting. All individuals present under a Corporate Membership have speaking rights.

4.1 defines the classes of membership, and **7.8** specifies the voting rights of each class. In fact, clearly specifying the rights of members is one of the key roles of the constitution, and this is where the previous proposal was too ambiguous.

The problem is that we want more flexibility in the benefits we can offer to members, but not to be able to change their rights. We assumed that in order to do so, we needed the ability to create new membership classes. But I now believe that this is not the case. There is no mention in the constitution of members' benefits: this is quite rightly a matter of policy. It would would not be appropriate to enshrine in the constitution the policy that Basic members are sent C Vu, Full members C Vu and Overload, and Corporate members five copies of each.

However, section 4.1 does place a restriction on the benefits we can offer to Corporate members: companies may only join for a single site. This prevents us from offering membership packages for multi-site companies, effectively placing a barrier on the scale of package we can offer. I strongly feel that this clause should be removed, and this is in fact my first proposal:

The phrase "for a single site" should be removed from section 4.1.

If this clause were removed, I would argue that we would be free to offer a scalable range of Corporate membership packages, which would suit companies of all sizes, and allow us to charge different membership fees according to the level of benefit being offered. All such members would still be classed Corporate members, regardless of the package.At this point, it is worth looking again at the two sections of the constitution quoted above. In particular, notice that section 4.1 defines Basic and Full membership classes, but then groups these under the term Individual, which is used in the rest of the constitution. In fact "Basic" and "Full" are only mentioned in section 4.1. As the only difference between these two classes of membership is in their benefits, their definition in the constitution is actually redundant. They should instead be two membership packages, both Individual class, but defined as a matter of policy rather than in the constitution. So, my second proposed change is:

Basic and Full membership should be replaced by Individual, leaving two classes of membership, Individual and Corporate.

Note that this does not propose abolishing Basic membership, we still plan to offer Basic and Full membership packages. It just means that it would not be mentioned in the constitution, but instead be a form of Individual membership.

And that is that. The changes I am proposing are quite simple, but key to enabling us to grow the association. I believe that they do so without introducing the lack of precision into the constitution that the previous proposal did. As I said earlier, I would appreciate anycomments you have on these issues, and hope that any and all concerns can be raised before the AGM in April.

Ewan Milne

Membership Report

David Hodge `<membership@accu.org>`

The journal shipment under the new system seems to have gone by without a hitch. As I am the person who creates the file for the shipments, it is important that you keep me informed of any changes in your mail address. Informing me of any changes in your email address is also important as I only send reminders out by email.

If you haven't renewed by the time the label file gets created then you do not get your journals. Please address all queries on journals not received to me as I hold a smallstock of spare journals.

David Hodge

Standards Report

Lois Goldthwaite `<standards@accu.org>`

Standards Report 1 Nov 2005

The original design goals of C++ include high performance and low-level control over hardware. These features, combined with the ability to program at a higher level of abstraction when compared with many other languages, make it an attractive choice for programmers producing complicated applications for complicated hardware. And indeed, the use of C++ in embedded environments is definitely increasing.

One prominent bit of evidence is that C++ has been chosen as the programming language for the Joint Strike Fighter plane, a project which is expected to use around 30,000 programmers at its peak. Modern aeroplanes are so complex, and so aerodynamically responsive, that software is as important to their ability to stay in the air as human pilots.

It goes without saying – but I'll say it anyway – that production values must be especially rigorous when programming such highly complicated applications where any malfunction could result in massive damage and loss of life. A dialog box saying 'Unrecoverable Application Error – please reboot your aeroplane' is not acceptable! Lockheed Martin has produced a coding standard for this project, known as JSF++, which is the product of over a year's work involving highly experienced avionics programmers, safety experts, and Bjarne Stroustrup, the inventor of C++. At the time of writing, JSF++ has not been publicly released, but this is expected to occur in the near future.

Two other initiatives have more recently been launched to develop coding standards for C++ programming in safety-critical environments. The UK-based Motor Industry Software Reliability Association (MISRA), which earlier produced guidelines for C programming which have received much attention in the industry, has chartered a committee to produce a corresponding document for C++.

And the parent body of the C and C++ (and other languages) standards committees in September created an 'Other Working Group' to investigate and document 'vulnerabilities' in various programming languages which may be used for safety-critical work. (Unlike the permanent working groups, the OWG will disband after a year unless its existence is specifically renewed.) The original motivation for this effort came from the Ada community, but they may widen their scope to C++ if some qualified experts volunteer to help with the work.

If progress reports from these groups are made available, I will try to pass on the news in future columns here. Meanwhile, if you want to volunteer to help with either of these efforts, please send a note to `standards@accu.org` and I will pass the offer on to the chairman of the appropriate group.

Lois Goldthwaite

Dialogue

Student Code Critique Competition 37

Set and collated by Roger Orr
Prizes provided by Blackwells Bookshops & Addison-Wesley

Please note that participation in this competition is open to all members. The title reflects the fact that the code used is normally provided by a student as part of their course work.

This item is part of the Dialogue section of C Vu, which is intended to designate it as an item where reader interaction is particularly important. Readers' comments and criticisms of published entries are always welcome, as are possible samples.

Before We Start

Remember that you can get the current problem set in the ACCU website (`http://www.accu.org/journals/`). This is aimed at people living overseas who get the magazine much later than members in the UK and Europe.

Student Code Critique 35 – Further Entry

Following on from the entry for SCC 35 in the last issue, Jim Hyslop `<jhyslop@dreampossible.ca>` wrote:

I'd like to congratulate Simon Farnsworth on a well thought out critique. I agree with his contention that "`GetReport()` *is a mess*." He is also correct in his final statement that the function should throw an exception. To me, that is the first and most important point I'd discuss, not the last one. Null references are not allowed, period. They invoke the dreaded Undefined Behaviour. Do your best not to accidentally create them, and never, ever deliberately create them.

My take on the class is the opposite of Simon's: it seems to me that Report is intended to add vector-like behaviour to a map, i.e. given an index *n*, find the *n*th entry in the map. Report `IS-A` map, with additional features. So, given our two different interpretations of the intent of the class, the next important improvement is to beef up the class's comments to explain in more details the "why" of the class, so that we don't have to guess at the class's purpose.

One important objective to keep in mind when critiquing other people's code, whether you are an instructor, mentor, colleague, or whatever, is not only to say what needs to be done, but why. For example, Simon recommends using `iter->` rather than `(*iter)`. Knowing why you should do this is important. Similarly, why should one use pre-increment over post-increment? I agree with the recommendation, and with his reasoning, but his reasoning is a little shallow: in what way is the code simpler? (I leave answering these questions, as well as the question: "why is preferring pre-increment over post-increment not a premature optimization?", as an exercise to the reader :-).

I do, however, disagree with his statement that "the optimizer should fix it for you". The optimizer can only fix this for builtin types. The exact type of a container's iterator is implementation-defined, so an implementation is not required to actually use a pointer: it could use a class or struct. `gcc`, for example, uses a struct as its iterator. In that case, the compiler cannot arbitrarily substitute pre-increment for post-increment. Now, back to the original submission.

I'm assuming that the member function definitions are shown inline as an SCC convention, in order to conserve space. Member functions should not, of course, be written as inline until and unless profiling shows them to be a bottleneck.

Retrieving the *n*th element in a map can be simplified in user code by taking advantage of the distance properties of iterators:

```
Report & Reports::GetReport( int index )
{
  if ( index <0 || index >=size() )
    throw out_of_range;
  return (begin() + index)->second;
}
```

- From 17 lines down to 3. Not bad.

Note that the iterator's `op+` does not do any range checking. If `index` is negative, or exceeds the size, you will invoke undefined behaviour (a crash, if you're lucky) so we need to do strict checking on `index` before we do anything else.

This is not necessarily more efficient than the original code, since iterator's `operator +(int distance)` (which is implemented in terms of `operator +=`) basically boils down to:

```
if ( distance >= 0 )
  while ( distance-- ) this->operator++();
else
  while ( distance++ ) this->operator--();
```

but it is easier to read and follow in user code.

If profiling indicates we're spending a lot of time in the increment, then this can be optimized by determining whether `index` is closer to `begin()` or to `end()`:

```
Report & Reports::GetReport( int index )
{
  if ( index <0 || index >=size() )
    throw out_of_range;
  if ( index > size()/2 )
    return (end() - (size() - index))->second;
  return (begin() + index)->second;
}
```

On variable naming: What is `nIndex`? Well, clearly it's an index, but what is its purpose? If one is going to use Hungarian Notation then that can be clearly indicated by using the proper prefix `i` rather than the generic `n` (remember, in true, Simonyi notation[1], `i` does not mean `integer`, it means `index`). `iter` is, similarly, clearly an iterator, but again: what is its purpose? The variable names do not convey any meaning to the reader. The author is also using a mix of Hungarian Notation, and non-HN. Pick one or the other, and stick to it (as an aside, the `iter` highlights the problems with using Hungarian Notation in C++ programming: each time you create a new type, you need to invent a new prefix. Personal preference: avoid HN, as it adds very little value.).

So, without good, descriptive variable names, we resort to studying the code to determine what the variables are. As Simon pointed out, the member variables seem to be an optimization (side note: in addition to the robustness concerns he noted, the optimization is neither thread-safe nor reentrant, although I suppose that could all be lumped under 'susceptible to changes between calls'. Side note 2: the author of the code seems to have recognized at least some of the problems Simon pointed out, since there is a test to see if `nIndex` exceeds the size of the map, implying that the author knows some elements could be removed between calls to `GetReport`). So, um... where was I? Oh, yeah. The next thing to do is give the member variables meaningful names, make them private,

```
private:
  int lastIndexRetrieved;
  iterator lastElementRetrieved;
```

and, of course, initialize them in the `ctor`:

```
Reports() : lastIndexRetrieved( 0 ),
    lastElementRetrieved( begin() ) {}
```

Notice that I did **not** recommend adding a comment to explain their purpose, I specifically recommended changing the variable names.

If the robustness concerns Simon noted are addressed (not likely, but possible if `Reports` is populated once and not modified), and profiling indicates we are spending a lot of time incrementing/decrementing iterators, then `GetReport` can be optimized as follows:

```
Report & Reports::GetReport( int index )
{
  if ( index < 0 || index >= size() )
    throw out_of_range;
  int distance = index - lastIndexRetrieved;
  lastIndexRetrieved = index;
  lastElementRetrieved += distance;
  return lastElementRetrieved->second;
}
```

1 Charles Simonyi, 'Hungarian Notation', http://msdn.microsoft.com/library/en-us/dnvsgen/html/hunganotat.asp

A variation on the `begin()/end()` optimization shown above can be applied, by determining which is the closest iterator to the desired index: `begin()`, `lastElementRetrieved`, or `end()`. If, for example, there are 1000 elements in the map, the last element retrieved was 14, and we want element 990, then it is faster to count back from `end()` rather than forward from `lastElementRetrieved`. But again, only if profiling indicates we're still spending a lot of time iterating through the map. You may end up spending more time figuring out which element is closest, than actually spinning through the list!

`ClearAll()` appears to me to be a convenience function, to allow the user to reset all data in the reports with a single function call, for subsequent re-population. Given the ownership issues Simon pointed out, we must assume that some external mechanism is responsible for adding and removing `Report` objects to and from the map, and for cleaning up the data on program termination. There are similar ownership issues around the key, `Data *`, which we again must assume some other object will take care of. Examining these lifetime issues would be part of a general code review.

In `ClearAll()`, I would suggest the programmer consider get in the habit of using `std::transform()` and `std::for_each()` instead of manually iterating through loops.

Student Code Critique 36 Entries

Here is a C program generating a couple of prime numbers as part of an exercise on encoding/decoding with public and private keys. There are two bugs with the program: it produces the same output each time it is run with one compiler (msvc) and it loops forever with another (gcc). Please critique the code to help the student resolve both these problems with the algorithm. Additionally suggest any improvements to the coding style and point out any other issues with the algorithms used. You can also broaden the critique to include a C++ solution if this may assist the student with their original task.

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
//need to generate number, then find out
//whether it is a prime, twice. Then need to
//generate e and see if it is a factor of n.
  int i1, rem1, i2, rem2, i3, rem3, rem4;
  int p, q;
  int n, phi, e;
  //These are the two prime numbers output
  int m, d;
  i1 = 0;
  i2 = 0;
  i3 = 0;
  while(i1!=1)
  {
    p = 100 + 99*rand()/((double)RAND_MAX+1);
    //p is random number between 100 and 200.
    i1=p-1;
    rem1 = p%i1;
    //find out whether the number is prime
    while(rem1!=0)
    {
      i1--;
      rem1 = p%i1;
    }
  }
  while(i2!=1)
  {
    q = 100 + 99*rand()/((double)RAND_MAX+1);
    i2=q-1;
    rem2 = q%i2;
    while(rem2!=0)
    {
      i2--;
      rem2 = q%i2;
    }
  }
  n = p*q;
  phi = (p-1)*(q-1);
  // phi is the number of primes less than n!
  // e picked such that gcd(e, phi) = 1
  while(i3!=1)
  {
    e = phi*rand()/((double)RAND_MAX+1);
    //e is a random number between 0 and phi.
    i3=e;
    rem3 = phi%i3;
    rem4 = 1;

    //this loop finds the highest value of i3
    //which divides both numbers. It needs to
    // be 1, so they are relatively prime
    while(rem4!=0 )
    {
      i3--;
      rem3 = phi%i3;

      if(rem3==0)
        rem4 = e%i3;
    }
  }

  //the loop will find the value of m such
  //that e*d mod phi = 1.
  m = 0;
  while((e*d)%phi !=1)
  {
    d = (m*phi+1)/e;
    m++;
  };
  printf( "(m,d) is (%i,%i)\n", m,d );
  return 0;
}
```

From Seyed H. Haeri (Hossein)

<shhaeri@math.sharif.edu>

When I start scanning this code, the first thing which meets my eyes is its appropriate commenting. So, one positive point for the student!

Next, I see the variable declarations at the beginning of the `main()` function. As mentioned in the problem specification, this is a C code, so the student has probably not had any other option. In case he/she is about to move to C++, which the spec lets us to assume so, the programmer should postpone them as late as possible.

Another point which arises whilst moving from C to C++, is that because there is no more any need to use `printf()`, and we use `cout` in return, it suffices to replace the top #includes with a mere `#include <iostream>`. (Note that I've dismissed the tailing `.h` on purpose.)

A negative point which I give to the student is because of the lack of good interaction with the user (of the programmer); the program outputs "*(m,d) is …*" without saying what `m` and `d` are. The program is supposed to say that beforehand. More explanation of what the program is about to do is not that bad. Furthermore, *(m,d)* is not a good representation of the purpose. If we suppose that somebody may some day use this program, that's mathematicians. Mathematicians are very likely to misunderstand (,) with GCD – as is usual in Number Theory.

I'm not sure whether this is an assignment in which the student is supposed to implement the algorithm for producing two random prime numbers. Assuming it is not, it suffices to use the `Boost` stuff to do that within much less lines. I don't know for when this code is, but if it is for recent years, the program is, therefore, overkill.

But what has caused the bug reported by the student? Well, a known problem: Random Number Generation. It turns out that built-in random generators such as `rand()`, although really produce random numbers, don't produce what we human-beings may consider so. What has caused the students report is also the same. He/she has encountered the problem because `rand()` produces the same number in consecutive invocations. That is, `rand()` of MSVC is always producing the same prime number, and that of GCC is always producing the same non-prime number. Therefore, MSVC always outputs the same number, and GCC goes in an infinite loop. (Note that this is a consequence of program's logic.)

Another important point about this code is the lack of good modularisation in it; none of those loops should be more than a function call. Assuming that the programmer knows how to implement each of below functions, (and he/she adds appropriates comments as well) he/she should have written something like this:

```
#include <iostream>
…
int main()
{
  int p = gimmmeARandom(100, 200);
  int q = gimmmeARandom(100, 200);
  int n = p * q;
  int phi = (p - 1) * (q - 1);
  int e;
  do
  {
    e = gimmmeARandom();
  }
  while(gcd(e, phi) != 1);
  int m = 0, d = 0;
  while((e * d) % phi != 1)
  {
    d = (m * phi + 1) / e;
    ++m;
  }
  cout << "Two random prime numbers
     : " << m << d;
  return 0;
}
```

Finally here are a few minor points:

1. The programmer has always used the postfix `++` operator, whilst in all of them, the prefix version does the job. So, he/she should replace all of them.
2. The variable `d` is not initialised. I don't know about C, but in C++, local variables are not automatically initialised. Therefore, the student should manually initialise that.

I suggest the student should read about the Boost Random Number Generation library.

From Ken Duffill `<ken@kendee.co.uk>`

On first look this code is quite busy and difficult to follow. There is a bunch of declarations, all with pretty meaningless and very similar names. Then there are four loops, obviously this is where the work goes on, but neither the original description, nor the comments, nor the code itself make clear what is meant to be going on.

This kind of code is very common with beginners, and even experienced developers like myself (I have been programming for nearly thirty years) may produce code like this during some quick prototyping excercise. The next phase of development, though, MUST be to get the code into some sort of order.

I will go through the process step by step so as to try to show what changes I would like to make and why I think they are important. Once we have done this it is possible that at least some of the bugs will have become obvious and been resolved along the way and the code is definitely going to be more easy to understand so if there is a fundamental flaw in the design we should be able to pick it up.

Right from the start we see that the code is one long function called `main.` Even though it isn't that long; one long function always leads to trouble. It is difficult to tell where the variables have any meaning. They are all declared in the same place and then some are used and forgotten, others are used to carry data from one part of the program to another, but it is not easy to tell which is which.

So, the first step is to put each of the four loops into separate functions, and only leave those variables declared in the main function that are needed to carry data between the functions.

Looking at the first loop, it seems that the variable `p` is the only data that is needed once the loop has finished. So we create a function (we will call it `function1` for now, once we understand its real purpose later on we will think of a more descriptive name) that returns an `int`. We cut the loop from `main` and paste it into the new function, replacing the original with the assignment `p = function1();`

Now we remove `i1` and rem1 from the declarations in `main` and put them into `function1` and add a new `int p` declaration in `function1`, and return `p` when the function is complete.

Compile the code using both compilers, to make sure we haven't broken anything.

Oops! The compilers baulk at a statement in `main i1 = 0;` So we cut that from `main` and paste it into `function1`. At this point one of my personal preferences takes over for a moment and instead of having the first few lines of function1 look like this:

```
int i1, rem1;
int p;
i1 = 0;
```

I change it to:

```
int i1 = 0;
int rem1;
int p;
```

That is to say one declaration per line and if a variable needs initialising, do it in the declaration so you don't forget.

Compile the code using both compilers, to make sure we haven't broken anything. That's better!

Run both versions, to make sure we haven't changed its behaviour.

Next we do the same for the second loop, calling this `function2`.

Now we can see that functions one and two are only different in that one has variables `i1`, `rem1` and `p` and the other variables `i2`, `rem2` and `q`. If we change `i1` and `i2` to `i` and `rem1` and `rem2` to `rem` in both functions and then change `q` to `p` in the second function, the functions become identical, so `function2` can be deleted and `main` can make two calls to `function1`, one assigning the result to `p`, the other assigning the result to `q`, instead.

In examining the next loop in `main` to decide how to turn it into a function we notice that the two lines of code before the loop include one that assigns `n` to `p*q`, and yet n is never used in the code. It is only referred to in the comment that states, "`phi is the number of primes less than n!`"

Just to make sure that we haven't missed something we delete this line, and the declaration of `n` and recompile. Sure enough, the compiler is happy. Now, if we have any "domain knowledge" we will understand that the value `n = p * q` is a very important part of the cryptographic algorithm so we may choose to leave it in for future use. If we do then we should place a comment to this affect. If we decide to remove it we need to change the comment to "`phi is the number of primes less than n!. Where n was the product of p and q.`"

We now create a function (`function3`, for now) that takes an integer parameter and returns an integer result. Cut the loop from `main` and paste it into the new function, replace the cut code in `main` with a call to `function3` passing in `phi` as calculated just before the loop, move all appropriate declarations from `main` to `function3`. Exactly as we did for `function1` and `function2`. We then compile and run to show we haven't broken anything.

Ooops, exactly as happened for `function1` and `function2` the compiler baulks because there is an assignment of `i3 = 0` left in `main`. Move this assignment into `function3`, taking this opportunity to implement my preferences (one declaration per line and if a variable needs initialising, do it in the declaration so you don't forget).

Compile again, and run.

Don't despair that we haven't fixed the bug yet, `gcc` still produces an `exe` that loops forever, and MSVC still gives us the same answer every time we run it. That's OK we haven't tried to fix anything yet we are just getting the code 'in shape'.

We repeat the refactoring process one last time, for the last loop. This time `function4` needs to accept two integer parameters (`e` and `phi`) and doesn't return anything, it just prints the results. Note the initialisation of `m = 0` just before the loop, let's do that in the declaration of `m` in the function this time.

Now `function4` is very small, and we notice straight away that there is an unnecessary semicolon at the end of this `while` loop, which we remove. Isn't it much easier to see these things in small functions? Also we see that there is an `int d` that is declared and then used, in the conditional of the `while` loop, before it has been initialised. Could this be our bug?

So following my preferences we want to initialise `d` in its declaration, but what do we initialise it to? Well in the body of the loop `d` is set to `(m*phi+1)/e` so lets use that. We know that `m` is initialised to zero so the initialisation of `d` becomes `1/e` because `m*phi` is zero. Now we see that in the loop, once `d` has been set `m` is incremented so maybe, because we have set `d` in the initialisation, `m` should be initialised to `1` rather than zero. Essentially we have done the first pass through the loop in our initialisation, at the same time as making sure we don't get any surprises if the runtime environment doesn't initialize in the way we expected.

Was this our bug? Well, no. Having compiled and run the code we get the same behaviour as before. But the code quality has improved.

Now our main is looking much smaller and more understandable. If we adopt my preferences for declarations we have a `main` that looks like this:

```
int p = function1();
int q = function1();
int phi = (p-1) * (q-1);
int e = function3(phi);
function4(e, phi);
```

with a few blank lines and comments in between these lines of code.

Now we can see that if the variable and function names made some more sense we might actually be able to see what is going on!

Just by looking at the comments we can get some better variable names in `main`.

- `p` and `q` are random numbers that are prime.
- `phi` is the number of primes less than `n!` Where `n` was the product of `p` and `q`. I could change this name to `xPrimes` or something similar, but it may well be that the algorithm being implemented is well known and EVERYBODY who knows it expects to use `phi`, in which case it would be better to leave it.
- I am still not sure what `e` is so we will leave that.
- `function1` clearly calculates a random number that is prime, cos that is what `p` and `q` are.

Also, it seems as if `e` and `phi` are only used as transports between `function3` and `function4`, so by moving the call of `function3` into the beginning of `function4` we can get `main` to look like this:

```
int p = FindRandomPrime();
int q = FindRandomPrime();
function4(( p - 1 ) * ( q - 1 ));
```

We could go one stage further now and lose `p` and `q` altogether the whole of `main` just becomes:

```
function4(( FindRandomPrime() - 1 ) *
  ( FindRandomPrime() - 1 ));
```

But that may be a step too far. Again, with some "domain knowledge" we will find that `p` and `q` are important numbers in the whole cryptographic process, so we will leave them be.

Our bug is still present though, and we need to look at the functions more closely.

The `FindRandomPrime` function and `function3` are very similar. Inside a loop we get a random number in a given range. Let's see if we can factor this bit of code out.

We create a function that returns an `int` and takes two `int` parameters. It is quite obvious now what this function is going to do, so we will give it a useful name straight away. The prototype will be something like this:

```
int GetRandomInRange( int from, int to );
```

We cut the line from `FindRandomPrime` and paste it into `GetRandomInRange`, and refactor it to account for the parameterisation so it becomes:

```
return from + (to-from-1)
*rand()/((double)RAND_MAX+1);
```

replacing that line in `FindRandomPrime` with a call to `GetRandomInRange` thus:

```
p = GetRandomInRange(100, 200);
```

We almost don't need the comment anymore. We replace the line in `function3` similarly:

```
e = GetRandomInRange(0, phi);
```

Comple and run... The bug is still there.

Now we ask ourselves does `GetRandomInRange` work? It is a one line function, so we should be able to understand it now. `rand()` returns a random integer between `0` and `RAND_MAX`, and this function appears to try to convert this to a random integer between from and to.

With this implementation there is a mixture of implicit and explicit casting in this expression. Now, as I have said earlier, I have been developing software for nearly thirty years, more than fifteen of which have been in C. I have the C standard on my desk and I still can't tell for sure how this expression will evaluate. Some integers get cast to doubles, and then the doubles get cast back to integers, but which, where and why? I dunno.

It is my belief that, even if I knew that this expression could be relied upon in all implementations to give me the correct answer, I cannot rely on everybody who may need to look at this code in the future to be able to, and if they can't they are going to break it someday (if it isn't already broken, of course). So as a matter of principal, I don't like expressions that mix explicit and implicit casts.

If we rewrite this expression as two expressions so as to avoid implicit casts altogether we get:

```
double range = ( double )rand()
  /((double)RAND_MAX + (double)1.0 );
return from + (int)(
  (double)(to-from-1)*range);
```

When we compile and run we now do not loop indefinitely in the `gcc` version. Hey, success! Of course, it gives us the same answer every time we run the application, but at least our two compilers now produce an application with the same behaviour.

I am still not entirely sure that this function actually delivers what it seems to want to in extreme cases, and there may be a hidden bug in the application as a result.

I would prefer to see:

```
double range = (double)rand() /
  (double)RAND_MAX;
return from + (int)((double)( to - from )
  * range );
```

Now that this code is contained in one simple function, some unit tests should be written to prove the answer is correct in all cases (or fix the code until it does). I will leave that as an exercise for the reader.

So, why are our random numbers not random. A quick look at the standard (or any C standard library reference) tells us the answer lies with `rand`. `rand` doesn't actually produce random numbers; rather, it selects numbers from a sequence of pseudo-random numbers. The starting point of this sequence will always be the same unless the `rand` library is properly initialised. `srand` should be called somewhere in the application before the first call to `rand`. The parameter to `srand` should be some value that is different every time the application runs. It is quite usual to use a value from the system timer for this purpose. So I will use a call to time thus:

```
srand((int)time(NULL));
```

This, of course will only work so long as the application is not run more frequently than once per second. Whilst `srand` is called at the application level this is reasonable, because we humans are slow beasts, but care must be taken if this code is refactored into libraries that may be called rapidly from other code.

We now have tidied our code up considerably, found the two causes of the problems observed and two other problems that had not yet produced observable errors. But there is still more that can be done. With a little more "domain knowledge" we can find better names for `function3` and `function4`.

I did some internet searches based upon the comments like "`e` picked such that `gcd(e, phi) = 1`" and "this loop finds the highest value of `i3` which divides both numbers. It needs to be 1, so they are relatively prime" and "the loop will find the value of `m` such that `e*d mod phi = 1`". From information gleaned, I refactored `function3` into `FindRandomRelativePrime` that calls functions called `Euclid` and `FindLargestFactor`. I am not a Crypto expert and I still can't find a better name for `function4`. I have attached the code so maybe someone else can finish the job.

Now that all the functionality is refactored into small, well named, functions whose purpose is clear and unambiguous then each function can be tested with unit tests, preferably automatically built and run as part of the development environment every time the code is changed.

Who knows what other problems will have been identified and fixed once this process is complete.

You will notice if you look at my code that I have changed the post-increments and post-decrements to pre-increments and pre-decrements. There are cases when the code produced can be more efficient if pre- rather than post- operations are used. It is true that in C these cases are rather rare and insignificant, but in C++ when the objects being pre- or post- operated are not native types but large objects then the saving of the creation of one temporary can be significant, and it is therefore a good habit to get into.

One final comment. The original problem stated that the application behaved differently when built with `MSVC` than with `gcc`. It is good practice to build your applications with more than one compiler whenever you can. Some interesting differences can come up.

When I was refactoring the code for `FindLargestFactor`. I decided that the original algorithm was marginally wasteful as the largest factor must be less than or equal to half the candidate. Because of this the candidate must be greater than 2 in order to have any factors. I put a conditional in right at the beginning of the function, but being lazy I only compiled with the `gcc` compiler. There was no problem. It was some time later when I compiled with `MSVC` again and the compiler complained because it didn't like declarations that follow code. Although the C99 standard allows this, for maximum portability it is not good practice. If you really MUST have new declarations after code then they should be put in a new block by the appropriate use of curly braces. In this case there is very little overhead to just moving the conditional to just after the declarations, but that isn't always the case.

Another incident was when I began using time to seed the random sequence. Again `gcc` was happy, but `MSVC` complained that I hadn't `#include`d `time.h`

One final final comment. It is also a good idea to pass your code through a static code checker such as Lint (or PcLint), this can also pick up some problems. Probably the problem with implicit casting that was the first bug we cured in this project would have been caught by a static checker.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int GetRandomInRange(int from, int to)
{
  double range = (double)rand() /
    (double)RAND_MAX;
  return from + (int)((double)( to - from )
    * range );
}
int FindLargestFactor(int candidate)
{
  // The largest factor of a candidate cannot
  // be bigger than half of the candidate,
  // so lets start there.
  int potentialFactor = (candidate >> 1);
  // The concept of factors only applies to
  // numbers greater than or equal to 2.
  if (candidate < 2)
  {
    return candidate;
  }
  // If there is no remainder when candidate
  // is divided by potentialFactor
  // then potentialFactor is a real factor.
  while(candidate % potentialFactor != 0)
  {
     // There was a remainder so lets try
     // another potentialFactor.
     --potentialFactor;
  }
  // We will always end up with a factor even
  // if it is one.
  return potentialFactor;
}
int FindRandomPrime()
{
  int p;
  do
  {
     p = GetRandomInRange( 100, 200 );
  // Find out whether p is prime.
  // If the largest factor of p is one then p
  // is prime, so we have finished.
  } while ( FindLargestFactor( p ) != 1 );
  return p;
}
int Euclid( int larger, int smaller )
{
    // Find whether the greatest common
    // denominator (gcd) of the two
    // parameters is one or not.
    int divisor = smaller;
    int dividend = larger;
    int remainder;
    while (( remainder = dividend % divisor )
           > 1 )
    {
        dividend = divisor;
        divisor = remainder;
    }
    //If the result is one then it is the gcd,
    //otherwise the result is zero. We are not
    //interested what it is, but if we were it
    //would be dividend at this point.
    return remainder;
}
int FindRandomRelativePrime( int phi )
{
  int e;
  do
  {
    e = GetRandomInRange( 1, phi );
  } while ( Euclid( phi, e ) != 1 );
  return e;
}
void function4( int phi )
{
  // phi is the number of primes less than
  // n!.  Where n was the product of p and q.
  int e = FindRandomRelativePrime( phi );
  int m = 1;
  int d = 1 / e;
  //the loop will find the value of m such
  //that e*d mod phi = 1
  while(( e * d ) % phi != 1 )
  {
     d = ( m * phi + 1 ) / e;
     ++m;
  }
  printf("(m,d) is (%i,%i)\n", m,d );
}
int main()
{
  srand((int)time(NULL));
  {
    int p = FindRandomPrime();
    int q = FindRandomPrime();
    function4(( p - 1 ) * ( q - 1 ));
  }
  return 0;
}
```

From Jim Hyslop <jhyslop@dreampossible.ca>

Let's deal with gcc's behaviour first. In debugging, `printf` can be your best friend, especially if you don't have an IDE to step through the code. So, after putting in various `printfs` to find the checkpoints (i.e. "About to find p", "About to find q", and so on), we find that it's never exiting the first loop:

```
while(i1!=1)
{
  p = 100 + 99*rand()/((double)RAND_MAX+1);
  //p is random number between 100 and 200.
  i1=p-1;
  rem1 = p%i1;
  //find out whether the number is prime
  while(rem1!=0)
  {
    i1--;
    rem1 = p%i1;
  }
}
```

Oops – what's this: `99 * rand()`. If `rand()` is greater than approx. 21 million (which it will be about 1 time out of 100) then it will overflow. We don't want that, since overflows are undefined behaviour (which, by the way, is probably why `MSVC` “works”). So, to fix it, we can just force the numerator to a double by using the constant `99.0` instead of `99`:

```
p = 100 + 99.0*rand() / ((double)RAND_MAX+1);
```

We don’t need to worry about the denominator overflowing, since `RAND_MAX` is first cast to a double, and **then** incremented.

Looking further down the code, we see that the random number generation is repeated. This needs to be refactored into a function which generates a random number in a given range:

```
int GenRand( int min, int max )
{
  return min + (double)max *
    rand()/((double)RAND_MAX+1);
}
```

Doing this by the way, also eliminates the need for the comments `//[varname] is a random value between x and y.`

Now, when you run it, `gcc` and `MSVC` behave the same: you always get the same results. That’s because `rand()` is not a true random number generator: it generates pseudo-random numbers in a sequence determined by the algorithm used. Unless you provide it with a unique seed each time you start the program, it will generate the same sequence of numbers. So the first thing to do is seed the random number generator. For purposes of this exercise, `srand( time(NULL) );` before the first call to `rand()` will suffice. Now, the program will generate different numbers each time you run it (unless you run it twice within one clock second).

Style Comments

Even C has the concept of local block variables. Take advantage of them, and declare variables in the most restrictive scope possible. The variables `i?` (e.g. `i1`, `i2`, etc.) and `rem?` are each only used within one while loop, so they can be declared there.

None of the variables are initialized. Uninitialized variables account for a large number of programming errors, so variables should be initialized whenever possible. It also makes the meat of your functions more compact, since you don’t occupy space within the logic simply initializing variables.

The variable names are not very useful. `i1`, `i2`, `i3` are meaningless. I seem to recall that `p`, `q`, `m`, `d`, `n`, `e`, and `phi` are specific terms used in the cryptographic equations so that is reasonable – but there should be a comment to that effect.

Comment placement is inconsistent. Some comments are placed before the line in question, some after, e.g.:

```
p = 100 + 99*rand()/((double)RAND_MAX+1);
//p is random number between 100 and 200.
[...]
//find out whether the number is prime
while(rem1!=0)
```

Consistency is important in allowing other programmers to easily read and understand your program. The comment for `p` should be placed before the statement.

Avoid repetition. There are two loops that perform identical work – they should be refactored (yes, you can refactor even in C) into a function. Applying meaningful names for `i1` and `rem1`, we get:

```
int FindRandomPrime( void )
{
  int primeCandidate, factor=0;
  while(factor!=1)
  {
    int remainder;
    primeCandidate = GenRand( 100, 200 );
    factor=primeCandidate-1;
    remainder = primeCandidate%factor;
    //find out whether the number is prime
    while(remainder!=0)
    {
      factor--;
      remainder = primeCandidate%factor;
    }
  }
  return primeCandidate;
}
```

This is a horribly inefficient way of finding primes. It tries all numbers from 1..`primeCandidate`. You don’t need to try any even numbers except 2, and you don’t need to try any numbers below `sqrt( primeCandidate )`. A much more efficient function might be:

```
int IsPrime( int value )
{
  int factor1 = 3,
      factor2 = sqrt( value ) + 1,
      divisorFound= value%2 == 0;
  while ( !divisorFound && factor1<factor2 )
  {
    divisorFound = value % factor1 == 0;
    factor1 += 2;
  }
  return divisorFound;
}
int FindRandomPrime()
{
  int primeNumber;
  do {
    primeNumber = GenRand( 100, 200 );
  } while ( !IsPrime( primeNumber ) );
  return primeNumber;
}
```

Note, by the way, that I’ve split `IsPrime` into a separate function: this allows you to hook in unit tests to validate that your `IsPrime` function works correctly.

I don’t have my copy of C99 handy, but I’d double check to see if `//` comments are allowed in C99. I know they are not allowed in C90.

A little more liberal use of whitespace would help readability (although that could be a restriction imposed by the printing requirements in the magazine).

Commentary

With three such varied entrants I don’t think much additional commentary is needed – the main items I wanted addressing are all covered: the amount of code duplication, the oddly predictable behaviour of `rand()`, and the problems caused by the mix of int and double variables.

However no one correctly identified the reason for the difference in runtime behaviour between `MSVC` and `gcc`. It is in fact caused by the different values for the (implementation defined) value `RAND_MAX`. This is defined as `0x7fff` with `msvc` and `0x7fffffff` with `gcc`. With `msvc` the expression `99*rand()/((double)RAND_MAX+1)` will evaluate to a number from 0 to 98 but with `gcc` on the platform being used it can only be 0 or -1 (caused by signed integer overflow).

The Winner of SCC 36

The editor’s choice is:

Jim Hyslop with a strong commendation to Ken Duffill

Please email `francis@robinton.demon.co.uk` to arrange for your prize.

Student Code Critique 37

(Submissions to `scc@accu.org` by Jan 3rd 2006)

For a change I thought I’d set a Java problem. Here is a student’s attempt at a simple class to provide access to a single database table, ADDRESS. Please critique the code and suggest what problems there may be with this class when using it in a larger application, and any issues with the simple test harness.

```
import java.sql.*;
class Scc37 {
  String[] drivers = {
    "sun.jdbc.odbc.JdbcOdbcDriver" };
  String database = "jdbc:odbc:ADDRESS";
  void setDrivers( String[] drivers ) {
    drivers = drivers;
  }
  void setDatabase( String database ) {
    database = database;
  }
  String selectAddress( String query ) {
    try {
      for ( int idx = 0;
            idx != drivers.length; ++idx )
        Class.forName( drivers[ idx ] );
```

[concluded at foot of next page]

Conference Report – AgileNorth

by Phran Ryder `<Phran@AgileNorth.org.uk>`

The First AgileNorth.org.uk conference was the first that I have been involved in organising, for nearly a decade. Frankly, I couldn't have wished for a better day, I couldn't have wished for more attendees, I couldn't have hoped for a more enthusiastic set of delegates, and I couldn't have got a better set of speakers. Oh and the generous sponsorship of UCLAN, The DSDM Consortium, BCS SPA, the Agile Alliance, Exoftware, and Rutherford Software helped to keep the admission price very low!

AgileNorth and the conference are for local technical and business staff who wish to learn and share their experiences of **becoming** and **being** agile. If you are not from the North we can still give you something and I am certain you can give use something. Please come to our monthly meetings detail of which are on our website `www.AgileNorth.org.uk` but be aware that Internet Service provider is changing so the web site is unreliable at the moment.

The sessions covered a wide range of Agile related topics: What is Agile, Agile in Large Organisations, XP Teamwork, Fitting Agile into non Agile organizations, Managing Agile projects, Test Driven Development, Planning, Refactoring and Experience reports.

Kevin Rutherford of Rutherford Software introduced agile development using the techniques of agile development. This was a fascinating. Kevin asked us what our key requirements where. This took the form of a list of questions of what we wanted to know about Agile software development. We then had several 10 minute iterations in which:

1. We, the audience, prioritized the requirements.
2. Kevin, implemented the requirements by answering the high priority questions.
3. Performed acceptance tests - the person who had asked the question stated whether the question was answered.
4. Raised new requirements – which were new questions people had.

Three parallel sessions then ran:

1. Steve Ash OOTAC (Object Orientated Training and Consultancy) outlined and discussed the philosophy, principles and process of Enterprise XP. Many people are drawn to Agile practices but many managers, at all levels, are wary of the lack of robust project governance and management (perceived in some cases). The practices have been proved to be advantageous for the day-to-day project activities but what is seen to be needed is a repeatable, higher level of visibility of the direction and progress of the project. EnterpriseXP is a 'work-in-progress' that has taken appropriate elements of PRINCE II and DSDM to add governance to XP projects initially.
2. Charles Weir of Penrillian discussed the interaction of external roles of XP teams. This session was a gold fish bowl. In golf fish bowls session the chairs are arranged in a circle. In the center of the circle are four more chairs. At any one time three of these central chairs are occupied. Only those in the central chair can speak. If you want to speak you sit in the fourth chair at which point one of the other three must leave. This format encouraged varied, interesting and informative debate.
3. Lindsay McEwan and Gavin Hope of Nonlinear Dynamics gave an experience report of their successes and failures on their road to becoming and being agile.

Lunch at the conference center was simply delicious and was accompanied by animated debate of the sessions so far. The first afternoon session was again split into three.

1. Jim Sutton – exchanged experiences of managing agile projects.
2. Sean Heally of Exoftware explained, demonstrated, and led a discussion on how to do Test Driven Development (TDD).
3. Isobel Nicholson and Peggy Gregory of the University of Central Lancashire (UCLAN) led a session looking at what we can (or should) do if we do not have an ideal environment but still want to be (more) Agile. For example, where senior management is still doubtful about the agile approach or where you are not empowered to make decisions?

The final session was split into two:

1. Rachel Davies of the Agile Experience introduced XP using the XP Game. This is a playful way to familiarize people with some of the more difficult concepts of the XP Planning Game, like velocity, story estimation, yesterday's weather and the XP lifecycle. This session was created by the XP Belgium group and more info can be found at `http://www.xp.be/xpgame/`.
2. Ivan Moore and Duncan Pierce who led an interaction session demonstrating how to refactor your code.

Phran Ryder

[continued from previous page]

```
    } catch (Exception e) {
      System.out.println(e);
    }
    String userName="";
    String password="";
    // Get connection
    Connection con = null;
    try {
      con = DriverManager.getConnection(
        database,userName,password);
    } catch(Exception e) {
      System.out.println(e);
    }

    // Execute query
    ResultSet results = null;
    try {
      results =
        con.createStatement().executeQuery(
        "SELECT * FROM ADDRESS WHERE "
        + query );
    } catch (Exception e) {
      System.out.println(e);
    }
    // Get all results
    String retVal = "";
    try {
      ResultSetMetaData rsmd =
        results.getMetaData();
      int numCols = rsmd.getColumnCount();
      int i, rowcount = 0;
      // break it off at 50 rows max
      while (results.next() && rowcount<40) {
        // Loop through each column, getting
        // the data and displaying
        for (i=1; i <= numCols; i++) {
          if (i > 1) retVal = retVal + ",";
          retVal = retVal +
            results.getString(i);
        }
        retVal = retVal + "\n";
        rowcount++;
      }
    } catch (Exception e) {
      System.out.println(e);
    }
    return retVal;
  }

  /** Test harness - 'args' list drivers */
  public static void main( String[] args ) {
    Scc37 scc37 = new Scc37();
    scc37.setDrivers( args );
    System.out.println( "Found:" +
      scc37.selectAddress(
        "name LIKE '%Hardy'" ) );
  }
}
```

Francis’ Scribbles

by Francis Glassborow <francis@robinton.demon.co.uk>

Safety Critical Programming & C++

Scott Meyers recently posted a request to comp.lang.c++.moderated for information about any current uses of C++ in safety critical programming. I am not going to report on the resulting thread other than to say that there is one EC++ (embedded C++) compiler (DO-178B Level A certifiable Embedded C++ (EC++) for its safety-critical INTEGRITY 178B RTOS) that meets pretty stringent requirements for use in a specific environment. It will come as no surprise that this compiler is based on the EDG front end coupled with the Dinkumware EC++ Library.

The language used for safety critical programming worries me far less than the human beings using it. This is the issue that I want to address here.

If you want to work in the UK on gas (note that this is not ‘gas’ in the US sense of the word) central heating systems you cannot legally do so without certification. Going out and buying a fist class set of professional tools that meet the BSI standards for use with gas appliances does not even start to turn you into a gas central heating engineer, nor should it.

Joining a professional body for plumbers is not enough either. You need competence as a plumber to deal with a central heating system, however you need something more before you are legally allowed to touch a customer’s gas appliances; you must be CORGI registered (for more information see http://www.corgi-gas-safety.com/section_about/corgi_council.asp). I have no doubt that there are many people who are competent to work on gas central heating systems who are not CORGI registered, however these people cannot legally do any work on such a system. Any professional (as opposed to cowboy, fly-by-night) plumber would know what the limits were on his work; no CORGI registration, no touching gas appliances.

Notice that there is no requirement that those designing gas appliances be registered or even have any professional qualification. Their designs will have to be tested according to the standards laid down by their National Body.

Who writes those Standards? People who jointly understand various safety aspects of using gas. It would be entirely coincidental if any of those people were CORGI registered. The collective knowledge of the committee writing a standard is important and so such a committee would benefit from being widely based.

Now start applying the same ideas to programming safety critical systems.

I find it disturbing that we do not yet have an adequate form of certification that covers both the basics of programming safely and requires endorsement for specific computer languages.

Having a certified EC++ compiler for a specific RTOS is great but that in itself is only a very small part of the problem. The human beings in the mix cause the problems. They (as a team) need a full understanding of the problem domain and the tools they are using.

Using some set of coding guidelines such as MISRA C does not turn an ordinary programmer into one that can be allowed to deal with issues where human life is at risk.

I contend that anyone who believes that the following code is safe and guaranteed to output 5 is not qualified to write guidelines for use of C in safety critical contexts.

```
#include <stdio.h>
int main(){
  int i = 2;
  int j = (i++) + (i++); /* A */
  printf("%d", j);
  return 0;
}
```

A full understanding of C is certainly achievable and anyone working on safety critical C code should have that level of understanding as a pre-requisite.

We need development tools that work as described and generate correct code from our correctly written source code. We need guidelines (preferably ones with tools to enforce them) to avoid problems caused by implicit problems in the language being used. Above all, we need programmers (software engineers if you insist on being grandiose) who fully understand what they are doing and what their code means. A competent programmer will never confuse undefined behaviour with unspecified behaviour.

We need guidelines that prohibit code such as that in line A in the above. However, we also need the programmer to understand why such code is forbidden and that no number of parentheses will fix it.

We also need programmers who do not think that testing will demonstrate that code is safe. All that testing will do is detect some faults, it will not prove that the code is fault free.

I do not need a professional qualification to understand these issues, nor do you. Yet it seems that quite a few people with professional qualifications do not understand them. At that point, it is fair to ask what the value of a professional qualification is.

Fragile Validation

I recently purchased a download version of ZipMagic 9.0. My experience highlights a serious weakness with such purchases and validation.

In order to install the application I have to type in the serial number. Unfortunately despite having been supplied with three different serial numbers, none of them allow me to install the product.

What is worse is that the supplier (Allume) has failed to respond to my emails. Sales decided it was not their problem and Technical support seem to have nothing to say after I supplied them with the information they asked for.

I wonder how many readers have had similar problems with either this product or other ones.

Problem 23

Problems with initialisation have been of concern to those responsible for working on the next version of the C++ Standard. Have a look at the following code and comment on any possible surprises.

```
#include <iostream>
struct X {
  int i;
  X(){}
};
struct Y: struct X{
  int j;
  Y(): X(), j() {}
};
Y y = Y();
int main(){
  std::cout << y.i << std::endl;
  return 1;
}
```

Note that there is more than one problem with the above code.

Problem 22 Revisited

Well the problem is that I have run out of a ready supply of little coding surprises and problems. It is time that you got involved. Please send in at least one coding surprise. If you do not have any then I guess you do not actually do much programming.

The surprise can be in any of the programming languages that are used regularly for application programming (C, C++, C#, Java, Python etc.)

I wrote the above in my last column. Sadly, there has been no response. I cannot believe that none of you has anything to offer so perhaps you just think yours is too trivial for consideration. In my experience, there are no trivial programming surprises.

Cryptic Clues for Numbers

Last Issue’s Clue

One for love too? Sounds like the right day for it!

I thought that one was easy; perhaps the problem was coming up with an alternative clue. 14/02 is Valentines Day (love usually clues a zero, or in cryptic crosswords it clues the letter o.

This Issue’s Clue

Help! Looks like a sailing dinghy. Hawaiian police series number 5. (3 digits)

As always, decide what number is provided by the above clue and then come up with an alternative clue. Email your clue to me at the address below.

*Francis Glassborow (*francis@robinton.demon.co.uk*) is a freelance computer consultant and long-term member of BSI language panels for C, C++ and more recently Java and C#. He is a regular member of the UK's delegations to WG14 and WG21. He is also the author of 'You Can Do It!' and introduction to programming for novices.*

Debuggers Should Go Backwards

Greg Law and Julian Smith <support@undo-software.com>

Introduction

The computer science community has shown a woeful lack of interest in debugging, which given the huge economic cost of debugging is somewhat mysterious. However, there may be signs of “green shoots” in the desert landscape of debugging tools. There are many tools and techniques that offer significant improvement over the trusty `printf`, and over the past year or two there have been some particularly noteworthy developments, including analytical tools such as Valgrind [13], along with a new breed of debuggers: *bidirectional debuggers*. These allow the programmer to step their program backwards as well as forwards, and so are much more helpful than traditional debuggers in taking the programmer to the source of their bug.

A Brief History of Debugging

Debugging is as old as programming itself. Maurice Wilkes said [18] of his experiences writing some of the world’s first programs in 1949:

> As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Despite high-level languages and other paradigm-shifting changes since 1949, most programmers still spend most of their time debugging. In his seminal book *The Mythical Man Month* [1], Frederic Brooks claimed that even the best programmers produce only ten lines of debugged code per day. In the 1995 revised edition he says this alarming statistic is as true as ever. The operative word here is *debugged*. A programmer may write 50 lines of code on the first two hours of Monday morning — the remainder of their week will be spent trying to get those 50 lines to work correctly.

Computing’s Dirty Little Secret

Henry Lieberman of MIT used his guest editorial of the *Communications of the ACM* [10] to talk about what he calls *‘‘the dirty little secret of computer science’*. That month’s publication is subtitled *The Debugging Scandal and What to Do About It*. In his editorial he says:

> What borders on scandal is the fact that the computer science community as a whole has largely ignored the debugging problem. This is inexcusable, considering the vast economic cost of debugging.

If you were programming in 1997, you’ll know that little has changed since Lieberman’s call to arms. Given the enormous economic costs of debugging, and the size of the market this therefore implies, the lack of attention on debugging aids is somewhat puzzling. Sure, there are have been a few interesting projects around debugging over the years, but not much relative to the amount of research on operating systems, languages, or just about anything else. One might suppose that part of the reason is that these are difficult problems to solve, but then language or operating system1 design and implementation isn’t exactly easy. We suspect the relative lack of effort spent on debugging is as much due to the lack of ‘glamour’ associated with it, as it is to do with the problems being difficult ones.

Are Things Getting Better or Worse?

While there has been some recent progress in programming and debugging tools (see sections 2 and 3), two trends in computer science are threatening to offset these advances.

The first is multithreading. More and more modern applications are multithreaded. And with SMP/multicore CPUs becoming mainstream and clock frequencies flat-lining, if we want to exploit the power in future generations of processors its likely our programs will need to become more multithreaded still.

If multithreading is making debugging harder, today’s security concerns are “raising the bar” when it comes to the quality standards that modern software must meet. Yesterday’s pathological/theoretical concerns that were never likely to bite in real life are today’s security vulnerabilities.

Today’s Debugging Tools

This section reviews some of the techniques commonly used to debug today’s code. It is by no means exhaustive; the intent is merely to give examples of the more common techniques.

We classify the different debugging techniques into three groups, listed below.

Undo db

BIDIRECTIONAL DEBUGGER

For C/C++

- Step through programs backwards
- Jump to any point in program history
- Rewind/replay fully deterministically

~~$495~~

$295

+ Free upgrade to v2.0 on release

Free 30-day trial

Free for non-commercial use

http://undo-software.com/

Programmatic Techniques

Here programmers modify or write their program in order to help find the bugs.

- **Print Statements**. This classic is still the most widely used technique. In some senses the technique’s wide use reflects its simplicity and convenience (it’s rare that this technique isn’t an option). But in other ways the fact that print statements remain our number-one way of debugging code is a reflection on the inadequacy of mainstream debugging tools.
- **Assertions**. An invaluable tool; most good programmers use assertions liberally.
- **Language Support**. If you’re lucky enough to be using a ‘safe language’ (e.g. Java or Python) then there are many classes of bugs that just can’t happen (e.g. memory corruption bugs). Sadly, there are still many jobs where these languages just aren’t an option, and one needs to use lower level languages such as C/C++ (or even assembler).
- **Test-suites**. There is no excuse for not having a test-suite for programming projects of even modest size. Not only do they help identify new bugs quickly, if used properly they can be an excellent way of preventing regressions.

Special Case Diagnosis/Analysis Tools

Some of the most interesting developments in debugging in recent years have come in the forms of tools that help programmers ?nd particular classes of errors in their programs. Some notable examples are:

- **Purify**. [7] This tool instruments a running process and finds common memory access violations. The full version is available for Solaris and sold as part of IBM’s Rational tool-suite for several thousand dollars.
- **BoundsChecker**. [3] Plugs into VisualStudio and finds many common errors in programs by instrumenting running programs. In a broad sense this is your answer to Purify if you’re running on Microsoft Windows (although the details are quite different).
- **Valgrind**. [13, 11] Started life as a sort of open source version of Purify, but has grown in to something much more generic. More recent versions of Valgrind allow one to plug several different ‘skins’ into the Valgrind core in order to check for many different classes of bug (e.g. touching unallocated memory, or potential deadlock conditions).
- **Coverity**. [4] A commercial product born out of an interesting research project [5] that extended `gcc` to find common errors at source code level. Before going commercial their tool found over 2,000 bugs in the Linux kernel. Sadly, you can’t buy the Coverity software — their consultants come to your organisation and run their magic on your source code (they don’t advertise their prices, but it is presumably a safe bet that they don’t come cheap!).
- **Code coverage tools**. Code coverage tools such as `gcov` are (in our experience) woefully rarely used. Most programmers are very surprised to see just how little of their code is covered by their test-suite (if they even have one).
- **Hardware protection**. Most modern computer systems have memory management hardware to catch illegal memory access when they happen. Many CPUs have other debugging features, such as the Intel’s debug registers that can generate a debug exception whenever a specific virtual address is accessed. These can be particularly useful when faced with obscure memory corruption bugs.

General-Purpose Debuggers

Special-case tools can be very useful for debugging. However, such tools are only useful if your bug is of a certain well-known kind (e.g. accessing a variable from two threads without protecting it with a mutex, or accessing unallocated memory). If your bug doesn't fit neatly into one of these categories, such tools don't offer any help.

That's one of the reasons why general-purposes debuggers tend to be used more frequently than do the kinds of special-case tools presented in section 2. However, today's general purposes debuggers have a major flaw: they let the programmer single-step their program forwards, but debugging involves thinking backwards. To debug a program is to reason backwards from the point of failure to determine the cause of that failure. On the first page of their book, *The Practice of Programming*[8], Brian Kernighan and Rob Pike give the following advice to programmers when debugging:

> Reason back from the state of the crashed program to determine what could have caused this. Debugging involves backwards reasoning, like solving murder mysteries. Something impossible occurred, and the only solid information is that it really did occur. So we must think backwards from the result to discover the reasons.

Many programmers use debuggers only to examine a program's state, and then reason backwards from here. Most of a debugger's intelligence is geared towards letting the programmer single-step forwards, but that's of limited help. To be really useful, a debugger needs to help the programmer walk through the program's execution in reverse.

Bidirectional Debuggers

Bidirectional debuggers walk the programmer though their program backwards as well as forwards. The ability to step backwards from a fault to diagnose its cause massively eases the burden of debugging — the programmer can now get straight to the source of the bug.

The aforementioned Kernighan and Pike go on to say in their book:

> If it's a hard bug, you'll be making it happen over and over as you track down the problem.

Bidirectional debuggers herald an end to this running a program again and again looking for the source of a bug; instead the programmer can just rewind and play forwards the program, 'honing in' on the source of their bug.

Bidirectional Debuggers for Java

Java programmers have had the benefit of bidirectional debugging for a few years now. The first of these was a research tool by Bill Lewis, known as ODB [9] (the Omniscient Debugger). His research has demonstrated how programmers of varying abilities were able quickly to track down bugs using ODB that otherwise would have taken much longer. At this year's OOPSLA conference, Bill offered $100 to anyone who could present a bug that his debugger could not find — ODB passed the challenge.

There are now commercial bidirectional debuggers for Java: Visicomp's RetroVue [2] and Omnicore's CodeGuide [15] are examples. Both have won many awards and accolades in the Java community.

Bidirectional Debugging of Entire Systems

The GreenHills SuperTrace Probe records all state transitions a computer system makes as it executes, and when used in conjunction with their *TimeMachine* debugger [14] provides programmers with bidirectional debugging of an entire computer system.

Likewise, the recently launched Hindsight [17] debugger from Virtutech allows bidirectional debugging of a complete system by simulating the hardware with software.

Both of these systems are expensive (the GreenHills SuperTrace Probe and TimeMachine debugger retail for $20,000; Virtutech does not have standard pricing for Hindsight), but if you want to debug a complete computer system (usually this means an embedded system or possibly an operating system), then they can be invaluable.

UndoDB: Bidirectional Debugging for C/C++ Programs

UndoDB has recently been released for the GNU/Linux operating system on i386 PCs. It is unique in that it allows bidirectional debugging of a straightforward binary Linux process. No expensive hardware or simulated hardware platform is necessary. Instead, UndoDB takes the form of a wrapper around the widely used GNU debugger, `gdb`[6]. This means that programmers who are familiar with `gdb` will feel right at home using UndoDB.

A Few Simple Commands

`UndoDB` works just like `gdb`, but adds a few new commands which revolutionise how `gdb` can be used. There are the new commands `bnext`, `bstep`, `bnexti`, `bstepi`, `bfinish` and `buntil`. These work exactly like their forwards counterparts `next`, `step`, `nexti`, `stepi`, `finish` and `until`, except that instead of stepping the program forwards one or more instructions, the program is reversed. For example, the step command moves the program forward to the next source line, possibly stepping to the first line of a different function if the current line includes a function call. Its `bstep` counterpart moves the program back one source line, possibly stepping to the last line of a different function if the previous source line included a function call.

See the UndoDB man page [16] for a full description of these commands.

Time

To say UndoDB lets one run programs backwards as well as forwards is not quite accurate. Rather UndoDB allows one to rewind a debugged program to any point in its history, and examine the program's state at that point. In the case of commands such as `bstepi` this means stepping back to the most recent time a particular source line was executed; by using the `bgoton` command the programmer can go back to an arbitrary point in the program's history. From any point in history, the user can issue the normal `gdb` commands such as `continue` or `next` to move forwards, as well as the backwards commands such as `bnext`.

UndoDB measures time using the notion of "simulated nanoseconds". A simulated nanosecond approximates a nanosecond were the program to execute normally, although there is no strict correlation between wall-clock time and simulated nanoseconds. The user can rewind their program almost instantly to any time in its history by passing the requisite simulated nanoseconds argument to the `bgoton` command. The programmer can find out how many simulated nanoseconds have elapsed in the current debugging session using the `bget` command.

Critically, no matter how many times the program is rewound and replayed, each instruction happens at exactly the same simulated nanosecond each time. More generally, the program is entirely deterministic: each time the program is at any given simulated nanosecond, it will be exactly the same state.

Where To Next?

The functionality of the current release of UndoDB is somewhat limited. Most notably, it only works on GNU/Linux on i386 or i686, and debugging of multithreaded programs or programs that use signals is not supported. In recognition that this precludes UndoDB's use with a large number of modern computer programs, anyone who buys a commercial-use license before the end of 2005 will receive a free upgrade to UndoDB version 2.0 as soon as it is released. Version 2.0 will include support for multithreaded programs and programs that use signals. (We feel that bidirectional debugging will be particularly effective when used with multithreaded programs, especially given the way programs can be rewound and replayed many times, all entirely deterministically.)

There are many other features that are planned for releases subsequent to 2.0. This includes support for other architectures (such as x86-64, ARM or MIPS), and other operating systems (such as Microsoft Windows, Apple's OS X, or the various flavours of BSD).

Also planned are advanced debugging features, such as the ability to change state at a point in the program's history and then replay (thus "rewriting history") or to find the most recent time in a program's history that completely arbitrary criteria were met (such as some expression evaluating to true).

Conclusion

Debugging has for too long been computing's dirty little secret. But things are beginning to change. Among a few new debugging technologies to emerge over the past few years, bidirectional debugging in particular promises to revolutionise the way programs are debugged, drastically reducing development times and at the same time improving software quality.

In the short time Java bidirectional debuggers have been available they have already shown impressive results. Of particular relevance to readers of this publication are new tools that bring bidirectional debugging to C and C++ programmers. For programmers who are writing code that controls the computer system directly (i.e. low-level embedded programmers and operating system programmers), the GreenHills TimeMachine debugger [14] and the Simics Hindsight [17] debuggers offer compelling (if expensive) solutions. For programmers of C/C++2 on GNU/Linux,

[concluded at foot of next page]

Test Driven Development of C# User Interfaces

Phran Ryder <Phran@AgileNorth.org.uk>

Introduction

In my last article I discussed the values in the Agile Manifesto and what they mean to mean. There are many practices that can be used to make yourself more agile. Short iterations, the planning game, pair programming, and refactoring are a few of the practices present in eXtreme programming. The practice of most value to me, and the practice that many recommend to use as a starting point, is Test Driven Development – TDD.

Once you have mastered test driven development there is a good chance you, like me, will wonder how you ever developed without it. That is a bold claim, and a fair response would be "Why?". Below, I tell you why I find it difficult, or at least very uncomfortable, to develop without **automated unit tests** – to me it would be like coming to work naked. But first you might want to know what Test Driven Development is. This tutorial should give you a step towards understanding and mastering test driven development - particularly in the arena of C# user interfaces.

What is TDD?

Test Driven Development is an iterative cycle in which you incrementally build the functionality required. Briefly the cycle (or at least my cycle) is:

- Decide what requirement or part of a requirement to satisfy next.
- Write an **automated** test for the requirement.
- Write the code so that and all other tests past.
- Refactor the code so that it is of good design and quality and all the tests still pass.

After completing the tutorial I hope you will see that there are finer grained steps to the cycle, e.g:

- Write test.
- Compile test - test probably won't compile.
- Add skeleton so test compiles.
- Run test - test will fail.
- Use dummy values to that the test passes.
- Add real code so the test pasts.

Why Use TDD?

There are many angles to the advantages of TDD. Here are my reasons:

- **Confidence and progress**. With each little new test that runs you are making progress. And after the short period it takes to run all the test, you know that all the progress you have made so far is still there. Contrast this will manual testing - with each improvement if you wanted the same confidence you would have to manually repeat all the tests. This would take too long. So most people would only test the part they have change, and perhaps a little more, and hope that they don't break anything.
- **Code quality**. When making the test pass, you don't have to worry about code standards, variable names, object structures, readability, re-use, patterns, etc. You only care about getting code that works – doing the simplest thing to make the test pass. When all the tests are running, you can then look for opportunities to improve the design, readability etc of the code knowing that the tests will prove that you haven't broken anything. Without automated unit tests improvements to the design and readability of code require manual re-run of the tests. Most people will shy away from this sticking to their first solution. The result is code that evolves in to a pile of uncontrolled spaghetti.
- **Progress again**. It becomes easier to know where you are up to by putting in a test that fails explaining what you were doing, and it becomes easier to decide what to do next.
- **Good design**. I have found that a natural by-product of TDD is simple useful classes - probably because they are easier to test.
- **Planning**. As each requirement is satisfied you have a measure of how much you have done, how fast you are going, and when you can expect to finish.
- **You know when you are done!** When there are no more tests to write you are finished - time to celebrate.

The Unit Test Harness

Testing code using automated tests requires an automated unit test harness of some form. This tutorial was tested using NUnit 2.1.

Reading the Article

For each block of code that is introduced there are a few interesting discussion points identified thus {1}. I would suggest that these are ignored during the first reading.

The Requirements

The problem is stolen from an article by Bill Wake that appeared in Extreme Programming Installed [1].

We have bibliographic data with author, title, and year of publication. Our goal is to write a system that can search that information for values we specify. A Paper prototype has produced an interface something like Figure 1 on the following page.

The user will enter text to search for and click the Find button. Any document that contains that text in the Author, Title or Year will be displayed.

[continued from previous page]

UndoDB extends the familiar gdb with bidirectional debugging capabilities.

Some have said that bidirectional debugging represents the biggest change to the way we debug software since the first symbolic debuggers appeared many decades ago. We agree.

References

[1] Frederic Brooks. *The Mythical Man Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley, 1995.

[2] Visicomp corp. *RetroVue Java Software Visualization Tool*. `http:// visicomp.com/product/retrovue/ index.html`.

[3] Compuware Corporation. Compuware boundschecker for devpartner. `http://www.compuware. com/products/devpartner/ bounds. htm`.

[4] Coverity Corporation. Breakthrough technology to find catastrohpic flaws in source code. `http://coverity.com/`.

[5] Dawson R. Engler, David Yu Chen, and Andy Chou. 'Bugs as inconsistent behavior: A general approach to inferring errors in systems code' in *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[6] The Free Software Foundation. Gdb: The gnu project debugger. `http://gnu.org/ software/gdb`.

[7] R. Hastings and B. Joyce. 'Purify: Fast detection of memory leaks and access errors' in *Proceedings of the Winter USENIX Conference*, December 1992.

[8] Brian Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley, 1999.

[9] Bill Lewis. *Omniscient debugging*. `http://lambdacs.com/ debugger/debugger.html`.

[10] Henry Lieberman. 'The debugging scandal and what to do about it.' *Communications of the ACM*, April 1997.

[11] Nicholas Nethercote and Julian Seward. 'Valgrind: A program supervision framework' in *Proceedings of the Third Workshop on Runtime Verification* (RV'03), Boulder, Colorado, USA, July 2003.

[12] Rob Pike. *Systems Software Research is Irrelevant*. `http://cm.bell-labs.com/ who/rob/utah2000.ps`, February 2000.

[13] Julian Seward, Nicholas Nethercote, Jeremy Fitzhardinge, et al. Valgrind. `http://www.valgrind.org/`.

[14] Green Hills Software. Green hills timemachine. `http://www.ghs.com/products/ timemachine.html`.

[15] Omnicore Software. Omnicore website. `http://omnicore.com`.

[16] Undo Software. Undodb man page. `http://undo-software.com/ undodb_man`.

[17] Virtutech. Simics hindsight. `http://www.virtutech.com/ products/simics-hindsight.html`.

[18] Maurice Wilkes. Familiar and unfamiliar quotations. `http:// www.norvig.com/quotations.html`.

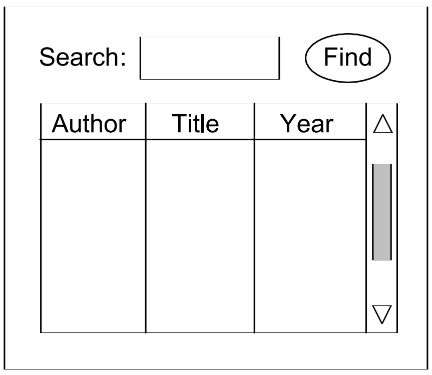


Figure 1: Prototype Interface

The Context

When developing software it is usual for many to produce and test the 'model' first and in isolation. In brief the model is the part of the application that provides the functionality. The model for this UI has already been produced - the interface for the model (placed in `model.cs`) is:

```
using System;
namespace TDD_in_C_Sharp
{
  public interface ISearcher // {2}
  {
    IResult Find(Query q);
  }

  public class Query // {3}
  {
    private string theQuery;
    public Query(String s) { theQuery = s;}
    virtual public String Value // {1}
    {
      get { return theQuery;}
    }
  }

  public interface IResult
  {
    int Count();
    IDocument Item(int i);
  }

  public interface IDocument
  {
    String Author();
    String Title();
    String Year();
  }
}
```

This should compile defining our interface with the rest of the system.

Interesting Points

1. I use virtual on all operations of a concrete class. Unfortunately, the default for C# is that functions aren't virtual. However, I know of very few reasons for having non virtual functions.
2. The model has largely been defined as interfaces rather than classes. There are many potential reasons for this. From a TDD perspective it is because it allows us to use mock objects. When testing a component it is likely that the component will use other components. However, we often don't want to rely on those other components for our tests to run - we'd rather test our component in isolation. The use of interfaces means that in our tests we can substitute the real components with mock ones that we have programmed to supply data and behave as we'd like.
3. Here one of the objects (Query) is so simple that we haven't bothered creating and interface and mocking it (we don't think that is likely to bite us in the future).

Test Widget Existance

The requirements prototyped a screen design. A first test is to ensure the key controls are on the form: a label, a query field, a table and a button. There may be other components on the form but we don't care about them.[1]

In TDD, after establishing what requirement to test, you write a test, and here it is - place in a new file (`TestSearchForm.cs`) but in the same name space. Note: you will have to add a reference to `NUnit.Framework.dll`.

```
using System;
using NUnit.Framework;

namespace TDD_in_C_Sharp
{
  [TestFixture]
  public class UI_Tests
  {
    [Test]
    public void WidgetsPresent()
    {
      SearchForm form = new SearchForm();
      Assertion.AssertNotNull(form.searchLabel);
      Assertion.AssertNotNull(form.queryField);
      Assertion.AssertNotNull(form.findButton);
      Assertion.AssertNotNull(form.resultTable);
    }
  }
}
```

This does not compile because we have not created the form {2}. So create a form as outlined above giving the controls the names used in the tests (e.g. `searchLabel`).

Try to compile. The compilation fails because the tests do not have access to the controls. Make the access internal by modifying the `SearchForm` class as follows {4}:

```
public class SearchForm : System.Windows.Forms.Form
{
  internal System.Windows.Forms.Label searchLabel;
// {4}
  internal System.Windows.Forms.TextBox queryField;
  internal System.Windows.Forms.Button findButton;
  internal
     System.Windows.Forms.DataGrid resultTable;
     //{3}
 ...
```

Run your test with NUnit - it should pass.

Interesting Points

1. We could test the tab order and a few other things (e.g. enablement and visibility) that is simple to do especially if you have a AssertFunction to do it for you.
2. We have created the test without creating the object being tested – defining the names of the widgets on the form. Thus this code will not compile. In practice the two are created at the same time. The order is not that important. The advantage of doing the test first is that you only write enough of the search panel to get it to compile so that you can run the test. A disadvantage is that you can't take advantage of intellisense.
3. In my first implementation I used a list view for the table. I did this because it was easy. A principle of TDD is to **do the simplest thing that could possibly work**. Once you have got your tests running you can consider better ways of working. Later I replaced the list view with the datagrid. Further, you often don't know what the simplest thing is. This means you are going to have to take a diversion to do some investigating - an activity often referred to as a spike.
4. Defining the accessibility to be internal means that the object, in this case controls, are visible to clients within this assembly. If we wanted to move the test code into a separate assembly, we would have to alter the accessibility or access the controls in some other way.

Test Initial Values

Here we are going to test that the controls are given the correct initial values {1}{2}. Add a new test to the test fixture:

```
 [Test]
public void InitialContents()
{
  SearchForm form  = new SearchForm();
  Assertion.AssertEquals("Search:",
form.searchLabel.Text );
  Assertion.AssertEquals("", form.queryField.Text);
  Assertion.AssertEquals("Find",
                         form.findButton.Text);
  Assertion.Assert(
    "Table starts empty",
    ((DataTable)(form.resultTable.DataSource))
     .Rows.Count == 0);
}
```

This code requires a reference to `System.Data`. It also fails because there is no `DataTable` so create one in the form's constructor:

```
public SearchForm()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  this.resultTable.DataSource = new DataTable();
}
```

The cast in the test smells {3}. It is used to get the code working and is necessary because `DataSource` is of type Object – time to refactor to make it beautiful – readable – we'll do this by providing the `DataTable` as a property – hiding the ugliness. Change the test first:

```
[Test]
public void InitialContents()
{
  SearchForm form  = new SearchForm();
  Assertion.AssertEquals("Search:",
                         form.searchLabel.Text );
  Assertion.AssertEquals("", form.queryField.Text);
  Assertion.AssertEquals("Find",
                         form.findButton.Text);
  Assertion.Assert(
    "Table starts empty",
    form.ResultTableAsDataTable().Rows.Count == 0);
}
```

And then add a method to the form to supply the `resultTable` as a `DataTable`:

```
internal DataTable ResultTableAsDataTable()
{
  return  (DataTable) resultTable.DataSource;
}
```

Interesting Points

1. Some people believe that automated unit testing of user interfaces is difficult. The approach taken here is to treat the User Interface as an object and test its behaviour as you would any other.
2. The volume of test that you write in each iteration depends on, amongst other things, confidence in what you are doing and personal style. For example, whether or not controls are present on a form and their initial values could be tested together if your prepared to risk of going faster I normally prefer to take very small steps and build up a consistent momentum.
3. The concept of foul smelling (offensive) code is an interesting one. It might be hard to read, hard to write or something else might be wrong with it. Either way your instinct tells you that it can be improved and made to smell good. Improving bad smelling code is part of the process of refactoring. That is, improving the code to make is easier to read, easier to extend, easier to maintain, and to remove duplication. Well written code just feels good!

Connect Search Form to a Searcher

So far we have created a form for displaying the search result, and checked its initial values. Now we want to attach the Form to objects and test the form's behaviour to be tested in more detail. First we will test the forms ability to attach to a searcher - any object implements the `ISearcher` interface. For our tests we will create an object that implements the `ISearcher` interface – `TestSearcher` {1}.

As ever we'll start by writing a test describing how we expect to be able to set and get the Form's Searcher.

```
[Test]
public void SearcherSetup()
{
  ISearcher s = new TestSearcher();
  SearchForm form = new SearchForm();
  Assertion.Assert ("Searcher not set",
                    form.Searcher != s);
  form.Searcher = s;
  Assertion.Assert("Searcher now set",
                   form.Searcher == s); // [3]
}
```

This won't compile because we don't yet have a `TestSearcher` and form doesn't have a Searcher property. So we implement a Skeleton `TestSearcher`. This in turn requires an `IResult`. This in turn requires an `IDocument`. So we'll implement skeletons for all of these. We'll create them in a separate file (`mockmodel.cs`) to keep the mock components separate:

```
public class TestSearcher: ISearcher
{
  public virtual IResult Find(Query q)
  {
    int count = 0;
    try {count = Convert.ToInt32(q.Value);}
    catch (Exception ignored) {}

    return new TestResult(count);
  }

  public virtual Query makeQuery(String s)
  {
    return new Query(s);
  }
}
```

And add a `Searcher` property to our search form [2]:

```
private ISearcher mySearcher;
public virtual ISearcher Searcher
{
  get { return mySearcher; }
  set {mySearcher = value; }
}
```

The test now runs and passes. So we are able to attach a searcher to the form but the form doesn't do much.

Interesting Points

1. We could use a proper `ISearcher`. However, we wouldn't have control of the behaviour of the searcher and we don't want to rely on it for our tests to pass. Instead we create an implementation of the interface (a mock object) solely for our testing purposes.
2. We want to the ability to associate a Searcher (`ISearch`) with our form. There are two ways of doing it. The Searcher could be supplied as the object is created and/or after the fact and thus allowing it to be changed. I have chosen to allow it to be changed as it makes for an easier tutorial - as ever we do the simplest thing first.
3. The test uses equality. Here equality uses the identity of the object, i.e. if they are the same object they are equal. An alternative is to use some definition based on the contents. E.g. is they contain the same particular values they are considered equal.

Mock the Searcher

The next thing to do is populate the form from a Searcher - an Object that supplies the `ISearcher` interface. So we'll create a test searcher that implements `ISearcher`. The string used in making a search will be an integer that we'll use to tell us how many items to return. We'll name the items `a0` (for first author), `t1` (second title), etc.

Before implementing `TestSearcher` we will, of course, write a test to test our `TestSearcher`.

```
[Test]
public void Searcher()
{
Assertion.AssertEquals(new Query("1").Value, "1");
// {1}
  IDocument doc = new TestDocument(1);
  Assertion.AssertEquals("y1", doc.Year());

  IResult result = new TestResult(2);
  Assertion.Assert(result.Count() == 2);
  Assertion.AssertEquals("a0",
                       result.Item(0).Author());
  TestSearcher searcher = new TestSearcher();
  result = searcher.Find(searcher.makeQuery("2"));
  Assertion.Assert("Result has 2 items",
                 result.Count() == 2);
  Assertion.AssertEquals("y1",
                       result.Item(1).Year());
}
```

This fails to compile because `TestDocument` and `TestResult` don't have a constructor that takes a single parameter (The integer will be used to name the documents). And the classes (`TestSeacher TestResult` and `TestDocument`) have no implementation. So we'll provide these:

```
public class TestSearcher: ISearcher
{
  public virtual IResult Find(Query q)
  {
    int count = 0;
    try {count = Convert.ToInt32(q.Value);}
    catch (Exception ignored) {}
    return new TestResult(count);
  }
  public virtual Query makeQuery(String s)
  {
    return new Query(s);
  }
}

public class TestResult: IResult
{
  int count;
  public TestResult(int n) {count = n;}
  public virtual int Count() {return count;}
  public virtual IDocument Item(int i)
     {return new TestDocument(i);}
}

public class TestDocument: IDocument
{
  int count;
  public TestDocument(int n) {count = n;}
  public virtual String Author()
     {return "a" + count;}
  public virtual String Title()
     {return "t" + count;}
  public virtual String Year() {return "y" + count;}
}
```

Run the test and it should pass.

Now we can test the display of the search results. When testing objects that can take a number of values it is usually a good idea to test that it works with 0,1 and lots. Let's start with zero.

Interesting Points

1. Several of the tests use `AssertEquals()` rather than `Assert()`. This is because the former can tell you what values weren't equal when the test fails.

Test 0

Here is the code for the test where the number of items found by the test is zero.

```
[Test]
public void InitialContents()
{
  SearchForm form  = new SearchForm();
  Assertion.AssertEquals("Search:",
                       form.searchLabel.Text );
  Assertion.AssertEquals("", form.queryField.Text);
  Assertion.AssertEquals("Find",
                       form.findButton.Text);
  Assertion.Assert("Table starts empty",
                 form.ResultTableAsDataTable()
                 .Rows.Count == 0); // [1]
}
```

This doesn't compile because we don't have a `findButton_Click ()` method on the form. So add an empty one.

```
private void findButton_Click(object sender,
System.EventArgs e)
{
}
```

This doesn't compile either, `findButton_Click` is not visible. So declare it as internal so that the test class can see it.... And the tests pass.

Interesting Points

1. For the initial state there is a choice of having a table that contains no rows or no table at all. We have used the former.

Test 1

```
[Test]
public void InitialContents()
{
  SearchForm form  = new SearchForm();
  Assertion.AssertEquals("Search:",
                       form.searchLabel.Text );
  Assertion.AssertEquals("", form.queryField.Text);
  Assertion.AssertEquals("Find",
                       form.findButton.Text);
  Assertion.Assert("Table starts empty",
     form.ResultTableAsDataTable().Rows.Count == 0);
}
```

The test fails because `findButton_Click()` has no implementation.

When the button is clicked, the string in the text field is translated into a query and the searcher finds a result for display in the table. However, there is a type mis-match: the Searcher gives us an `IResult`, but the display table needs a `DataSource`.

Thus, we need something to adapt the `IResult` output to the `DataSource` needed. To achieve this we will create an Adapter class that is given an `IResult` and provides the `DataTable` interface. {1}

At this point I was tempted to write a test for the adapter but, doing the simplest possible thing, the adapter only has to do enough to satisfy the buttons needs so instead we'll start by implementing the button as if the adapter existed. In the form:

```
internal void
findButton_Click(object sender, System.EventArgs e)
{
  Query q = new Query(queryField.Text);
  resultTable.DataSource = new DataTableAdapter
     (Searcher.Find(q));
}
```

And to make it compile we'll stub the class placed class in its own file (`DataTableAdaptor.cs`).

```
using System;
using System.Data;
namespace TDD_in_C_Sharp
```

```
{
  public class DataTableAdapter :  DataTable
  {
    public DataTableAdapter(IResult theResult)
    {
    }
  }
}
```

The test fails as we need to implement the constructor:

```
public DataTableAdapter(IResult theResult)
{
  this.Columns.Add("Author");
  this.Columns.Add("Title");
  this.Columns.Add("Year");
  for (int index=0; index < theResult.Count();
       index++)
  {
    IDocument doc = theResult.Item(index);
    DataRow newRow = this.NewRow();
    newRow["Author"] =
       theResult.Item(index).Author();
    newRow["Title"] = theResult.Item(index).Title();
    newRow["Year"] = theResult.Item(index).Year();
    this.Rows.Add(newRow);
  }
}
```

Test1 now passes. And so does test N:

```
[Test]
public void TestN()
{
  ISearcher searcher = new TestSearcher();
  SearchForm form = new SearchForm();
  form.Searcher = new TestSearcher();
  form.queryField.Text="5";
  form.findButton_Click(form.findButton ,
     new System.EventArgs ());
  Assertion.Assert("5-row result",
     form.ResultTableAsDataTable().Rows.Count == 5);
  Assertion.AssertEquals("a0",
     form.ResultTableAsDataTable().Rows[0][0] );
  Assertion.AssertEquals("t3",
     form.ResultTableAsDataTable().Rows[3][1] );
  Assertion.AssertEquals("y4",
     form.ResultTableAsDataTable().Rows[4][2] );
}
```

Interesting Points

1. Adapter is a design pattern described in more detail in *Design Patterns* [2]

Test For Left Overs

After running one search we want to make sure that the results aren't present in any subsequent search.

```
public void LeftOvers()
{
  ISearcher searcher = new TestSearcher();
  SearchForm form = new SearchForm();
  form.Searcher = new TestSearcher();
  form.queryField.Text="5";
  form.findButton_Click(form.findButton,
     new System.EventArgs());
Assertion.Assert(form.ResultTableAsDataTable()
     .Rows.Count == 5);
  form.queryField.Text="3";
  form.findButton_Click(form.findButton ,
     new System.EventArgs());

Assertion.Assert(form.ResultTableAsDataTable()
     .Rows.Count == 3);
}
```

This test passes.

Test the User Interface

Some times you might want to test some aspects of the appearance of the user interface. For example, the relative position of controls or overlapping controls. I haven't yet found it necessary considering that aspect to be part of functional testing. It is, however, useful to test the tab order:

```
[Test]
public void TabOrder()
{
  SearchForm form = new SearchForm();
  NUnit.Forms.FormAssertions.AssertTabOrder(form,
    "queryField,findButton,resultTable");
}
```

Refactoring - Improving the Design

We have got our application working - GREAT! Quality developers would now work hard to improve the design - if possible.

The design metaphor used mirrored that used by Bill Wake in his original article. It has given a basis for TDD of C# (and .net) user interfaces. However, it is frequently stated that UI layer should be very thin. What does this mean? Is the design so far thin enough?

The Humble dialog [3] article argues that the behaviour of the dialog (or Form) should be placed in an intermediate 'Smart' object. The Smart object knows what should be displayed and tells the form to display it. It knows how to respond to events such as button clicks and the form should ask it to take the appropriate response to a click. The form then becomes a set of properties defining what to display, and a set of event handlers that delegate to the Smart object. Testing involves driving this smart object with Mock UI objects.

Martin Fowler suggests a similar approach in his article *Model View Presenter*. Back in 1994 Phran Ryder, in his MSc Thesis, used a similar approach in which he referred to the smart object as the view-controller.

What Are the Pros and Cons of this Approach?

Cons

1. It makes it more difficult to take advantage of the ADO set of objects. Microsoft's ADO controls and classes can make it easier to rapidly create an application. Such applications are to a great extent the Smart GUI described in Domain Driven Design [4]
2. When the application is simple why go to the trouble of adding another layer.

Pros

1. The idea that forms should be properties and delegating event handlers makes it easy to see whether the form is doing anything it shouldn't. It is be important that the form only contains behaviour that is relevant to its responsibilities.
2. Decouples the form and the model. Without the smart object, the form could be coupled to many parts of its model. The smart object would deal with all that coupling. Splitting the form into two would be easy. But if you wanted to split the form in two you could create the intermediate object at that point.

So have we got anything in our form that should not be there?
1. We have a `mySearcher` bound to the model – the form need not know about searchers.
2. We have code binding the data source to an ADO object in this case a dataTable.
3. We have a Find button code to perform the query and bind the ADO result to the data source – this code could be moved to the smart object.
4. All the initialisation code is created by .net. Can't do much about this.

So there are a few opportunities for improvement....

References

[1] *Extreme Programming Installed*. Ron Jeffries, Ann Anderson, Chet Hendrickson. Addison Wesley 2001. ISBN: 0-201-70842-6

[2] *Design Patterns:Elements of Reusable Object-Oriented Software*. Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley 1995. ISBN: 0 201 63361 2

[3] `http://www.objectmentor.com/resources/articles/ThHumbleDialog.pdf`

[4] *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Eric Evans. Addison-Wesley 2004 .ISBN: 0 321 12521 5

A Reflection on Defensive Programming

Roger Orr <rogero@howzatt.demon.co.uk>

In his last editorial Paul put out a plea for articles on four specific subjects, one of which was 'Defensive Programming'. This got me reflecting first of all about the basic question: "What is defensive programming?"

I started finding an answer with Wikipedia [1] which began *Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software* and although the whole article was a reasonable summary I felt more could be said.

What is Defensive Programming Defending Against?

There are a number of different things covered by this phrase. In the first instance, a defensive programmer is protecting themselves [2] against their own mistakes. All programmers are optimists [3] and so each one of us tends to ignore the reality that every line of code we write has a probability of being wrong. The probability may be lower for some of us than for others (we all think that) but it is never zero, and the consequence is that the bigger the program is, the more certain it is to contain at least some incorrect code.

Most programs are not written by a single programmer, and so defensive programming also means defending your code against other people's misuse of it. In an ideal world it should be obvious to others how to use the code I've written and this reduces the need for other forms of defence. However even where other people do not find the code obvious to use we should try to make it easy for them to understand what they have done wrong and how to get it right. This ideal seems can be quite hard to achieve in practice, but it is worth reminding ourselves of it from time to time!

Once written our programs will, we hope, be used. So we need to be defensive against users, possibly untrained and often not terribly computer literate, whose understanding of the purpose and use of our programs might be rather different from our own. Additionally, the program will be running on another person's machine and this could be a machine with a very different hardware configuration, resources, software versions, etc.

Again we hope our programs will continue to be used, perhaps for some to come. So we have to prepare for change and therefore another part of defensive programming is trying to make our programs safe against possible future changes; but this is hard. *It's tough to make predictions, especially about the future.* (Yogi Berra.) A simple example from the world of SQL is avoiding the use of `SELECT * FROM ...` as this usually breaks when the database schema changes.

Finally one of the ever present needs in recent times is defence against malicious crackers, particularly but not exclusively on the Internet. It is part of being a computing professional in this day and age to consider the security requirements of every program that we write and then to make informed judgments about the correct level of security audit required.

How Does "Defensive" Code Behave?

The ultimate goal of writing defensively is simply to prevent problems occurring. An example from everyday life is that of an electric plug and socket with two prongs. A defensive design makes the plug slightly asymmetric so it can only be put together correctly.

In practice however not all problems can be prevented; in which case defensive programming means trying to make any problems that do occur localised and of low cost to the users of the program. A less defensive approach to the plug and socket in the previous paragraph is to protect the user (and the equipment) from any fatal consequences of inserting the plug incorrectly. As an example from the world of IT, I used a full screen editor in the 80s that never ever lost any work done in an edit session (except for the actual screenful being typed in) even when the mainframe we were using crashed. That was a defensively written program - and some modern word processors could do with emulating this behaviour!

A secondary aim is to try and make any problems that do occur as easy to resolve as possible. When a problem does occur defensive programming tries to ensure that enough information is collected and logged when unexpected situations are encountered so that the actual problem can quickly be identified and resolved. For example, this may mean writing code that stops the program quickly on an error rather than attempting to carry on with bad data.

If you are an active programmer consider the current programs you are working on. With the possible exception of Tex [4] all substantial programs contain bugs so the chances are that you are writing - and will ship - buggy code. The defensive programmer's approach is to try and ensure that the bugs do as little damage as possible and that they can be reported and resolved rapidly.

Why Are Programs Not Written Defensively?

I remarked above that "All programmers are optimists". The first step is realising that your own code needs to be written defensively. Depending on the individual programmer concerned this can take a long time to really sink in. Even after 20 years in IT I am still surprised by some of the stupid little bugs I introduce.

However, even when we recognise the need for it, there are several difficulties with writing defensive code. First of all, it is hard to write defensively since, almost by definition, you are trying to face the unexpected. One of the things that comes with improved levels of skill is better awareness of the sorts of things that can go wrong with programs and the techniques that can be used to obviate them. Every bug you find gives you an opportunity to improve your awareness of similar potential bugs in the future.

Secondly, many of the techniques that make programs more defensive add code (not all do - in particular defensive design decisions may have no impact on eventual code size). This extra code takes time to write and may have a detrimental effect on performance. However there are some factors standing against these downsides. Firstly, as I'm sure most readers of C Vu already believe, fixing problems usually becomes more expensive the later in the development process that they are discovered and so the cost of writing this extra code may well be recouped in the reduction of time spent fixing bugs later on. Secondly it is almost always better to get the right answer slowly than the wrong answer quickly; as most commentators on optimisation point out it is generally easier to make a slow program faster than it is to make an incorrect program correct.

A final reason may be lack of good examples. Have you noticed how many times in articles about computer programming the code shown comes with the comment "Error handling omitted for clarity"? Although this may be a true excuse the consequence is that we rarely see good examples of defensive code. In fact, I suppose you could say we see more of the opposite of 'defensive' code - 'offensive' code!

What Can We Do?

A good example of defensive programming comes from *How to Hunt Elephants* [4].

COMPUTER SCIENTISTS hunt elephants by exercising Algorithm A:
1. Go to Africa
2. Start at the Cape of Good hope.
3. Work northward in an orderly manner, traversing the continent alternately east and west.
4. During each traverse pass:
   Catch each animal seen.
   Compare each animal caught to a known elephant.
   Stop when a match is detected.

EXPERIENCED COMPUTER PROGRAMMERS modify Algorithm A by placing a known elephant in Cairo to ensure that the algorithm will terminate.

This amusing example makes a serious point about writing code that copes with the unexpected. The experienced programmer in the story has already faced the possibility that his search finds no elephants before even starting the search! As we write code, we should be conscious of the assumptions that we are making (examples might be 'this file exists' or 'this security permission is held by this user') and make a decision about how to cope when our assumption is not valid before we even execute the program. So for example a parallel to placing an elephant in Cairo might be providing a default setting if a configuration file is unreadable.

Another facet of this preparing for the worst comes into play when writing code for other people to use. We should ask ourselves how an interface might be abused (either accidentally or maliciously) and how we might prevent such abuse. This might be as simple as changing the arguments to a method call. For example Herb Sutter recently pointed out

[concluded at foot of next page]

Let's Do C# and MySQL – Part 2 – A Beginning

Paul F. Johnson <paul@all-the-johnsons.co.uk>

Last Time...

Last time, I didn't even touch on any form of programming, but I did cover what MySQL is, how to set up a database and set up some tables which we will make use of over time. It was quite a useful exercise for me as I was able to consolidate material which for some reason didn't quite gel when I was originally taught it.

I did not cover inserting data, deleting data or even interrogation of data held within the database. There was a reason for this – it gives me a basis for introducing the MySQL API and how to code it up. But first, a health warning.

ByteFX.Data.MySQL and MySQL.Data

Mono ships with the ByteFX library which includes the definitions required for working with MySQL. It doesn't ship the version from the MySQL website, but that can certainly be compiled and is very easy to get running with the current release of mono. From what I have been told, VisualStudio.NET also ships with ByteFX so I will be using that instead of the version from the MySQL website. It shouldn't really make much of a difference as to which one is used as they are following the same API1. The names of the methods used to call a particular activity may not be the same though.

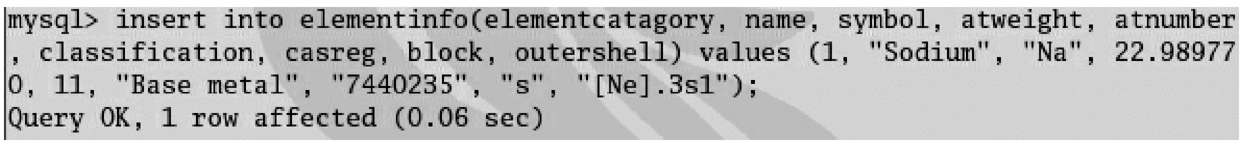
The best way though is that before the programming begins, that you understand how to use the MySQL interface to insert, delete and query data. After that, everything will fall into place.

Inserting Data via the MySQL Interface

This is very simple indeed and follows the following form

```
insert into <tablename>(<table_elements>)
values(<values_to_be_inserted>);
```

In the case of the database started from the last issue (and which is still available from my home website), I did the following:



```
mysql> insert into elementinfo(elementcatagory, name, symbol, atweight, atnumber
, classification, casreg, block, outershell) values (1, "Sodium", "Na", 22.98977
0, 11, "Base metal", "7440235", "s", "[Ne].3s1");
Query OK, 1 row affected (0.06 sec)
```

As I say, nothing to it.

I have not updated `id` – this is set as `auto_increment`. This means though that it may not be the number you would expect it to be. It is now down as being 108 (the database I provided has the last id number as 107). The problem is that I actually am using `id` incorrectly (and quite deliberately so). I had planned to create the database and tables in one go with `id` also being the element's number. While this isn't that important in the bigger scheme of things, it does mean that I have to enter the data in the correct elemental order with any snafu's requiring to be deleted.

`id` can set manually in the same way as any element within the table. It is, after all, nothing special.

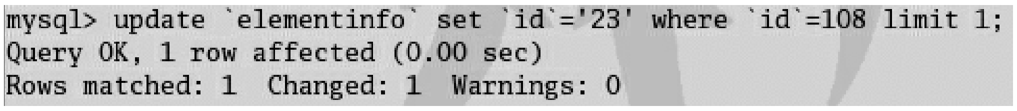
Editing Data

It is possible to edit the `id` datafield without resorting to violence as well (which is quite handy as it means that if instead of a table of periodic elements, you had something like a stock list, you can dynamically alter the number of items in stock quickly and simply).

To edit a parameter

```
update `<tablename>`
set `<element_name>` ='<new_value>'
where `<element_name>`=<old_value> limit 1;
```

or for my example



```
mysql> update `elementinfo` set `id`='23' where `id`=108 limit 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Deleting Data

Deleting data is an operation that everyone will need to do at some point or other with a database. An entry may be so messed up that it is not worth issuing a number of update commands to the server. It is simpler to just delete the row and re-enter the data.

To remove a row

```
delete from `<table_name>`
where `<primary_key>`=<value> limit 1;
```

[continued from previous page]

that the C++ standard library call **`std::transform`** takes three iterators and so it is all too easy to invoke it wrongly like this:

```
transform( in.begin(), in2.begin(), in2.end(),
           out ); // 1: oops
```

But if we had made the user write:

```
transform( range(in.begin(), in.end()),
           in2.begin(), out ); // 2: right
```

then the possibility of making a mistake would be drastically reduced as the function arguments themselves make the use of call clearer.

A bigger problem with some interfaces is an implicit ordering of method calls that seemed obvious to the writer but is less obvious to the user. For example in objects with state - such as TCP sockets - it is often not documented which actions are valid in which state(s). To make things harder any restrictions are often not enforced and method calls may just fail silently if they are made when the object is in the incorrect state.

We should reflect on our own bug writing technique. What sort of bugs do we ourselves write most often? If we can identify something we get wrong habitually we can then look at improving our technique to try and prevent such cases in the future. For example, **`strcpy`** in 'C' often causes people problems with buffer overruns so using alternatives, whether **`strncpy`** or **`std::string`** in C++, can prevent problems before they occur.

When we try writing defensively part of the work is examining what should be checked and deciding how certain the check must be. For example, if we are writing code that passes input from the user into a database query we must be aware of the possibility of 'hijacking' the query by embedding escape characters in the input field. This might be either accidental or malicious and our judgement of the degree of checking required must take account the risks involved: a public Web site usually needs better checking than an intranet-based team resource.

Summary

Defensiveness in programming is not a single issue subject but covers defence against a range of people and situations. Different situations will have different tradeoffs but it is our responsibility to improve our awareness of the issues and to give ourselves feedback so we can make the code we write better defended.

Roger Orr

Bibliography and Notes

[1] Wikipedia
`http://en.wikipedia.org/wiki/Defensive_programming`

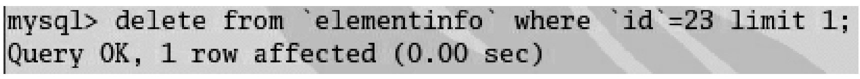
[2] The grammatically correct alternative "him or herself" is, in my opinion, just too ugly.

[3] Fred Brooks "The Mythical Man Month"

[4] Tex reportedly bug free.
`http://web.mit.edu/klund/www/urk/texvword.html`

[5] From *A Random Walk in Science*, also in many places on the Internet.

And again, from my database:



```
mysql> delete from `elementinfo` where `id`=23 limit 1;
Query OK, 1 row affected (0.00 sec)
```

Deleting a Table or Database

Of course, there may be times when an entire table is so completely mucked up that the only real way to resolve the problem is to delete the table, or in extreme cases, the database. This is performed using the `drop` command. Once this command is issued, there is no going back. The table or database is gone. No more. Made like an old oak table and vanished.

```
drop table `<table_name>`
```

or

```
drop database `<database_name>`
```

Finding Data from the Table

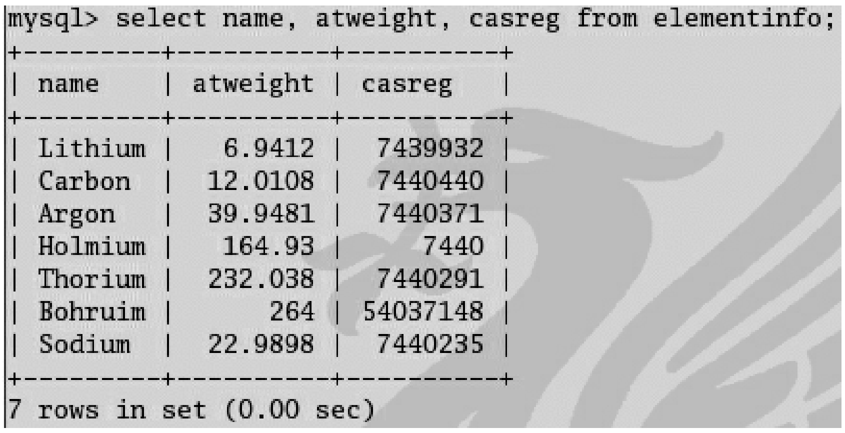
Once data is in the table, you will someday wish to make use of it. Again, not a difficult task and a task which is very easy to modify

```
select <table_elements> from <table_name>;
```

Say I wish to see all elements, their atomic weights and the CAS registry number, all I do is issue

```
select name, atweight, casreg
from elementinfo;
```

which will give



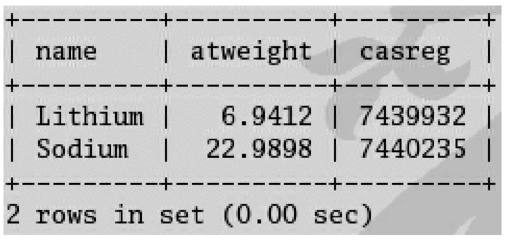
```
mysql> select name, atweight, casreg from elementinfo;
+---------+----------+----------+
| name    | atweight | casreg   |
+---------+----------+----------+
| Lithium |   6.9412 |  7439932 |
| Carbon  |  12.0108 |  7440440 |
| Argon   |  39.9481 |  7440371 |
| Holmium |   164.93 |     7440 |
| Thorium |  232.038 |  7440291 |
| Bohruim |      264 | 54037148 |
| Sodium  |  22.9898 |  7440235 |
+---------+----------+----------+
7 rows in set (0.00 sec)
```

While the database is small, that is useful. However, there are 120 elements in the periodic table – that will mean that the data presented will certainly fill more than the server window. The search command is flexible and allows the user to specify to print (say) all base metal elements. This is achieved by using `where` within the `select` command sequence.

```
select <element_names> from <table_name>
where (...);
```

For me to select just the base metal elements, I would use

```
select name, atweight, casreg from elementinfo
where (classification = "Base Metal");
```



```
+---------+----------+---------+
| name    | atweight | casreg  |
+---------+----------+---------+
| Lithium |   6.9412 | 7439932 |
| Sodium  |  22.9898 | 7440235 |
+---------+----------+---------+
2 rows in set (0.00 sec)
```

The `where` modified can be used in the same was as an `if` conditional can in programming. You can include `or` and `and` and even sort the output!

```
select name, atweight, casreg from elementinfo
where (classification="Base Metal" and
atnumber < 12); (See figure 1)

select name, atweight, casreg from elementinfo
where (classification="Base Metal" or
block="f"); (See figure 2)

select name, atweight, casreg from elementinfo
where ((classification="Base Metal" and
atnumber < 10) or block="f") order by "id";
(See figure 3)
```

Powerful stuff!

As you can also see from the times, retrieval is almost instant. From tests I've performed, even with absolutely huge databases (in excess of 2/3rd of a million rows in a single table with the database containing another 6 tables, each with around ¼ million rows), retrieval from the database still gives very small times – even with a large conditional in the `where` conditional.

We've now covered the essentials that we will later use for our database. We have gone through the data types, how to create, modify and delete as well as interrogate our database and tables. Let's get on with some coding!

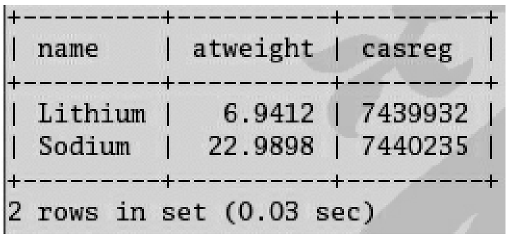
Using C# with MySQL

While it would be simple for me to hack out the source using C and MySQL, I have deliberately chosen C# for two reasons

1. With mono and its implementation of `System.Windows.Forms`, I am able to create a single executable which will work on any machine with either the mono runtime or .NET 1.1 installed. This covers the majority of computers running. The final binary does not need recompiling to run on another platform.
2. I like C#

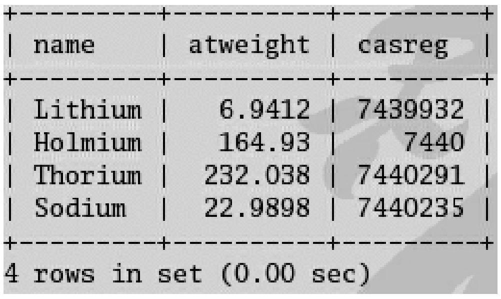
Alright, the second isn't a reason as such, but hey, who is writing this?

What I'll cover for the rest of this article is connecting to the server, reading data, inserting data and editing data. It will not be that hard.



```
+---------+----------+---------+
| name    | atweight | casreg  |
+---------+----------+---------+
| Lithium |   6.9412 | 7439932 |
| Sodium  |  22.9898 | 7440235 |
+---------+----------+---------+
2 rows in set (0.03 sec)
```

Figure 1

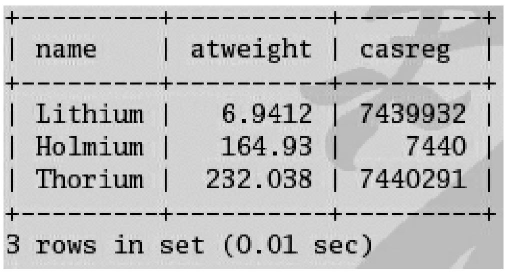


```
+---------+----------+---------+
| name    | atweight | casreg  |
+---------+----------+---------+
| Lithium |   6.9412 | 7439932 |
| Holmium |   164.93 |    7440 |
| Thorium |  232.038 | 7440291 |
| Sodium  |  22.9898 | 7440235 |
+---------+----------+---------+
4 rows in set (0.00 sec)
```

Figure 2

The Basic Method

As these are stand-alone examples, they will all follow the same method. There is actually nothing difficult about any of what we are about to do, essentially, we will be using a small number of methods to perform the tasks we need.



```
+---------+----------+---------+
| name    | atweight | casreg  |
+---------+----------+---------+
| Lithium |   6.9412 | 7439932 |
| Holmium |   164.93 |    7440 |
| Thorium |  232.038 | 7440291 |
+---------+----------+---------+
3 rows in set (0.01 sec)
```

Figure 3

What makes this even simpler, is all that we need to do is define a string which contains what we want to do and then pass that to the C# controller. Who could ask for anything more?

As you can see in the flowchart (figure 4) I will test to see if an error has occurred at the connection stage. I'll also be testing after any connection or process is passed to the database. It is essential to do this as the application will complain violently if you attempt to insert, delete or interrogate a table when the query to the database fails.

There are two ways to catch the error. The simplest is that if the query fails, a non-zero is returned, so using `if (!query) ...` should suffice. The second method is to use `try` / `catch` (much in the same way as you would for C++). Below is the simplest method to connect to the MySQL server

```
using System;
using System.Data;
using ByteFX.Data.MySqlClient;
public class Test
```

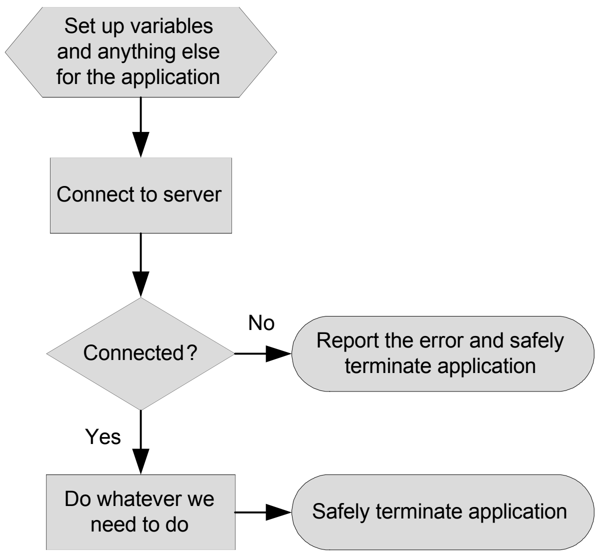

Set up variables and anything else for the application
Connect to server
Connected?
No
Report the error and safely terminate application
Yes
Do whatever we need to do
Safely terminate application

Figure 4

```
{
  public static void Main(string [] args)
  {
    string connectionString = "Server=localhost;"+
       "Database=theelements";+
       "User ID=paul;"+
       "Password=paulpassword;";
    IDbConnection dbcon;
    dbcon = new MySqlConnection(connectionString);
    dbcon.Open();
    if (!dbcon)
    {
      Console.WriteLine("Unable to connect to the
                        MySQL server");
      exit(1);
    }
    else
      Console.WriteLine("Connection made");
    dbcon.Close();
   dbcon = null;
  }
}
```

`IDbConnection dbcon;` creates an instance of the C# database connection class. At this point, this can be a connection to any number of different database types. `dbcon.Open()` opens the connection to the database which by virtue of the line before it, is given as a MySQL connection.

This can be compiled by issuing (`<compiler>` can be either `csc` or `mcs` here) `<compiler> mysql.cs -r:System.Data -r:ByteFX.Data` which will create an executable called `mysql.exe`.

Test the binary. Assuming everything went well, you will have seen the line `Connection made` before the application terminated. If it didn't work, you will need to ensure that your listing matches mine and that the MySQL server is running (it's caught me out a couple of times before now).

We can now connect to any MySQL database! Let's do something a bit more taxing – reading data from the database (which is defined in the `commandString` line).

This creates another instance of `IDbCommand`, this time it is an instance of `dbcon.CreateCommand()`. The command to be used is created and `dbcmd.CommandText` is set to equal the command. The command is then acted upon. `CommandText` is able to either get or set from the database by the type passed to it. If it is a `string` (as it is here), the command is a `get` (the information is retrieved). If it is passed a `void`, the command is set. Again, test using `if (!(dbcmd.CommandText = sqlcommand)) ...`

I need to explain `IDbCommand` here as this interface represents a database command. To execute a command against a database, there has to be an open connection and a properly configured command object for the database type. These "command objects" can be created using a constructor, but it is far simpler to use the `CreateCommand` method. This returns a command object of the correct type for the database type and even goes as far as to configure the command object with the basic information obtained in the connection.

Property	Description
CommandText	A string containing the text of the SQL command to execute OR the name of a stored procedure (I'll cover these at a later time). This must be compatible with the value specified in the CommandType property
CommandTimeout	An int which sets a time (in seconds) to wait for the command to return before timing out and raising an exception. The default is 30 seconds and the parameter is commonly omitted.
CommandType	A value of the `System.Data.CommandType` enumeration that specifies the type of command represented by the command object. For most data providers, there are 3 valid values: 1. StoredProcedure. For use when you wish to execute a stored procedure 2. Text. This is for executing a SQL text command 3. TableDirect. This will return the entire contents of one or more tables. The default is Text.
Connection	An IDbConnection instance that provides the connection to the database which you want to act upon. If the command is created using the `IDbConnection.CreateCommand` method, the property with be automatically be set to the IDbConnection instance from which the command was created.
Parameters	A `System.Data.IDataParameterCollection` instance containing the set of parameters to substitute into the command.
Transaction	A `System.Data.IDbTransaction` instance representing the transaction into which to enlist the command.

Table 1: Common command object properties

The command object should have the properties in table 1 set; these are common to all command implementations
You will meet these later in the article.

Next is to read the data. This is performed using `IDataReader`. This provides a means of reading one or more forward-only streams of result sets obtained by executing a command at a data source and has quite a number of classes. The important aspect to this though is that it is a sequential reader (in other words, it grabs one line at a time until the end of the read process).

```
sqlcommand = "select elementname, atweight, casreg
from elementinfo";
dbcmd.CommandText = sqlcommand;
IDataReader reader = dbcmd.ExecuteReader();
while (reader.Read())
{
  string element = (string) reader["elementname"];
  float atweight = (float) reader["atweight"];
  long cas = (long) reader["casreg"];
  Console.WriteLine("Element - " + element + ",
     weight "+ atweight + ", CAS : "+ cas;
}
```

Handy Hint for Linux Users

Normally, you will need to test the application using `mono<appname>.exe`. A simpler way to get around this is to edit your `.bashrc` to include the line:

```
echo `:CLR:M::MZ::/usr/bin/mono:' >
/proc/sys/fs/binfmt_misc/register
```

This will only work if your kernel has the use misc. binary formats installed - you will know this if you have a directory called `/proc/sys/fs/binfmt_misc`.

Again, a command is passed to `CommandText`. However, this time the command needs something to do the reading for it. The `IDataReader` class performs such a task and it is easy to see what is happening with the interesting part being two fold. The first is the `IDataReader` is positioned before the first element of the database by default, so a `Read()` has to be performed before anything useful comes out. The second is that as the database could contain any number of field types, the `Read()` returns everything as `void*` which then needs to be recast to the correct type for the data.

To terminate the application, `reader` and `dbcon` have to be closed and before `dbcon` is closed, `dbcmd` has to be disposed of.

```
Reader.Close();
reader = null;
dbcmd.Dispose();
dbcmd = null;
dbcon.Close();
dbcon = null;
```

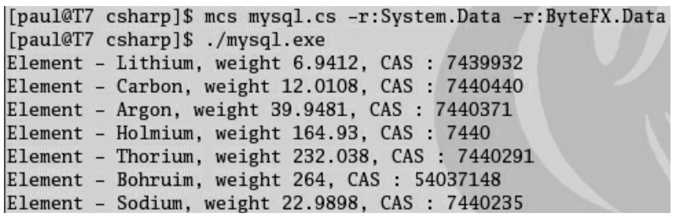
The full listing is now

```
using System;
using System.Data;
using ByteFX.Data.MySqlClient;
public class Test
{
  public static void Main(string [] args)
  {
    string connectionString = "Server=localhost;"+
       "Database=theelements;"+ "User ID=paul;"+
       "Password=thelinuxman;";
    IDbConnection dbcon;
    dbcon = new MySqlConnection(connectionString);
    dbcon.Open();
    IDbCommand dbcmd = dbcon.CreateCommand();
    string sql = "SELECT name, atweight, casreg "+
                 "FROM elementinfo";
    dbcmd.CommandText = sql;
    IDataReader reader = dbcmd.ExecuteReader();
    while(reader.Read())
    {
      string element = (string) reader["name"];
      float atweight = (float) reader["atweight"];
      long cas = (long) reader["casreg"];
      Console.WriteLine("Element - "+ element + ",
         weight "+ atweight +", CAS : "+ cas);
    }
    reader.Close();
    reader = null;
    dbcmd.Dispose();
    dbcmd = null;
    dbcon.Close();
    dbcon = null;
  }
}
```

Member	Comments
Property	
FieldCount	Gets the number of columns in the current row
IsClosed	Returns true if IDataReader is closed, else false
Item	Returns an object representing the value of the specified column in the current row. The column can be specified either by the name or zero-based integer index. This is the indexer for data reader classes.
Method	
GetDataTypeName	Gets the name of the data source data type for a given column
GetFieldType	Gets a System.Type instance representing the data type of the value contained in the column specified (uses a zero-based integer index)
GetName	Gets the name of a column specified (uses a zero-based integer index)
GetOrdinal	Gets the zero-based column ordinal for the column with the specified name
GetSchemaTable	Returns a System.Data.DataTable instance that contains metadata describing the columns contained in IDataReader.
IsDBNull	Returns true if the value in a specified column contains a NULL value, else false.
NextResult	If IDataReader includes multiple result sets (due to multiple statements being executed), NextResult moves to the next set of results.
Read	Advances the reader to the next record.

Table 2: Commonly used members of the DataReader classes

If this application is now compiled and executed, the following output is seen.



```
[paul@T7 csharp]$ mcs mysql.cs -r:System.Data -r:ByteFX.Data
[paul@T7 csharp]$ ./mysql.exe
Element - Lithium, weight 6.9412, CAS : 7439932
Element - Carbon, weight 12.0108, CAS : 7440440
Element - Argon, weight 39.9481, CAS : 7440371
Element - Holmium, weight 164.93, CAS : 7440
Element - Thorium, weight 232.038, CAS : 7440291
Element - Bohruim, weight 264, CAS : 54037148
Element - Sodium, weight 22.9898, CAS : 7440235
```

That's reading out of the way – by altering the `sqlcommand` line, you can include `where` and `order by` parameters. Writing and editing is not a great deal more difficult. However, `IDataReader` does warrant a deeper look.

The `IDataReader` extends the `System.Data.IDataRecorder` interface and together, these declare the functionality that gives the access to all aspects of the data contained in the result. Table 2 describes some of the more commonly used members of the interfaces.

In addition to that little lot, the data reader also provides a pile of other methods which takes an integer argument (the zero-based index of the column) – the names are self-explanatory : `GetBoolean`, `GetByte`, `GetBytes`, `GetChar`, `GetChars`, `GetDateTime`, `GetDecimal`, `GetDouble`, `GetFloat`, `GetGuid`, `GetInt16`, `GetInt32`, `GetInt64`, `GetString`, GetValue and `GetValues`.

Writing and Editing in C#

This is, again, quite simple. We have seen that with reading, `CreateCommand` was called. Virtually the same procedure is used for writing and updating.

Inserting fresh data is virtually the same as if you were using the MySQL monitor

```
IDbCommand dbcmd = dbcon.CreateCommand();
string mysqlcommand = "insert into
   elementinfo(id, name, ...) ";
mysqlcommand += "values(23, @name, ...)";
dbcmd.Parameters.Add("@name", "Sodium");
dbcmd.CommandText = mysqlcommand;
```

You can see a slight addition here, the use of `Parameters.Add(...)`. The reason for using this is that as you know, when adding data, you can be adding absolutely anything in. If you want to pass in a string literal though, you would normally need these to be passed in quotes. Effectively, that is what is happening here. By using `Parameters.Add()`, we can pass anything in without the requirement to use quotes.

```
IDbCommand dbcmd = dbcon.CreateCommand();
string mysqlcommand = "update elementinfo
set id = 23 where id = 108"
dbcmd.CommandText = mysqlcommand;
if (dbcmd.ExecuteNonQuery() == 1)
  Console.WriteLine("ID updated");
else
  Console.WriteLine("ID not updated");
```

For any operation which doesn't return database data (such as table creation, table deletion or inserting), `ExecuteNonQuery()` should be used. This method returns an int that specified the number of rows affected (`create table` returns `-1`). Both of these code snippets can be placed into code listing already given

That's enough for this time. Next time, I'll expand on this via a simple menu system for interrogation, insertion and deletion of data and move that simple menu system to use `System.Windows.Forms`

Paul F. Johnson

Grid and Utility Computing - The Return of the Bureau

Alan Lenton `<alan@ibgames.com>`

I recently read a piece about Sun Microsystems 'Sun Grid' computing system. It was launched about a year ago with a great deal of razzmatazz. It offered computing power on tap at a cost of US$1.00 per hour per processor, and storage at US$1.00 per Gbyte per month. Grid and Utility Computing has been all the rage in the computer trade press and among the pundits for a number of years. Indeed, one commentator I read a couple of years ago claimed that the advent of grid computing would cause IT vice-presidents to undergo the same extinction as 'Electricity' VPs did in the twenties with the advent of the national electricity grid.

It was with a rather wry smile, therefore, that I read in the article that Sun was unable, even after running Sun Grid for a year, to name a single customer!

So What Are Grid and Utility Computing?

Sloppy usage – typical of marketing hype – has led to the two words becoming interchangeable, but I would suggest that they are both about the efficient utilisation of computing resources. Each approaches the problem from different ends, but most of the spin merchants are actually talking about utility computing, not grid computing.

Grid computing is about tapping the unused processor power of existing computers, while utility computing is about having extra computing power on tap for peak usage, but only buying the resources you use, instead of having extra hardware lying around that is only used for brief peak periods.

Grid computing gained a big fillip with `seti@home`. This program brought together three things: the Internet, a supercomputing type application, and desktop computers not currently being used. The Internet was used to network the computers into something approaching a supercomputer to crunch masses of data pulled in by a radio telescope system. Its success in failing to find an extra terrestrial civilisation inspired the bloggerati to proclaim that this was the one true way forward (again).

The problem with this model is two-fold. First there is the design and programming problem. You have to be able to break the program down into discrete packets – lots of them – which can all be run completely independently. Now it is, of course, usually possible, not to say desirable, to break a problem up into independent parts (at least from a programming point of view), but there is a limit to which most problems can efficiently be broken down, and let's face it, computing power is only one of a number of limitations that a real life running program can face.

The other problem is that the number of processors available at any given time cannot be predicted in advance. This makes the concept useless for time bounded programs. It works just fine for searching for aliens that may or may not exist, or the speculative study of how proteins fold. For time bounded solution requirements, though, the computing power is just too unpredictable, which is probably why there is no `weather_today@home` program.

Utility computing is a totally different kettle of fish. The computing power is 'delivered' to your building by cable/fibre, you just plug in your terminal, log on to a remote server farm and run what applications you want.

It has a certain superficial attraction, especially to large companies with wildly fluctuating computing needs. If you are buying computing power only as you need it, then you don't have to make sure you have enough machines for peak consumption. There's also the advantage that you never pay for more than you use - and most business PCs are only used during working hours – i.e. for only a third to a half of the full day.

Utility computing is the natural successor to the large computing bureaux of the 70s and 80s. It is being pushed by the big computing companies, especially IBM, HP and Sun. It's in some ways difficult to see why they are so enthusiastic, because the implications for them if they are successful are not good. It may be that they simply haven't thought it through properly, which seems odd, but I suppose is possible. There are also major obstacles in the way and serious disadvantages from the consumer point of view.

Leaving aside, for the minute, everything else, let's assume that one of these companies succeeds in establishing utility computing as the way everyone gets their computing power. What then? Well the first thing to note is utilities of this nature are always natural monopolies, at the very least at a local level. In the industrialised world at least this has one of two consequences: either the utility is publicly owned, or if it isn't publicly owned it is heavily regulated. The latter is the most likely case in the US and the UK.

Do these big companies really want their activities regulated by local and national oversight boards? I cannot imagine why they would. And, interestingly enough, I can't think of any private utility company that hasn't tried to diversify –out– of its utility sector during the last 20 years. Indeed some of the most spectacular and massive corporate failures recently have been utility companies diversifying in search of larger profits than those allowed in their original business – Enron being only the most glaring example.

There is also the strategic question of whether putting all the data in a few massive data centres makes it more vulnerable to terrorist strikes in the post 9/11 and London Bombing period.*

But, over and above the dire consequences of success for the operating companies, there are serious flaws in the logic of utility computing. The most obvious question to ask is whether computing power is indeed the same sort of beast as electricity or water. I would suggest not, and for two main reasons.

First, it seems to me that the crux of the point is that it is easily possible for urban dwellers to obtain computing power relatively cheaply, while most cannot do so in the case of electricity or water.

A decent computer costs less than a washing machine. Few people – even companies – possess their own rivers, dams, coal mines, oil wells, or even the space to install a reasonable size generator. Interestingly, for instance, the London Underground was powered from its own power station at Lotts Road until late in the 20th century. It was economic for it to generate its own electricity, so it did so until the land it was sited on (Chelsea) became too valuable for industrial use.

As an aside, I would argue that if a new generation of electric generators which were both compact and cheap came to the market, we would see a steady move away from the electricity utilities by consumers. People, and companies, prefer to have their own resources rather than continually buy in resources from a utility. It's not just a financial thing – it's a matter of convenience too. How many people do you know who would rather use a launderette than their own washing machine, even though for most of the week the machine is unused?

The other problem is that computing power and storage are not a utility in the classic sense. The supplier doesn't push computing power or storage down a fibre optic pipe for you to use, like water or electricity. Quite to the contrary. Everything is at their end in the server and storage farms. This isn't a utility model – it's a computing bureau model, which everyone abandoned as soon as computing power became cheap enough to do so.

Then why are companies pushing utility computing, and why are big corporations starting to look interested?

Well, on the one hand, the big providers are seeking a way to re-establish the control over computing which they lost with the coming of age of the personal computer. On the other hand the primitive nature and lack of commoditisation of software (note – software, not hardware) makes anything that means you don't have to deal with Information Technology (IT) yourself looks attractive.

It is the latter that is currently driving the move to outsourcing by large corporations. And grid computing is in many ways a continuation of this trend. However, also notable is the struggle by a number of firms that outsourced their IT in the eighties and early nineties to bring their computing back in-house so they can regain control of their strategic IT.

The more you survey the domain, the more obvious it becomes that there is a large dose of wishful thinking going on here. Clearly the protagonists - both grid buyers and sellers – really do believe that the grass is greener on the other side...

The computing bureau is dead. Long live the bureau!

*My more astute readers will recognise this as a stock sales weasel 'the terroists are coming' soundbite designed to part gullible government ministers from large quantities of public cash in return for several vats of digital snake oil...

Alan Lenton

Professionalism in Programming #35

Together we stand (part two)

by Pete Goodliffe <pete@cthree.org>

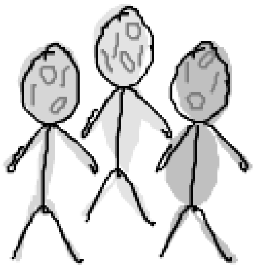
Recovering from failure is often easier than building from success.
Michael Eisner

In the last issue of C Vu we started a journey into the world of software development teamwork. We looked at team structure, personal skills for good teamwork, at team working tools, and team organisation. Phew! That's quite a lot of stuff to take in, but I guess after two months you've finally digested it all, and are now back ready for more.

If you're not, I'd quickly take an Alka-Seltzer and prepare for the next instalment…

This time we'll look at "team diseases". I don't think they're contagious, but handle with care just in case.

Team Diseases



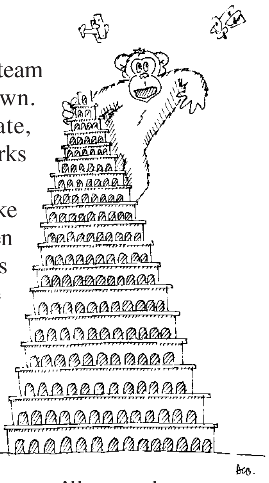
You can have a good spread of programmer types, wonderful team organisation, and use the best tools to their maximum, but still have a dysfunctional team. There are many reasons that teams fail to produce results, and just as we stereotyped different species of programmer [1] we can generalise categories of doomed development teams – to see what we can learn from them.

So here are some of the classic team disasters. In each case we'll see:

- their particular road to ruin,
- the warning signs (so the you can recognise when you're headed in this direction),
- how to turn around a team stuck in that particular rut, and
- how to be a successful programmer in that team situation (sometimes *despite* the team)[1].

Hopefully you won't recognise your current team in the following list:

1. Tower of Babel



Just like the Biblical builders, a Babel-esque team suffers a massive communication breakdown. Once the programmers fail to communicate, development work is doomed - if anything works then it's more likely by luck than by design.

With ineffective communication people make incorrect assumptions. Bits of work fall between the cracks, potential error cases are ignored, faults get forgotten about, programmers duplicate effort, interfaces are misused, problems aren't addressed, and small slippages, unnoticed, grow into mammoth project delays because no one is monitoring progress.

Of course, the original Babel builders were fragmented by multiple spoken languages[2]; without multilingual programmers you'll struggle to communicate in this situation. However, these kinds of project rarely suffer Babel syndrome – with language barriers to cross people make more of an effort.

It's not only different spoken languages that can separate developers. Different backgrounds, methodologies, programming languages, even different personalities cause team members to misunderstand one another. A small seed of confusion, unchecked, will eventually grow; resentment and frustration will build up. At worst, Babel teams end up not talking at all, with each programmer sitting in their own corner, doing their own thing.

This problem can brew within the immediate software team, and also between interacting teams. Extra-team Babel syndrome kicks in when developers fail to talk to testers, or the management team are disconnected from development.

1. I don't claim that these strategies will solve the team's general problem; they're deliberately short-sighted ways to get your work done now, with minimum risk of problems.
2. Genesis 11:1-9

Warning Signs

You can tell that your team is headed towards Babel when one developer can't be bothered to ask another about something, feeling it's not worth the effort. It creeps in with a lack of detailed specifications, and with ambiguous code contracts. You might see either too few, or too many emails flying about. Too many emails means that everybody's 'shouting', and no one's listening – nobody has time to keep up with the constant barrage of information.

On the road to Babel there are no team meetings, and no one person knows exactly what's going on in the project. Pick someone at random: they can't tell you whether development is on course or not.

Turn Arounds

Talk to people. Go on - open the floodgates! Soon they'll all be doing it.

Babel attitudes are difficult to redress once the rot has set in because morale has been dragged to an all time low, apathy is rampant, and no one believes that change is possible. The most effective strategy is to work at boosting team morale, to bring the developers closer together. Do something social to shake the team up: consider a team building exercise, even a simple trip out for a drink together. Buy some cakes one lunchtime, and share them with the team.

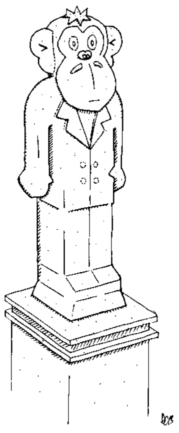
Then develop some strategies to force people to talk to one another. Create small focus groups to scope new features. Put two people in charge of a piece of design work. Introduce pair programming.

Success Strategies

To write good code in the face of such problems you have to be very disciplined. Before you start a work package ensure that it's rigorously defined. Write the specification yourself if you have to, and mail it to all the involved people to get their buy-in (provide a time limit for comments, stating that no feedback is assumed to be agreement). Then it's clear when you've succeeded because you have fulfilled the agreed spec.

Lock down all your external code interfaces fully, so there's no confusion about what you're relying on, or what people can expect of your code.

2. Dictatorship



This is the original one man show, a team led by a strong-willed, strong-personality, who is (usually) a highly skilled programmer. Other programmers are required to be 'yes men', even if they don't want to be, following the Dictator's mandates without question.

In some teams this works fine - with a well-chosen benevolent leader and a team who respect him. Problems loom when a Dictator's personality doesn't support his position, or when he is technically substandard. If his ego gets in the way then the team is in trouble: they will resent him, and grind to a frustrated halt.

When fashioned on purpose, this kind of team is a hierarchy, with lines of defined authority. This structure was likened to a *surgical team* by Frederick Brooks [2]. The surgical team places the most highly-qualified technical individual, the lead surgeon[3], at the top of the pile: acting as a code writer, *not* a manager. He performs the bulk of the development and has ultimate responsibility if bad things happen (if the patient dies). He is backed up by a deliberately chosen team. This includes a junior surgeon who performs smaller, lower risk tasks, supports the lead surgeon, and learns the trade. The team also involves the software equivalent of anaesthetists, nurses, and perhaps more junior surgeons learning skills (e.g. sewing the patient up).

There are two dangers with this kind of team. The first comes when external pressures force the Dictator to become more of a manager; his technical specialism pretty much guarantees his management will suck. His focus will shift away from the software and the project will collapse. The second danger is a self-appointed Dictator, who isn't recognised by the team. Work flow will stall as the team is neither structured nor prepared to support his leadership.

Warning signs

This team structure tends to develop slowly and subtly, as a would-be Dictator slowly modifies the focus of his work role and presumes his level of authority. You can see a Dictated team brewing when you often find yourself saying:

- I can't do this without consulting ...
- Oh, ... will moan if we do it like that.
- But ... says we must do ... first.

3. Usually this guy is a technology specialist, as defined by Belbin's team roles.

Turn Arounds

If you have a Dictator who is not a worthy lead surgeon, then you must address the situation. Otherwise the team will petrify under this authoritarian tyrant. Either work the issues through with him (in all honesty, this is unlikely to work – change is hard, especially for people with an inflated ego), or unseat them from their throne by confronting a manager on the issue.

The problem with doing this is that after overthrowing the king, without a team restructure, you need a new king. Lead surgeons are hard to come by, and so it's probably better to restructure the team.

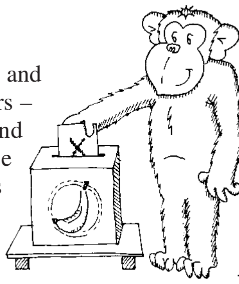
Success Strategies

In a (functional or dysfunctional) Dictatorship, determine your level of authority and responsibility. Get this agreed by the people whose opinion really counts – your manager or team leader.

However, once you've asserted your rightful development role, you (and the other programmers) must still listen to and work with the Dictator, even if you don't like his current position. Otherwise you won't work well together and won't write complementary code. There must be consensus in the design, or the software will not work.

Don't be disrespectful or rude about a Dictator – it'll bring the team morale down and make you more cross.

3. Development Democracy



An old proverb says: *all men are created equal*, and here this is outworked. This is a team of peers – programmers with similar levels of skill and complementary personalities – who organise themselves in a non-hierarchical fashion. It's an unusual beast in the corporate world, which expects that someone must be 'boss'. The idea of a self-organising team seems heretical. However, it has been shown to be a team model that can work well. Some Democratic teams run by periodically electing a leader from their ranks, based on whose skills are most in demand at this stage of the project. Often there is no a clear leader, and all decisions are taken by consensus.

We tend to forget the other half of that proverb: *all men are created equal, but by practice grow apart.* It takes a special set of individuals to make this team culture work. The danger with a team founded on this laudable principle is that as it grows, or when a certain member leaves (the one who crystallises the group into making decisions), things begin to drift. The team can lose it's focus, failing to agree on anything, and failing to produce results in a timely fashion. In the worst case, the team ends up arguing forever about a single issue, contemplating its navel, and never actually achieving anything.

With endless meetings and circular discussions the team is in danger of *analysis paralysis*: of becoming focused on process, not on delivery of the project. Like a real democracy, the genuine team business can get lost in a sea of politicking.

You can accidentally end up with a Development Democracy if you have a ineffective team leader who is incapable of making decisions. This kind of bumbling leader will slowly phase himself out without realising it. The frustrated team ends up jointly taking over his role – forcing decisions to be made and choosing the direction of development.

Democracy is a particularly difficult team structure in a crisis, even when established on purpose. If personality friction prevents the election of the right man for the situation, then an outside leader must be brought in to steer the project.

Warning Signs

You can smell a sick Democracy a mile off: the rate of decision making drops like a stone. If there is a software team leader then everyone bypasses him, rather than be stalled by his dithering. He is now a leader in name only; no one recognises him as being an authority on the software or able to achieve anything.

Without strong leadership, no one is assigned responsibility for each task; it's never clear who should be ensuring a task's completion, and so nothing gets done. Weeks can go by without a specification being completed, and with no visible progress.

In a rampant Development Democracy the smallest decision forces the team into committee mode, and it takes days to conclude. Or a decision is made: *let's say 'yes' until we decide to do something else.* "Let your 'yes' be yes, and your 'no' be no" [4], otherwise you'll spend ages ripping up old code and redoing it whenever someone changes their mind.

You might also notice that junior programmers feel alienated because they'll never be elected leader.

Turn Arounds

Democracies aim to remove a specific bottleneck: where all decisions must be made by the boss, who is not always the most appropriate person (especially when they're not technical). In a dysfunctional Democracy there is no decision making process, and no decisions are made at any level. To return to a healthy Democracy, ensure that leadership can move around the team freely and that replacing the leader is easy. Don't attempt to run a Democracy unless you have enough potential leaders.

As with any other slipping project, make sure that problems are visible to everybody, both developers and managers. Make sure that it's clear who's responsibility this problem is – especially if it's not yours!

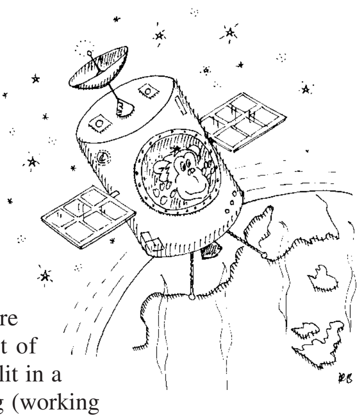
You can attempt to correct indecisive Democracies by showing some strong will; don't be content to let matters continually slide. You'll probably get a name as a trouble maker, but eventually you'll also get a name as someone who achieves results. Beware, though, of the danger of becoming a demi-Dictator as a backlash.

Success Strategies

For your own sanity, avoid ditherers – the people who cannot decide the simplest thing.

Ensure that you are allotted a well-defined part of the project, and have clear and realistic deadlines. This is a major anchor against the ebb and flow of uncertain leadership.

4. Satellite Station



A Satellite team – split from the main development team – present their own world of potential pain and pitfalls. It's hard to work as a cohesive unit when part of the team is physically separated. Think about using a severed limb; it's not dissimilar.

The Satellite might be an entire peripheral department, or a part of your immediate software team split in a different location. Telecommuting (working from home) is a special case, with only one person in the Satellite.

It's not unusual for upper management to be in a 'head office' elsewhere, but since they have little input to the day-to-day programming activities this isn't problematic. However, if test or other development teams are many miles away then you need to put measures in place to ensure that the project succeeds. You must be deliberate about this – split teams don't work together by accident.

Programming requires close team interaction because our individual pieces of code must interact closely. Anything that threatens our human interactions threatens our code too. These are the kinds of threat presented by Satellite teams, so take them seriously:

- Physically disjoint development teams lose those informal, spontaneous conversations that spring up beside the coffee machine. The chance for easy dynamic cooperation disappears. With it goes a level of shared insight and group understanding of the code
- There is a lack of cohesion in development. Each site's local practices and development culture will differ (if only slightly). Inconsistent methodologies make handing over work complex.
- Since you don't know people in the Satellite very well, there is an inevitable lack of trust and familiarity. A them and us attitude quickly emerges.
- Another old proverb says: "out of sight, out of mind". When you don't see Satellite programmers regularly you'll forget to ask how they're progressing, and won't think whether your change impacts them (technically or procedurally).
- Satellites make the simplest conversation difficult. You need greater awareness of other programmers' schedules; when they're in meetings or on holiday.
- With cross-country projects, time zone changes become especially problematic. There is a smaller communication window between teams, and a larger eclipse period.

4. Matt 5:37, unless you're a Babel builder, in which case your 'yes' might be *Oui* and your 'no' *Nein*!

Warning Signs

Geographically split teams are obvious, but also be wary of separated teams within the same office. Splitting developers into different rooms, or even across corridors, imposes an artificial divide that can impede collaboration.

Watch for separation between departments, too. It can be just as damaging. For example, test teams are often hived off separately from the developers, sometimes in a different office or section of the building. This is a real shame; it hinders essential interaction between the teams and so the QA process is not as fluid as it could be.

Turn Arounds

A Satellite Station team is not necessarily doomed; it just requires careful monitoring and management. The problems are not insurmountable, but definitely inconvenient - avoid them if you can.

An essential survival strategy is to get all team members meeting face-to-face very early on in the project. This helps to build a rapport, trust and understanding. Regular meetings are even better. When the team assembles provide food and drinks; this sets people at ease and creates a more social atmosphere.

Arrange the Satellite so that their work requires the least collaboration and coordination with the mothership. This will minimise the impact of any problems.

Avoid code interaction problems by defining interfaces between the separate sites' work early on. But beware of designing your code around the team – you might not be creating the most appropriate design. Programming is a process of making pragmatic choices; each individual case is slightly different, and this is balance you have to weigh up each time.

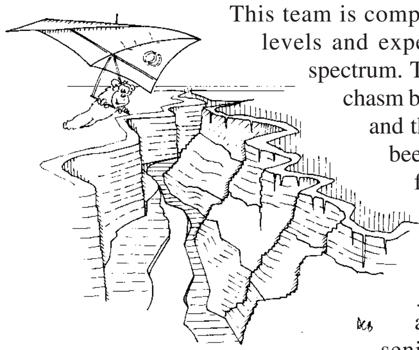
Groupware becomes an essential tool in a Satellite, to make communication effective. Also consider using instant message communication between sites. And remember: don't be scared of the telephone!

Success Strategies

If you have to work with off-site people, make sure that you know them well – personally and professionally. It makes a big difference. You'll know how they react, and when they are being sincere or sarcastic. Make an effort to be friendly to Satellite programmers – it's easy to be mistaken for a grumpy idiot when they only ever phone you at inconvenient times.

Make sure that you know exactly who is off site. Learn everybody's name, find out what they do, and how to contact them. Work at improving your communications skills. Don't be afraid to contact someone when you need to – think whether you'd talk to them if they were sat beside you.

5. The Grand Canyon



This team is comprised of members with skill levels and experience at either end of the spectrum. There is a clear skills gap; the chasm between the 'senior' developers and the 'junior' developers has not been bridged, and so two distinct factions have grown. In almost every Grand Canyon team, this is both a social and technical phenomenon – the junior programmers socialise amongst themselves, and the senior programmers socialise amongst themselves. This isn't helped when the senior developers are seated together in one lump, and the junior developers in a separate ghetto.

The reason for Grand Canyon culture is often historical: a project starts with a small number of crack developers who must quickly establish an architecture and get proof-of-concept code out the door. They are naturally seated together and learn to work as a swift, cohesive unit. As the project progresses more programmers are required, and junior members are brought in. Because of the existing office layout they are seated on the periphery, and then given smaller programming tasks in order to learn the structure of the system.

Without careful checking, senior developers can adopt a superior attitude and look down on the junior developers. They hand over small, tedious chunks of work and continue with the interesting grander design work. The senior developers reason that it would take prohibitively long to teach a junior about the bigger picture, and there is an element of truth there. In this way, the junior developers never get a chance to gain more responsibility and do more 'fun' programming. They get frustrated and disillusioned.

Junior programmers want to learn their trade, and still have youthful enthusiasm and a passion for programming. Senior programmers may have a very different (more jaded?) world view, with aspirations for management or more senior development roles. These different personal motivations pull the factions in different directions.

Warning Signs

Watch your team if it is currently growing. Look carefully at the demographics of the members and watch how work is allotted amongst them. Monitor the social dynamics of your team – unhealthy teams develop cliques.

Turn Arounds

The problem in a Grand Canyon is that the team is not mixing; there are polarised factions. The fix is simple: adopt strategies that will mix them up. For example:

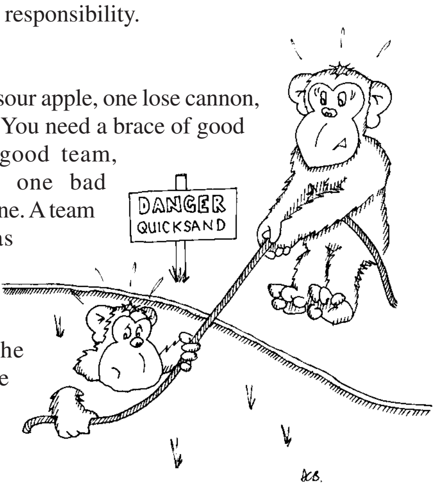
- Change the seating plan, so that both factions are interspersed. This might consume valuable development time, but a day of desk moving might win weeks of productivity.
- Introduce team meetings to spread information.
- Start pair programming, mixing senior and junior programmers. Get the junior one to drive, whilst the senior navigates. This is a discipline for the senior, and educational for the junior.
- Begin a mentoring scheme to train junior developers. Although this will emphasise the skills divide, it will also force the factions closer.
- Look at all the developer's job titles – do they foster a dangerous and unnecessary pecking order?

Success Strategies

Treat everyone as an equal, as a peer.

- If you're a senior programmer, recognise that the juniors need to learn. You were once a novice too, and didn't understand how the world worked. Don't hog all the interesting programming tasks. Be willing to let others take responsibility.
- If you're a junior programmer, ask for more challenging tasks. Seek to learn. Perform your current task as well as you can – this will prove that you are ready for greater responsibility.

6. Quicksand



It takes just one person, one sour apple, one lose cannon, to bring a team to its knees. You need a brace of good programmers to make a good team, however you only need one bad programmer to make a bad one. A team stuck in Quicksand has unwittingly fallen foul of a rogue member. This can be subtle; they might not have even spotted where the problem stems, and the culprit probably has no intention of causing any harm.

You might suffer Quicksand for a number of reasons:

- A technically incompetent programmer (probably the Cowboy coder [1]). This guy isn't easy to spot immediately, and no one will notice whilst he's writing poor code. The timebomb has been laid, and later the project will be stalled until his mess has been purged and replaced.
- A morale drain sits under a little black cloud and demoralises the entire team, sucking out all enthusiasm and cheer. Within a few weeks no one can bring themselves to write any code, and they're all considering jumping off the nearest bridge.
- A mis-manager performs the exact opposite actions of a good manager, constantly changing decisions, altering priorities, shifting time-scales, and promising the impossible to customers. The team members don't know where they stand because the ground is always moving under their feet.
- A time warp programmer bends the laws of relativity, so that time slows down around him. Anything coming his way takes a phenomenally long time to process. Decisions stall on his input, his coding work doesn't

[concluded at foot of next page]

Silas's Corner

Silas S. Brown <ssb22@cam.ac.uk>

ROX Filer

ROX Filer (available from `rox.sourceforge.net` or provided as a package in all major Linux distributions) is a drag-and-drop file manager for Linux that is progressing well. A file manager can be useful if you are trying to browse or re-organise a project that employs files and directories to represent its structure, as opposed to ignoring the filing system and using grep to find everything, which can be tempting when it is necessary to have too many automatically-generated files in the same directory or to utilise mail folders and other databases which do not necessarily use the filing system.

A major advantage of ROX over other file managers is that it is lightweight, while still being graphical. Being lightweight is particularly useful if you are on older hardware or a heavily-used server - you can load and run it without consuming large amounts of RAM or causing a lot of disk activity, which can make it more suitable than KDE or Gnome on such systems.

Some of the user interface ideas have been taken from Acorn's RISC OS, such as automatic window resizing (a directory window is never larger than it needs to be). It doesn't yet have everything that RISC OS has; for example, there are no Director-like menus for rapid browsing of complex directory structures, and the menus that are there do not have all the functionality of RISC OS menus such asthe ability to right-click on an item to select it without dismissing the menu (useful in a complex hierarchy; the GIMP's floating menus can achieve the same thing but they require an extra step). Needless to say there is not as much integration with applications as there is on RISC OS, although some effort is being made to produce applications that do co-operate with ROX, and it's often possible to configure existing applications to send commands to ROX via the command line, which can remotely control an existing instance of ROX as easily as starting a new one. ROX does have some nice touches that are not found in RISC OS, such as briefly flashing the directory you came from when you go up a directory. On balance it is worth knowing about. ROX is reasonably customisable, and hopefully in future it will have better support for unusual characters in filenames.

Silas Brown

[continued from previous page]

get done, and meetings always start late because he can't quite make the start. There's always a good reason – perhaps he is doing other 'very important' jobs – but he amasses a backlog of tasks and never gets round to anything. Eventually other programmers get fed up and bypass him.

In a Quicksand team one member's weakness can quickly destroy the entire team's productivity. This is especially dangerous when the culprit is high up the food chain. The more responsibility he has, the more dire the consequences.

Warning Signs

Look for the one guy who doesn't gel with the team. He's the person that everyone complains about[5], or the programmer who always works alone (because everyone avoids him).

Turn Arounds

The most drastic but probably the easiest fix is to get rid of the Quicksand cause. First you have to identify him, and sometimes that's quite difficult.

Calls of unfair dismissal frighten managers, who will be reluctant to fire someone because 'a few people can't get on with him'. It takes some major league incompetence to make this a likely outcome.

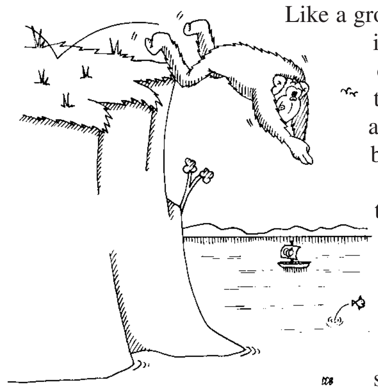
So you've got to find a way to minimise the chaos he can cause, or work out ways to integrate him into the team better.

Success Strategies

Most importantly: don't be the Quicksand!

Presuming you're not, try to insulate yourself as much as possible from the effects of a Quicksand team member. Limit interaction with him, for the sake your blood pressure. Don't rely on his code too much, and try to avoid his input as much as possible. Don't get sucked into his bad practices, and don't over-react to him - acting the exact opposite and making matters worse.

7. Lemmings



Like a group of cute, furry animals with an insane urge to launch themselves off the nearest cliff, this team is far too willing - even eager - to accommodate the brief they've been given. Even when it's bogus.

The team comprises of very trusting, very loyal members. They are technically competent, but don't see beyond their specific instructions. Their enthusiasm and eagerness are commendable, but without a visionary member - someone who asks *why?*, who looks beyond the spec to what's really required - the team is in constant danger of delivering what was asked, but not what was needed.

Lemming teams are particularly vulnerable to the demands of start-up companies. The disease starts when managers ask: *write this code quickly; we'll redo it properly later.* Later never comes, instead the Lemmings hear: *the company needs more code, fast, so just bolt this on quickly too.* Before long the team culture is to dance when someone plays music. The work gets slowly more and more difficult, with ever-more herculean tasks and an ever-decaying codebase.

Eventually the team find themselves a broken mess at the bottom of a sixty foot cliff. Game over.

Warning Signs

If you're not happy with the specification you're currently working to then you may be in a team of Lemmings. You need a realistic faith in your current project, or you're a mere code mercenary. When you find yourself listening to vacuous promises and being committed to unreasonable work, and when no one argues or points out flaws in the plan, welcome to Lemming country. We hope you enjoy your stay.

Turn Arounds

Review what your team is doing right now. Don't stop working, but take a view from the customer requirements right through to final delivery. Will the code you're working on provide what's ultimately needed? Is it a short sighted hack that won't stand the strain of many years in your codebase, or many years of use?

Success Strategies

Question the work you are given. Understand the motivation for it. Stand up for good programming principles, and never believe that you'll be allowed to fix code later, unless you can see it scheduled on a plan that you believe in.

Next Time

Having thoroughly depressed ourselves at the sight of failing software development teams, we'll conclude this series by developing some principles for healthy collaborative software construction, and looking at the natural life cycle of a software team. Until then, enjoy the software factory…

Pete Goodliffe

References

[1] Pete Goodliffe. 'Professionalism in programming #30 - #31: Code Monkeys' in *C Vu 17.1* and *17.2*, 2005.

[2] Frederick P Brooks Jr. *The Mythical Man Month*. Anniversary Edition. Addison-Wesley, 1995.

With thanks to David Brookes, cartoonist extraordinaire, for the excellent monkey pictures.

5. They'll complain behind his back, which is a part of what drags the team into Quicksand. No one addresses the problem head on. No one likes to rock the boat. It will take more effort to confront him than anyone can be bothered to invest.

Cryptographic Mistakes Made in Programming

Jonathan Wignall <jwignall@sthelens.ac.uk>

Cryptography is regarded by many, including some of its practitioners, as a black art. This is not without good reason as very few people truly understand the subject. Most cryptographers are mathematicians, and now virtually all programmers are not true mathematicians (unlike in the early days of computing). As a result programmers who actually understand cryptography are very rare.

Hearing the warnings about attempting to program cryptography algorithms yourself, it is often viewed as easier, and safer, to use pre-built cryptographic libraries to handle any encryption requirements. It's natural to assume if these libraries have been constructed by cryptographic experts, then use of them should result in secure data. Unfortunately this assumption is the root of a lot of encryption mistakes by programmers.

In cryptography the strength of the algorithm plus the length of the key are only one part of the security of encryption. Given poor usage, or poor implementation the most secure encryption systems can be broken. This is a common mistake in the field of cryptographic programming; the assumption that strong cryptography always makes the data secure.

So what errors can we, as programmers, make that turns a secure encryption algorithm into an insecure system? To demonstrate this let's look at the only encryption algorithm that is mathematically unbreakable, the one time pad.

One time pad works via XOR'ing data bits with a pad as long as the message. The pad is selected at random. The result is data that is only de-cryptable with the pad and is unbreakable without the pad. The one time pad is impracticable in the real world to implement, but is ideal to demonstrate how programming mistakes can ruin encryption, if the programmer isn't thinking when coding, even this 'unbreakable' system can become breakable.

Let's look at a few of the more common mistakes that can be made:

Mistake 1: Leaking Data Bits

With eight bit ASCII, most characters used in English documents are in the lower seven bits. Due to different development platforms, some encryption libraries may be configured to assume a seven bit ASCII input. If we mismatch the two we could end up encrypting seven out of eight bits in our data and end up 'leaking' one unencrypted bit per byte.

The resulting data will look encrypted, but could give clues over the plain text in the encrypted data. Values greater the 128 decimal, are probably the pound sign in financial documents for example.

Mistake 2: Repeating the Key

Most public and symmetric key encryption systems (real world cryptography) repeat the key. The longer the key and the less it ss repeated in encrypting data, so is regarded as more secure. Data by users may also be repeated, if this occurs then comparisons 'block by block' could reveal the encryption key.

A good example of this is the weakness in WEP (Wireless Encryption Protocol), the encryption standard used with 802.11B wireless networks. This uses a 24bit field and a wireless access point transmits a lot of similar packets, it is guaranteed to reuse the same key stream at some point. By comparing two encrypted packets its possible to find the plain encrypted text. Once that's done its trivial to decrypt other encrypted data.

The algorithm WEP uses RC4 which most would accept as a decent algorithm for most uses, what lets WEP down in this case, is how the algorithm was used, and the data it was used to encrypt.

Mistake 3: Formatting the Data Prior to Encryption

In the following logon system, a user types in a password. The systems breaks the password up into chunks of three characters and upcases each character. Separately each chunk is encrypted with the algorithm and compared to the stored encrypted password to see if it matches. Passwords are limited to letters and numbers only.

The algorithm is solid, but unfortunately the possible inputs to it are now only 46,665. This is a low number and hence allows anyone with access to the password file the possibility of working out the password in less than a second.

The programmer has effectively ruined a secure encryption system, by the way the data was processed prior to encryption.

Mistake 4: Key Selection

A good algorithm (or one time pad) needs a truly random key. Problems occur with generating random numbers on computers as computer generated numbers aren't truly random. It has been known for programmers to use their compilers default random number generators to create keys, a repeatable process.

To get around this some programmers develop their own method's of key randomisation, but these often fail to produce keys that aren't predictable, or result in picking a key that's weaker than could be.

Just because a key is random, doesn't mean it is secure. Depending upon the algorithm some keys are stronger than others, particularly with public key cryptography. This is similar to physical locks as most lock pickers can, if they see a key, judge how hard it will be to pick that lock. The same make of lock will be easier to pick, or harder, depending on how the key is cut.

The failure to concentrate on good key generation is a major drawback in many practical deployments of cryptography.

Mistake 5: Leaking Data and Keys

We have encrypted data stored on media, we wish to work with that data. so we load into memory our decryption key, decrypt all the data off the media and work with the unencrypted data in memory. The result? We are trusting the operating system of the computer to keep the data in memory secure, and not dump memory to disk. Virtual memory is a very large concern as both the plain text, and encrypted text could be present to aid an attacker working out the key. In the worst case scenario the key could be in the virtual memory.

Mistake 6: Insecure Algorithm

If the lock is weak, any length of key will result in weak encryption. Several algorithms over the centuries have been proven to have flaws that aid an attacker in decrypting data. It doesn't mean you shouldn't use these weak algorithms, but only use them if the data doesn't need to be secure, but instead just look secure. This is often acceptable if the people who are after the data, lack the resources to break the code.

Conclusion

Encryption needs to be approached carefully and not just assumed to be an add-on. Many problems can be solved by taking care in program design. For example if we keep most data encrypted and only decrypt data when needed we can avoid having large samples of decrypted data in memory. Other techniques involve being careful over how the data is encrypted or what data is selected for encryption.

Be Sure To:

- Pick a strong modern algorithm
- Generate real random keys, that are not weak for your encryption system.
- Use a long key length if security is not secondary to performance concerns.
- Decrypt data only when needed
- Be careful about pre-processing data prior to encryption.
- Avoid encrypting the same data multiple times.

Encryption many be a black art, but understanding how it works isn't needed to incorporate secure encryption into programs. Instead the programmer needs to know how poor implementation occurs and how to avoid reproducing it, unless your aim is to just give the your users the illusion their data is secure!

Jonathan Wignall

Matrix Linking - 'Sharp as C' (continued)

George Shagov <georgeshagov@mail.ru>

Introduction

Everyone knows what static and dynamic linking is. It would be a waste of time to delve either into their comparison or description once again. The basic problem with standard linkage is that if bug is found, we need to rebuild the code and restart the system. Having repeated again and again these procedures, it has become one of most boring and, perhaps also the time-consuming, task during the developers working hours. The idea of matrix linking is intended to annihilate these steps of recompiling and/or restarting the system.

The Idea, The Theory

Let me put some theory in the beginning, it will be not so simple and funny as practice, yet putting the practice before the theory might look strange. Having this in mind there is a reason to skip this clause and go directly to the next one, leaving this for the more leisure time of yours.

In theory, abstract theory, we have some source code, implemented by means of some algorithmic language. Let us call this source code an algorithm. Let those modules which we will have as result of compilation of our source code (an algorithm) be called "basic modules". These "basic modules" are program modules. These modules are to be loadable at run-time.

What I'm trying to describe is in general terms is an application based on C language. In other words, that all we have is the C-source code which after compilation becomes shared libraries. Actually, that's it.

Also let me invent one additional program module; let it be called a "dispatching" one. This dispatching module will have two-sized matrix of pointers. Let us say the number of rows in this matrix will be equal to amount of "basic" modules, and each particular row will be corresponded to the "basic" module in such a way that the number of columns in the row will be equal to the number of functions in the corresponded module, and, as you might guess, each particular element in this row will correspond to the function of the module (it will be described a little bit lately in details).

Line #	Lexeme	Comments	
1	Declare function1	Declaration of function1	
2	Begin	Body of function1	
3	End		
4			
5	Declare function2	Declaration of function2	
6	Begin		Body of function2
7	Call function1	Call for function1	
8	End		

Diagram 1

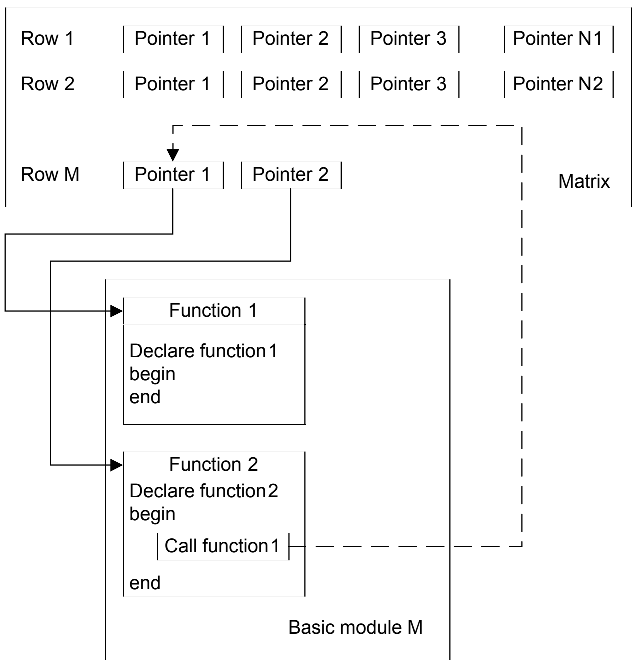


Figure 1

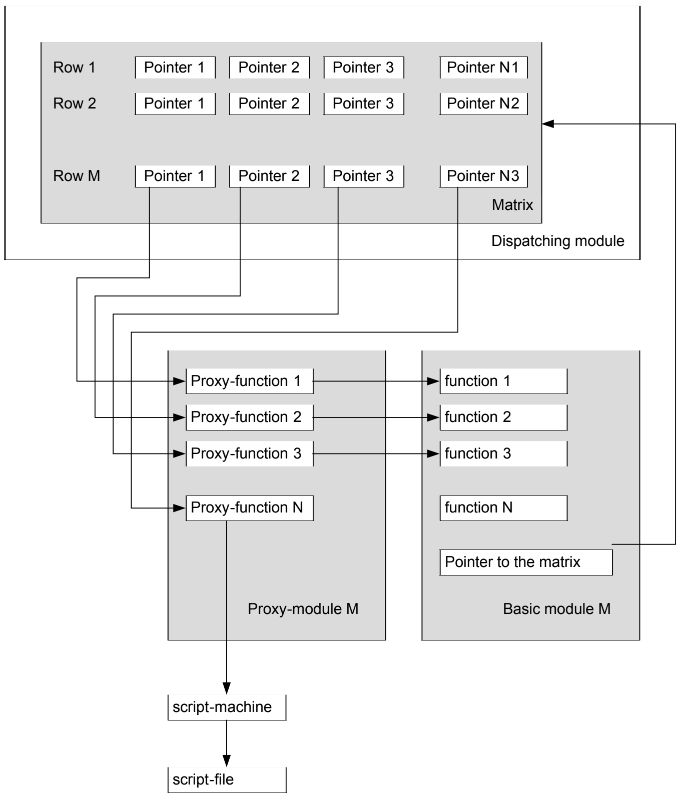


Figure 2

The so-called "basic" module has a pecularity, or rather to say that peculiarity has its compiled code. Let us consider a sample:

In diagram 1 we have an abstract source code (algorithm), written in abstract language. The algorithm represents two functions – `function1` and `function2`. `function2` has a call to `function1`.

Let us say this source code of some basic module and this module has a number `M`, which corresponds to row `M` in a matrix. The peculiarity of the compiled code of such a module will be that the call for `function2` inside `function1` will not be compiled as a direct call to a `function1`, yet instead of that it will be compiled as a call to a function which is placed in row `M` column `1` of the matrix (in this case `function1` corresponds to a column 1).

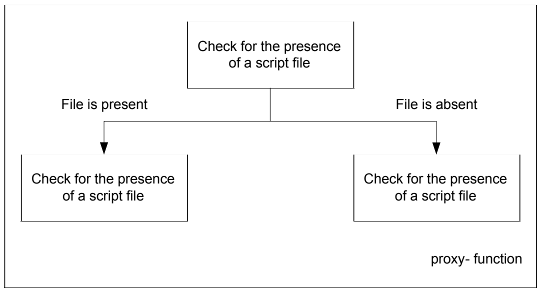
Let us consider figure 1. The solid line means that the value of the element in the matrix points to a corresponding function in the corresponding module, for instance Pointer 1 in Row M has a value of function 1 of module M.

The dotted line means that the call of a specific function will be 'translated' to a call of a corresponding function, the pointer of which is placed in a corresponding position in a matrix. For instance the call for `function1` of module M will be realized as a call to a function whose pointer is placed in Row M column 1 of a matrix.

Perhaps there is one more thing to be mentioned before we start with practice. This is the so-called "proxy-module". We have the program module similar to "basic" one, (the same stuff of functions) with that difference that the functions of the "proxy module" are not implementing the algorithm, but rather make a decision about what "basic" module to use in order to run such a function or, it might be, what particular script-file, implementing such a function of the algorithm, to be executed, using a script-machine.

Let us consider the figure 2.

The algorithm of a proxy function in this sample looks pretty simple, videlicet: in case the script-file is found it's going to be executed, using the script-machine, if not – the corresponding function from a corresponding basic module will be executed.



Check for the presence of a script file
File is present
File is absent
Check for the presence of a script file
Check for the presence of a script file
proxy- function

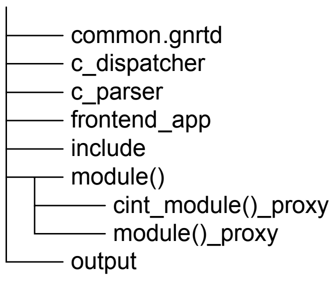
Figure 3

The Practice

Just one, but very important limitation: the source code is implemented in plain C.

This section will show how it works. (The sample code is written for Microsoft windows platform)

On the right is an application tree. Inside the folder `frontend_app`, the main (console) application is located. There is only one file there: `frontend_app.cpp`



```
/** frontend_app.cpp
 *  (c) George Shagov, 2005
 */
#include "./../include/os.h"
#include <stdio.h>
#include <ctype.h>
#include "./../include/my_structs.h"
#include "./../common.gnrtd/
   script01.fntypes.gnrtd.h"
#include "G__ci.h"

typedef int (__cdecl *D_EXECUTE)();
typedef HINSTANCE (__cdecl *D_GETMODULE)
   (const char*);
static char* s_sUsage = "Usage:\ntype 'd' to execute
   dispatching script\ntype 'e' to execute the entry
   point.\ntype 'x' or 'q' to exit\n";

int main(int argc, char* argv[])
{
   /*
    * declarations
    */
   char c = ' ';
   HINSTANCE hInstanceEntry = NULL;
   c__my_entry_point_type pEntry = NULL;
   char sDispatherPath[128];
   HINSTANCE hInstanceDisp = NULL;
   D_EXECUTE pExecute = NULL;
   D_GETMODULE pGetModule = NULL;
   SMyStructure myStruct;
   char sMyString[32];

/*getting dispatching module and its entry-points*/
  sprintf(sDispatherPath, "%sc_dispatcher.dll",
     PATH_TO_OUTPUT);
  hInstanceDisp = LOADLIBRARY(sDispatherPath,
     RTLD_NOW);
  pExecute = (D_EXECUTE)GETPROCADDRESS
     (hInstanceDisp, "g_Execute");
  pGetModule = (D_GETMODULE)GETPROCADDRESS
     (hInstanceDisp, "g_GetModule");

  /*main loop*/
  while (c != 'x' && c != 'q')
  {
    /*initial data for the script*/
    strcpy(sMyString, "My string here.");
    myStruct.m_nVal = 0;
    strcpy(myStruct.m_sString, "initial");
    switch(c)
    {
    case 'd':
      /*loading "basic" module here*/
      printf("dispatching...\n");
      pExecute();
      break;
    case 'e':

      /*executing the script using loaded "basic"
        module */
      {
        printf("executing\n");
        G__init_cint(CINT_COMMAND_STRING);
        hInstanceEntry = pGetModule("module0");
        pEntry = (c__my_entry_point_type)
           GETPROCADDRESS(hInstanceEntry,
           "c__my_entry_point_impl");
        pEntry(argc, sMyString, &myStruct);
        G__scratch_all(); /* Clean up Cint */
      }
        break;
      default:
        break;
    }
    printf(s_sUsage);
    c=getchar();
  }
    return 0;
}
```

As you can see here there are two procedures here:

1. The dispatching procedure. The realization of which is placed into the `c_dispatcher` module. And the basic idea is that the dispatcher loads the module we need. (case 'd')
2. The second procedure is executing the algorithm itself. (case 'e')

Here is the algorithm (script) which is placed in the `module0` folder:

```
/** script01.c_
 * (c) George Shagov, 2005 */
int c__get_value_1_impl(char* pString)
{
  return 1;
}
int c__get_value_2_impl(int nArg)
{
  return 2;
}
int c__call_in_case_varables_are_equal_impl
   (SMyStructure* pMyStruct)
{
  pMyStruct->m_nVal = 0;
  strcpy(pMyStruct->m_sString, "equal");
  return 0;
}
int c__call_in_case_varables_are_not_equal_impl
   (SMyStructure* pMyStruct)
{
  pMyStruct->m_nVal = 0;
  strcpy(pMyStruct->m_sString, "not equal");
  return 0;
}
int c__re_entry_impl(int nArg, char* pString,
   SMyStructure* pMyStruct)
{
  int nVar1 = c__get_value_1(pString);
  int nVar2 = c__get_value_2(nArg);
  if (nVar1 == nVar2)
  {
    c__call_in_case_varables_are_equal(pMyStruct);
  }
  else
  {
    c__call_in_case_varables_are_not_equal
       (pMyStruct);
  }
  return 11;
}
int c__my_entry_point_impl(int nArg, char* pString,
   SMyStructure* pMyStruct)
{
  int nRet;
  printf("-----------\nbefore:\n");
  printf("nArg: %d, string: %s\n", nArg, pString);
  printf("pMyStruct->m_nVal: %d,
     pMyStruct->m_sString: %s\n", pMyStruct->m_nVal,
     pMyStruct->m_sString);
  nRet = c__re_entry(nArg, pString, pMyStruct);
  printf("++++++after:\n");
  printf("nArg: %d, string: %s\n", nArg, pString);
  printf("pMyStruct->m_nVal: %d,
     pMyStruct->m_sString: %s\n", pMyStruct->m_nVal,
     pMyStruct->m_sString);
  printf("ret: %d\n-------------\n", nRet);
  return nRet;
}
```

One more thing to be shown:

```
/** my_structs.h
 * (c) George Shagov, 2005 */
#ifndef __MY_STRUCTS_H__
#define __MY_STRUCTS_H__
typedef struct SMyStructure
{
  int m_nVal;
  char m_sString[16];
} SMyStructure;
#endif /* __MY_STRUCTS_H__ */
```

`c__my_entry_point_impl` is an entry point to be called from `frontend_app`. `Script01.gnrtd.c` is the mere copy of the original script. `Script01.gnrtd.h` represents the declarations.
To complete the idea,some additional code needs to be shown:

```
#define c__get_value_1 c__get_value_1_stub
#define c__get_value_2 c__get_value_2_stub
#define c__call_in_case_varables_are_equal
   c__call_in_case_varables_are_equal_stub
#define c__call_in_case_varables_are_not_equal
   c__call_in_case_varables_are_not_equal_stub
#define c__re_entry c__re_entry_stub
#define c__my_entry_point c__my_entry_point_stub
```

What these `_stub` functions are, will be explained a little bit later.

Now, getting back to the theory. The C script here is a matrix, with an atomic element – a function. This says that calls inside the script should not go directly to the implementation but rather should go through to that element which is placed at the corresponding position in the matrix. It may look complex at first sight, but it gives us exactly that flexibility which we are looking for. The approach says also that any particular element in the matrix might be substituted to any other without restarting the system. All the calls to this element will go through the newly 'loaded' functionality.

This is the basic principle; let's examine it working.

Each particular module, realizing script functionality, should also provide additional entry points in order to identify itself and instantiate the matrix. The matrix is pretty simple, like this:

```
typedef struct S_FnTable
{
  void* _pTable[C__MAX_FUNCTIONS];
}
S_FnTable;
typedef S_FnTable Matrix[C__MAX_MODULES];
```

This additional functionality, by means of which each particular 'basic' module to be extended may look like this:

```
/** module0.dll.c
 *  (c) George Shagov, 2005 */
#include "./../include/os.h"
#include "./../include/c_fn_s.h"
#include "module0.dll.h "
S_FnTable g_pFnTables[C__MAX_MODULES];
static int g_nModuleID = 0;
int g_GetModuleID()
{
  return g_nModuleID;
}
```

```
int g_SetFnTable(int nModuleID,
   const S_FnTable* pTable)
{
  memcpy(g_pFnTables[nModuleID]._pTable, pTable,
     sizeof(S_FnTable));
  return 0;
}
```

And also the code, which fulfills the matrix, or rather to say one raw section of it:

```
/** script01.fntable.c
 * (c) George Shagov, 2005 */
/*******************************************
 * this file is automatically generated from
 * script01.c_ - do not mofify it
************************************** *****/
#include "./../include/os.h"
#include "./../include/my_structs.h"
#include "./../common.gnrtd/
   script01.fntypes.gnrtd.h"
#include "./../include/c_fn_s.h"
#include "./script01.gnrtd.h"
#include "module0.dll.h"
int g_GetFnTable(S_FnTable* pTable)
{
  void* pProc = NULL;
  pTable->_pTable[c__get_value_1_ID] =
     (void*)c__get_value_1_impl;
  pTable->_pTable[c__get_value_2_ID] =
     (void*)c__get_value_2_impl;
  pTable->_pTable
     [c__call_in_case_varables_are_equal_ID] =
     (void*)c__call_in_case_varables_are_equal_impl;
  pTable->_pTable
     [c__call_in_case_varables_are_not_equal_ID] =
     (void*)c__call_in_case_varables_are_not_equal
     _impl;
  pTable->_pTable[c__re_entry_ID] =
     (void*)c__re_entry_impl;
  pTable->_pTable[c__my_entry_point_ID] =
     (void*)c__my_entry_point_impl;
  return 0;
}
```

As you can see, each particular module 'knows' its id, is able to retrieve it and also has a functionality to fill up its own matrix. Let us take a look at functionality of `c_dispatcher`:

```
/** c_dispatcher.cpp
 *  (c) George Shagov, 2005*/
#include <stdio.h>
#include "./../include/os.h"
#include "c_dispatcher.h"
#include "./../include/c_fn_s.h"
#include "./../include/c_modules_fndecl.h"
#include "G__ci.h"          /* Cint header file */
#define D__MAX_MODULES 2

typedef struct SModule
{
  char* _sName;
  HINSTANCE _hInstance;
} SModule;
Matrix* g_pMatrix = NULL;
  extern void G__c_setup();/*defined in G__clink.c*/
SModule s_Modules[D__MAX_MODULES] =
{
  { "module0", NULL },
  { "module1", NULL }
};
int s_GetModuleID(const char* sName)
{
  int i=0;
  for (i=0; i<D__MAX_MODULES; i++)
    if (0 == strcmp(s_Modules[i]._sName, sName))
       return i;
  return -1;
}
int g_LoadModule(const char* sModule,
                 const char* sPrefix)
{
  char sDll[128];
  HINSTANCE hModule = NULL;
  C__GETMODULEID pModuleID = NULL;
  C__GETFNTABLE pGetFnTable = NULL;
  C__SETMATRIX pSetMatrix = NULL;
  int nModuleID = -1;
  /* loading module */
  sprintf(sDll, "%s%s_%s.dll", PATH_TO_OUTPUT,
     sModule, sPrefix);
  hModule = LOADLIBRARY(sDll, RTLD_NOW);
  if (!hModule)
  {
    printf("Unable to load library: %s\n", sDll);
    return 1;
  }
  /* getting entry points from loaded module */
  pModuleID =
     (C__GETMODULEID)GETPROCADDRESS(hModule,
     "g_GetModuleID");
  pGetFnTable =
     (C__GETFNTABLE)GETPROCADDRESS(hModule,
     "g_GetFnTable");
  pSetMatrix = (C__SETMATRIX)GETPROCADDRESS(hModule,
     "g_SetMatrix");
  nModuleID = pModuleID();
   /* by this call we are getting function table
      of the module*/
  pGetFnTable(&((*g_pMatrix)[nModuleID]));
  /* setting up global matrix */
  pSetMatrix(g_pMatrix);
  /* freeing the previous module */
  if (s_Modules[nModuleID]._hInstance != NULL)
    FREELIBRARY(s_Modules[nModuleID]._hInstance);
  /* loading the new one */
  s_Modules[nModuleID]._hInstance = hModule;
  return 0;
}
int g_Execute()
{
  char sExecute[128];
  char sPthToScript[128];
  G__value ret;
  sprintf(sPthToScript, "%s %sc_dispath.script.c",
     CINT_COMMAND_STRING, PATH_TO_OUTPUT);
  G__init_cint(sPthToScript); /* initialize Cint */
  G__c_setup();
  sprintf(sExecute,"c__execute();");
  ret = G__calc(sExecute); /* Call Cint parser */
  G__scratch_all(); /* Clean up Cint */
  return 0;
}
HINSTANCE g_GetModule(const char* sModule)
{
  int nModuleID = s_GetModuleID(sModule);
  if (-1 == nModuleID)
  {
    printf("invalid name:%s", sModule);
    return 0;
  }
  return s_Modules[nModuleID]._hInstance;
}
```

There is code related to so-called ‘dispatching’ procedure (`g_Execute`). This procedure calls to the external script by means of `cint`. (`cint` is free C-interpreter, powerful enough and very suitable for this demo), `c-script` calls for `g_LoadModule`.

The code of external script (`c_dispatch.script.c`):

```
#include <stdio.h>
int c__execute()
{
  printf("c__execute\n");
  g_LoadModule("module0", "stub");
  return 0;
}
```

It is possible to understand by now that at the first step we should load our module, and only after – to execute, which is obvious.
Let us take a look at the original script. It’s easy to see that all declarations are performed using `_impl` suffix. It’s intentional. Then, what happens after ‘real’ call to the function, which is not suffixed. There happens a call to a so-called stub function. The stub-function looks like this:

```
extern int g_nModuleID;
extern Matrix* g_pGlobalMatrix;
int c__get_value_1_stub( char* pString)
{
  c__get_value_1_type pFn = (c__get_value_1_type)
     (*g_pGlobalMatrix)[module0_ID].
     _pTable[c__get_value_1_ID];
  printf("c__get_value_1_stub\n");
  return pFn(pString);
}
```

As you can see here, the actual call is delegated to a function whose pointer is in the matrix. Have you got the trick? Simple, isn’t it?

So, it means we can substitute any matrix’s element (which is pointer to a function) to whatsoever (and whenever) we would like to. Let us see how it works now:

The context of the `c_\output` folder (after getting the project built) looks like this:

```
c_dispatcher.dll
c_dispath.script.c
frontend_app.exe
module0_stub.dll
```

And the code of external script (`c_dispatch.script.c`):

```
#include <stdio.h>
int c__execute()
{
  printf("c__execute\n");
  g_LoadModule("module0", "stub");
  return 0;
}
```

It means we will load a stub module. `Module0_stub.dll` is the compiled module of our script. Starting the application and typing **d** (dispatching) we get:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
d
dispatching...
c__execute
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

It means «`out`» module is loaded, executing:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
```

```
executing
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
c__get_value_1_stub
c__get_value_2_stub
c__call_in_case_varables_are_not_equal_stub
++++++after:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0,
   pMyStruct->m_sString: not equal
ret: 11
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

As we can see it works. We have our calls going through the matrix, as we see in stub's output.

Let us here introduce a new entity, let it have a name 'proxy-module'. This is a separate module, which will export exactly the same functions which 'basic' module does, yet these functions will do nothing but delegate the call to 'basic' module.

Let us take a look at the proxy-function:

```
static int s_IsFunctionOverloaded(const char*
sFnName)
{
  char sFile[128];
  FILE* f = NULL;
  sprintf(sFile, "%s%s.c", PATH_TO_OUTPUT, sFnName);
  f = fopen(sFile, "r");
  if (!f)
    return 0;
  fclose(f);
  return 1;
}
int c__get_value_1_impl( char* pString)
{
  if (s_IsFunctionOverloaded("c__get_value_1")) {
    char tmp[128];
    char sPath[128];
    int nRet;
    sprintf(sPath, "%sc__get_value_1.c",
       PATH_TO_OUTPUT);
    G__loadfile(sPath); /* initialize Cint */
    printf("proxy: c__get_value_1_impl -- cint\n");
    sprintf(tmp,"c__get_value_1_impl
       ((void*)0x%08p);", (void*)pString);
    nRet = G__calc(tmp).obj.i; /* Call Cint parser*/
    G__unloadfile(sPath); /* initialize Cint */
    return nRet;
  } else {
    c__get_value_1_type pFn = (c__get_value_1_type)
       GETPROCADDRESS(s_hStubModule,
       "c__get_value_1_impl");
    printf("proxy: c__get_value_1_impl\n");
    return pFn(pString);
  }
}
```

As you can see first it checks for a `file: <function_name>.c` which, if it is present, calls for `cint` to execute it, no – executes the corresponded function from a basic module. That's it.

The context of `c_\output` folder (after getting the project built) looks like this:

```
dispatcher.dll
c_dispath.script.c
cint_module0_proxy.dll
frontend_app.exe
module0_stub.dll
module0_proxy.dll
```

At the first we should change `c_dispath.script.c` file, it should look like this:

```
#include <stdio.h>
int c__execute()
{
  printf("c__execute\n");
  g_LoadModule("module0", "proxy");
  return 0;
}
```

Reloading the module, typing **d** in the console, executing, typing **e**:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
d
dispatching...
c__execute
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl
c__get_value_1_stub
proxy: c__get_value_1_impl
c__get_value_2_stub
proxy: c__get_value_2_impl
c__call_in_case_varables_are_not_equal_stub
proxy: c__call_in_case_varables_are_not_equal_impl
++++++after:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0,
   pMyStruct->m_sString: not equal
ret: 11
-------------
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

So, now we are going through the proxy. Let us see how the trick works. In order to do that we should create a file, let it be `c__get_value_1.c` and put there the functionality of `c__get_value_1` function, like this, for instance:

```
// my_script.cpp : Defines the entry point for the
// DLL application.
#include <stdio.h>
#include "..\\include\\my_structs.h"
#pragma include_noerr <cint_module0_proxy.dll>
int c__get_value_1_impl(char* pString)
```

```
{
  pString[1] = 'X';
  printf("c__get_value_1 ==>> str: %s\n", pString);
  return 2;
}
```

And the output:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl
c__get_value_1_stub
proxy: c__get_value_1_impl -- cint
c__get_value_1 ==>> str: MX string here.
c__get_value_2_stub
proxy: c__get_value_2_impl
c__call_in_case_varables_are_equal_stub
proxy: c__call_in_case_varables_are_equal_impl
++++++after:
nArg: 1, string: MX string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: equal
ret: 11
```

The differences are underlined.

Let us go further and create a file `c__re_entry.c` with the contents:

```
#include "..\\include\\my_structs.h"
#pragma include_noerr <cint_module0_proxy.dll>
int c__re_entry_impl(int nArg, char* pString,
SMyStructure* pMyStruct)
{
  printf("\"I'll not be juggled with.\nTo hell,
     allegiance! Vows, to the blackest
     devil!\nConscience and grace, to the
     profoundest pit!\nI dare damnation. To this
     point I stand,\"\n");
  printf("...for this is script\n");
  int nVar1 = c__get_value_1(pString);
  int nVar2 = c__get_value_2(nArg);
  if (nVar1 == nVar2)
  {
    c__call_in_case_varables_are_equal(pMyStruct);
  }
  else
  {
    c__call_in_case_varables_are_not_equal
       (pMyStruct);
  }
  return 11;
}
```

And the output:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl -- cint
"I'll not be juggled with.
To hell, allegiance! Vows, to the blackest devil!
Conscience and grace, to the profoundest pit!
I dare damnation. To this point I stand,"
...for this is script
proxy: c__get_value_1_impl -- cint
c__get_value_1 ==>> str: MX string here.
proxy: c__get_value_2_impl
proxy: c__call_in_case_varables_are_equal_impl
++++++after:
nArg: 1, string: MX string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: equal
ret: 11
-------------
```

So, the context of output folder by now is this:

```
c__get_value_1.c
c__re_entry.c
c_dispatcher.dll
c_dispath.script.c
cint_module0_proxy.dll
frontend_app.exe
module0_native.dll
module0_proxy.dll
module0_stub.dll
```

Performance

Of cource, using a script instead of native code does mean significant loss of performance, yet there are to things to say:

1. In the systems where performance is a key point, such as real-time systems, no substitution is to be allowed. It means there should not be any dispatcher library and all the calls to be compiled as direct ones and linked during the compilation. In this approach there will not be any loss of performance. Yet in development in QA where possibility for substitution is highly required but performance does not play a significant role, this approach will be applicable.
2. The first rule is too strict. As you can see in the output folder there is `module_native.dll`. What is this? Let us called this library a 'native module'. This is exactly the basic module with the exception that all of the 'local' calls (calls inside one module) are compiled like a direct ones, no stub involved. Any calls which go to the 'external' module should go through the matrix. In my opinion, this decision is more than enough in order to get the performance problem sorted.

Pros and Cons

In brief.

Disadvantages.

- The solving task in general approach, using OOP languages, does look too complicated.
- The build procedure becomes more complicated and additional parsing is required.
- There should be an interpreter supplied
- Read the performance section
- Using C as a script may cause some problems, as C, by default, has direct access to memory and has no mechanism of automatic unwinding, which might potentially cause leaks. Yet, that should be a C-script, not a C, it means that all functions which uses access to memory should be exposed as entities.

Benefits

- Ability to change the business logic run-time.
- A control. Just think what we able to do having all entry-points in our hands.
  - Logging
  - Error handling
  - Parameters tracing

George Shagov

Reviews

Bookcase

The Editor Writes:

Quite a few editions back, Francis made a comment along the lines of the pricing of the books being incorrectly priced due to exchange rates (my paraphrasing). While there are many arguments for and against this point of view, it started me thinking and so I decided to look into not the difference due to exchange rate, but the variety of prices available both from the high street and from the likes of Amazon and Blackwells Online.

There can be a difference of anywhere upto £15 on the same book!

While this has nothing to do with the publisher, it does highlight a problem that I've been facing over the past couple of issues. Namely, is it really worth putting a price on the books? While they may well act as a guide, it is probable that if something is £39.99 at one online shop, it could well be £25.99 on another and £22.99 in your local high street.

Is this really that big an issue? Well, yes it is. As I (and others) have said in the past, the book reviews section is one of the most important parts of C Vu. All the reviews end up on the ACCU website – a site cited frequently for independent reviews. If someone sees a book listed for 40 pounds and it's not a bad book, they may well decide against it and buy one for half the price, which is an utter turkey. Yet if no price is listed, it is probable they will make their mind up on the review rather than price (they can check that quickly enough for themselves).

As a trial run on this issue, I'm dropping the prices. Feel free to let me know if you want them to return and more over, in what format (price range, best buy – you get the idea).

Remember folks – we need YOU to review books. If you're interested in reviewing a book, a full list of books which are currently available can be found on the members only area section of the ACCU website.

Paul

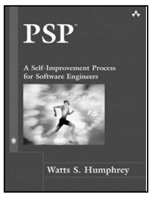
Core Java 2, Volume II - Advanced Features (J2SE 5.0), 7th Ed. by Cay Horstmann & Gary Cornell. ISBN: 0-13-111826-9 Publisher: Prentice HallPages: 1002
Reviewer : Alan Barclay

This is the latest revision of one of the best selling and most respected texts on (some of) the advanced features of the Java programming language. Completely updated for J2SE 5.0 it provides an excellent and concise insight into Threads, Collections, Networking, Database Programming, Distributed Objects, Advanced GUI, Native Methods, XML and others. This book is the perfect follow on to Volume 1 - which covered the core language fundamentals and the most important common library APIs.

Any one of the topics in this volume might command a whole book of their own but for an outstanding introduction for the average user then look no further. Typically the information provided will be perfectly adequate for understanding and making good use of the topic in question.

For example, the chapters on Advanced Swing and AWT measuring just under 300 pages (as big as some other books) contain a wealth of information on advanced subjects including Swing lists, trees, tables and layouts and AWT printing, image manipulation and drag and drop.

Like its twin this volume has a smaller font size than that used in previous editions and contains numerous code examples, high quality illustrations and screen shots, Notes, Tips and Cautions. The only think which I feel is missing from this work is a greater mention of the relevance and usefulness of the JDK API documentation.
Available on the authors' related website are sample chapters, source code download and a not insignificant printing errata (for all editions in the series). Whether to complete your collection or just as a text on advanced features then this book is highly recommended.



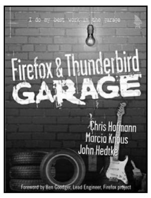
PSP by Watts Humphrey. ISBN 0-321-30549-3. Pub: Addison-Wesley , pp346
Reviewer : Francis Glassborow

The author opens the Preface with: *The record of most development groups is poor, but the record of software groups is particularly bad.* The next paragraph starts: *It is tempting to blame others for our difficulties, but a victim like attitude doesn't solve problems.*

The Personal Software Process described in this book is the author's attempt to provide a process that will result in your improvement. You may wish to argue with the detail but if you want to reject all attempts at improving the individual performance then you do not belong in the world of software development.

One of the benefits of reading this book is that the principles can be applied on an individual basis. Even if you are a lone programmer who has to do it all, applying the principles outlined in this book will lead to progressive improvement in the quality of your work.

Before you make up your mind (and, no, I cannot justify the space needed for a complete over-view of this book) surf over to `www.sei.cmu.edu/tsp/psp`. If you are not already involved in some professional development programme you should give very serious consideration to reading this book and acting on it. If you already have a professional development process in place, you still might be able to refine and improve it by reading this book.



Firefox & Thunderbird Garage by Chris Hoffman et al. ISBN 0-13-187004-1 Prentice Hall
Reviewer : Francis Glassborow

I am not going to write very much about this book because its subject matter is off topic for programmers as programmers. Of course programmers are almost always intensive and possibly sophisticated users of the Internet and so browsers and email applications are going to be of interest to them.

Two of the three authors of this book have been extensively involved in the development of Firefox (a browser) and Thunderbird (an email and news application). The third author (John Hedtke)is a technical author with a wealth of experience and a couple of dozen books to his name as well as hundreds of other pieces of technical writing (manuals, help systems and articles).

In effect, this book is the documentation for the two products from Mozilla. As the products themselves come free, paying something for the documentation seems reasonable.

The contents appear to be accurate and well written. I am rather under-whelmed by some aspects of the presentation of the material and really do not like some of the typefaces being used. Perhaps that is more a reflection on my age than anything that is fundamentally wrong.

If you use either of the products and would like to use them better, this book is worth a look and as computer books go, the price is not too outrageous.

Never Threaten to Eat Your Co-Workers: Best of Blogs Edited by Alan Graham and Bonnie Burton ISBN 1-59059-321-9 Apress
Reviewer: Francis Glassborow

This is a collation of extracts from Blogs written by 24 people. Some people are responsible for nine or ten extracts, others for only one. Of course every item is already out there on the web so there is nothing original in this book. However, there is nothing wrong with collations and this one might make a good Christmas present for someone you know as long as they are not a prim and proper maiden aunt.

Hackish C++ Pranks & Tricks Michael Flenov ISBN 1-931769-38-9 Alist
Reviewer : Francis Glassborow

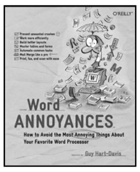
I thought I had already written a review of this book, but if so it fell into a black-hole. Perhaps it is because that is where I believe copies of this book should go.

The author likes to think that he has drawn the line between 'hacking' and 'cracking'. As most of us know, a hacker is a skilled programmer and might reasonably be applied as a complimentary term. Unfortunately the author thinks that playing tricks (or jokes as he likes to call them) on users such as making the clipboard misbehave is OK. Sorry, I do not find such things fun and they can cause thousands of hours of lost time, missed deadlines and deep frustration to the innocent recipient of such jokes. No I am not a spoil sport, but I think that authors should be more responsible and that publishers should filter out such book proposals.

Of course, the books subtitle, 'Pranks & Tricks' does give a warning and there is the argument that understanding abuse allows us to spot it and remove it from other code is not good enough. The author shows no awareness of the cost of pranks in a world with the level of connectivity that we have today.

As most of the books material depends on using some version of MS Windows, the sufferers from his misplaced sense of humour will be those who are unable to handle them. They finish by paying considerable sums of money to others to sort out the mess these jokes have made.

No, I do not like this book, and even if the C++ were impeccable rather than heavily reliant on features of MS Windows (that are not part of C++), I would still not recommend it. I strongly urge that the publisher show a greater degree of responsibility and remove it from sale. If they do not, I can only assume that they do not use machines running Microsoft operating systems.



Word Annoyances by Guy Hart-Davis. ISBN 0-596-00954-2. O'Reilly , pp192

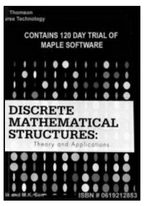
Reviewer: Francis Glassborow

I will keep this short because this book is off topic for programmers, though as many of you use Word you may be interested in this book.

The book is aimed at long-term users of MS Word who are happy to customise the version they are using.

For example, there are four items in the section on menus; Display Full Menus (how to display a full menus immediately, both on a once off basis and always), Prevent Menu Items from Changing Position (how to remove the default setting of personalised menus), Cut Menus Down to Size (how to remove – and add – entries from a menu) and Create a Work Menu. That last item has proved very useful to me because it showed me how to create a menu that contains documents (e.g. mail merge files) that I use often. Relying on the recently used list breaks down when I have just worked through a bundle of documents and thereby lost the direct route to documents that I use frequently.

This is a good book for the experienced user of MS Word who would like to browse through fixes to annoyances (some of which s/he might not even realise could be fixed).



Discrete Mathematical Structures: Theory and Applications by D.S. Malik and M.K. Sen. ISBN: 0-619-21285-3 Thomson

Reviewer: Francis Glassborow

This is very much a book for students who need to know about this area of mathematics. It covers its subject area well. The authors assume that the reader will know enough programming in some appropriate computer language. However the authors give Maple, C++ and Java as their examples of suitable languages. I am not going to dispute the suitability but there Maple is rather different from the other two. The end of chapter programming exercises can vary from trivial to hard depending on which language is being used.

If you are a student you will probably have relatively little choice when it comes to a textbook. If you have decided to spend time on a self-study course your problem will be that you will need to find someone to mentor you because this is a tough subject and it would be easy to go astray. If you are responsible for selecting a textbook for a University course on this subject, this book would be worth considering.

While the authors claim a degree of language independence, you should note that the book comes with a (120-day trial) copy of Maple software. Even the student licence is $125 for a download version and you are not going to finish this book in 3 months (unless you are a very dedicated student). While the programming exercises can be done in almost any computer language, the authors generally expect you to be using Maple and that is the language they use for examples.

Home Networking - The Missing Manual by Scott Lowe. ISBN: 0-596-00558-X O'Reilly , pp265

Reviewer: Francis Glassborow

Once upon a time people homes had just one radio (or wireless), then they had just one TV, then just one computer, one telephone … You get the picture, technology gets cheaper every month with the result that more people can afford it.

Home networking is about much more than just connecting all the family computers to a single router so that all can share a single Internet connection. There are now a multitude of devices that can be networked and the cost encourages people to want to do so. As a computer programmer (well why else are you reading this review) your neighbours, friends and relations will be convinced that you are just the person to help them set up their home network. Well your family probably expects you to start with providing one for the family home. Your company network specialist is fed up to the back teeth with trying to explain it to your colleagues, and probably does not know how to connect your computer to the family TV.

The only weakness in this book is that it concerns itself with Windows and Macintosh based networks (including mixtures of the two) and has nothing to say about Linux based systems. To be honest, that seems fair when considering the target readership.

If you want to set up and maintain a home network you will find this book a valuable companion; it takes you through the process from start to finish. I wish I had read it when I was recently setting up a friend's PC to work with her husband's Mac-based network and Internet connection.

Always Use Protection: A Teen's Guide to Safe Computing by Daniel Appleman ISBN: 59059-326-X Apress , pp264

Reviewer: Francis Glassborow

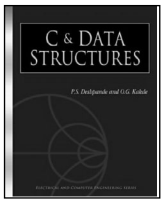
Here is a quote from the 'Introduction to Parents':

> It's not enough for teens to know not to give out their personal information online or to arrange to meet with strangers. Teens today need to understand the fundamentals of computer security. They need to know how to protect their computers and privacy. They need to know about programs they download that might spy on their activities. They need to know how to protect their passwords and why.

My only quarrel with that is that it should not be limited to teens. Every computer user needs the fundamentals of safe computing. It is not just teenagers who download attractive but dangerous programs. Indeed, adults can be much more seriously damaged.

The book is well written and covers the subject well without expecting much by the way of specialist understanding. If you are using a Windows based machine and can surf the Internet you have enough technical knowledge and a relevant machine. Buy a copy for your teenage child, nephew, niece or the daughter of a friend. Then read it before you wrap it up, unless you are an expert on safe usage of Windows based PCs you will learn something useful.

The big downside for this book is that it is Windows specific (actually not too big a downside because a high proportion of malware is targeted at that category of PC. There is also a slight issue with the book being US centric but not one that I find seriously detracting from the value of this excellent book.



C & Data Structures by P S Deshpande & O G Kakde. ISBN 1-58450-338-6 Charles River Media , pp700C

Reviewer: Francis Glassborow

This is one of those books where the authors want to write about implementing data structures (probably because it is a nice well-defined topic found in most university computer science curricula) but decide that they first have to introduce the reader to programming in C. These are two entirely different topics. If you can already program in C you would not want the first 163 pages, and if you cannot then the first 163 pages are almost certainly not enough (even if they were high quality).

Let me address that last comment by showing you a couple of pieces of code (chosen more or less at random from the first part of the book. I promise that even I did not realise how bad it was till I started copying it. Originally it was only such things as the use of 'class' as a variable name – producing an unnecessary incompatibility with C++ – and the loss of several semi-colons.).

On page 139 the author provides the following code:

```
struct address  \\ A
{
  plot char [30], struc char[30];
  city char[30]
}
struct student   \\ B
{
  name char[30];
  marks float;
  struct address adr;
}
main ( )
{
  struct student student1;  \\D
  struct student class[20];  \\E
  class[1].marks = 70;  // F
```

```
   /* remainder suppressed but it is
   just a hard coded initialisation
   of class[1] followed by five
   printf() statements to output the
   contents of class[1] to stdout./*
```

There then follows a section explaining the code. The first item is:

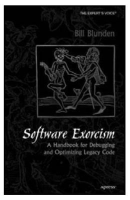
1.Statement A declares the address of a structure containing the members plot, street and city.

I think that that 'explanation' demonstrates beyond doubt that at least some of the errors in the code are the author's direct fault. This book takes the concept of typos to a new height, indeed, assuming the authors can actually program in C and test their code, I cannot imagine how some of the errors occurred during the process of typesetting the book.

I am sometimes willing to be a little forgiving of authors who insist on writing `void main(){`, but that is the least of the problems with this book. The only use I can think of for this book is as a source of practice material for code reviews. Even with all the typos and errors removed this book still uses a style of C coding that would be a disaster in any modern development environment. I would recommend that this book be withdrawn and never published as a second edition. There are already numerous considerably better (though not perfect) books on data structures.

I have tried to find out something about the authors but the book provides no information and I have failed to find any elsewhere. They do acknowledge help from a computer science graduate from IIT Mumbai and from two of their own students. My feeling is that the code looks like student code, so perhaps it is literally their student's code without any correction.

Please could publishers stop wasting trees with this kind of book, quite apart from the waste of paper, readers will learn a dangerous style of programming in C.



Software Exorcism: a Handbook for Debugging and Optimizing Legacy Code by Bill Blunden ISBN: 1-59059-234-4 Apress Reviewer: Francis Glassborow

Interestingly, considering the title, this book starts with a chapter titled 'Preventive Medicine' in which the author addresses techniques for writing code that will be easy to modify (and so easy to correct when a bug is found). Of course, the problem is that that is too late for legacy code. There is lots of sensible advice in this chapter along with explanations that may help in the adoption of those techniques.

The author covers debugging tactics in chapter 2. He starts with ways to ensure that you are dealing with a bug and concludes with methods for tracking maintenance work. We see here that the author takes a down-to-earth attitude to software maintenance. The person doing maintenance is often the newcomer to the group and frequently relatively new to programming. Ideally, we would expect them to get help from other team members; in practice, help may not be so forthcoming.

In chapter 3, under the heading 'Understand the Problem' he offers advice as to how to understand legacy source code. Again, the author's advice is practical and addresses the common problems faced by the newcomer to a piece of legacy code.

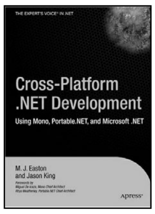
The rest of the book is largely optional. In chapter 4, the author explains how debuggers work. Many readers will find that interesting but it will not do much to help a newcomer improve their skills at maintaining legacy code. Chapters 5 and 6 cover optimization (space and speed). At this stage, the author effectively assumes that you are working on an Intel x86 based machine. The book then concludes with some final words of advice.

The new programmer faced with doing maintenance on legacy code will get a lot of help and virtual support from reading the first three chapters. The fourth chapter may satisfy their curiosity and the last (seventh) chapter contains some good long-term advice. I am much less enthusiastic about the two chapters on optimisation, not least because in my experience programmer attempts at optimisation produces fragile, hard to maintain code. Indeed, I think that chapters 6 & 7 directly conflict with the ideas in chapter 1.

Let me give an example from chapter 6. On page 279 the author recommends that the cases in a switch statement should be given in order of decreasing frequency. As anyone who understands how modern optimising compilers deal with switch statements already knows, that is advice for pessimising performance. All but the simplest of switch statements are implemented as jump tables, or with some other mechanism that does not penalise cases that occur late in the list. However, where a jump table is possible, it relies on the cases being in numerical order.

This book will be useful to many but the section on optimisation needs to be read with a considerable level of suspicion. Remember the old advice about optimising, "Don't". So perhaps the advice for this book should be "Do not read chapters 5 & 6, or at least not yet."

If you are in the target readership (programmers having to maintain legacy C and C++ source code) the book is worth buying for the first half but the second half is, at best, out of date.



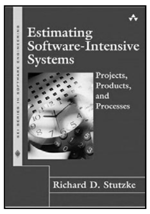
Cross Platform .NET Development by M. J. Easton and Jason King ISBN: 1-59059-330-8 Apress, 527pp Reviewer: Alan Lenton

I was impressed by this book, in spite of the tendency of the authors to use what they obviously believe is the sort of language that will appeal to geeks. The book is a compendium of useful information for anyone considering using .NET as a development and production platform on Windows, Linux and the Mac.

Topics covered range from common cross platform pitfalls, through GUI toolkits, native code and distributed application. The chapter on testing and build strategies, which goes into using NUnit and NAnt I found particularly useful.

I'm not sure how big an audience this book will command, given that its subject is guaranteed to enrage purists on both sides – .NET is the work of the devil or Linux is the spawn of evil – but if you do happen to be working, or considering working in this area, then I would have no hesitation in recommending it.

I should point out one severe limitation on working with cross platform .NET, though. At the moment only C# compilers are available on Mono and Portable .NET (used for the Mac). This tends to put C++ and VB developers at a disadvantage. Of course, you can develop in Visual Studio .NET and move the assemblies over to Mono and Portable.NET, but you will probably find that somewhat restrictive, given the amount of native code used in some of the key .NET libraries. On the other hand, if that's what you need to do, this book will give you a solid basis for choosing which libraries to use, and which to avoid. A useful book.

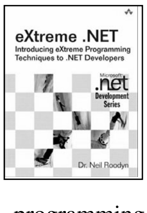


Estimating Software-Intensive Systems by Richard D. Stutzke ISBN: 0-201-70312-2 Addison-Wesley Reviewed by Paul Thomas

Every so often, I stumble across a book that is a lot more interesting than I had expected. Less often still, I find a book full of useful information I didn't even know I needed to know. The name doesn't give an adequate impression of the scope of this book which has more information on the planning side of project management than most managers know exists.

I don't intend to manage any large scale projects myself, but the book seems to fit any size project. Plenty of space is given to modern styles in the agile vein as well as the more traditional planned styles for government-size projects. The layout is clear and logical so the reader can concentrate on the sections relevant to them without missing anything. The internal and external references are comprehensive and appear just where you might need them. The book can be used to learn about estimation techniques, as a detailed reference for practitioners or even as a step-by-step guide (seriously useful).

The scope is staggering. From the ever popular Price-to-Win method (ignore the facts and quote based on what the customer wants to pay) to complex parametric feedback-led estimation techniques, it should be easy to find a model to fit. What makes this book so attractive is that it also serves as an impartial overview of project management styles. Even requirements specification is covered. Basically, anything that can affect estimation is included. If you think about it, that's a large field. I doubt I shall ever look at software metrics as an imposition again. Highly recommended.

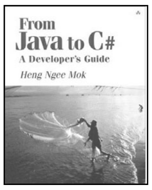


eXtreme .NET by Dr Neil Roodyn ISBN 0-321-30363-6 Addison-Wesley, 298pp, Reviewer: Alan Lenton

This book is intended to introduce the ideas and techniques of eXtreme programming (XP) to .NET developers. It assumes that you already know something about .NET development. The examples assume you are using Visual Studio .NET 2003 and can program in C#. As a C++ programmer I had little difficulty following the code examples, and neither would a Java programmer. The book also covers the use of the tools NUnit and NAnt.

I was impressed by this book, and I think it achieves what it set out to do. There are, to my mind, two particular strengths in this slim volume. The first is that while recognising that there will be times when you can't implement all aspects of XP techniques, the author disposes of the idea of 'nearly' eXtreme programming ("We do eXtreme programming here, except that we don't do...") very smartly. His answer to special case pleading was the best I've ever seen - "Of course it's a special case, that's why you are paid as a professional programmer. We are all special cases. That's no reason not to use test driven development."

The other real strength was that in the material on test driven development (TDD), the author doesn't shy away from the question of GUIs and TDD. GUI tools, especially the rapid application development type, pose serious problems both for separation of interface and core code, and in the testing of the interface itself. Dr Roodyn not only tackles these problems head on, but almost uniquely in my experience understands that developing a professional quality GUI is a eXtreme process in its own right. Recommended.



From Java to C#, A developers guide by Heng Ngee Mok. Pearson Education 2003. ISBN 0321136225
Reviewer: Christer Lofving

In the latest 4-5 years I have mainly been working with different aspects of Java based technology. Now my employer wants me to pick up C#. This means I should fit well in the rather narrow target group this title is aimed for, namely experienced Java programmers "converting" to C#.

If I have to conclude the book in one single word, it would be just "narrow". The text gives you kind of the same feeling as when reading one of these "exam cram" titles, only focusing on what is important for that peculiar certification. (Only the quizzes are missing).

Mr. Heng Ngee Mok seems to be an experienced Java developer, and also being fluent in C#. On top of that he is a good pedagog. In every chapter there is a distinct focus on the languages differences, which are explained in detail.

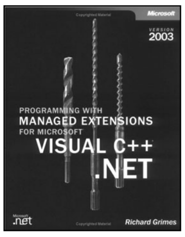
Similarities are just simply pointed out. It is easy (and recommended) to go along and work through the small but enlightening code examples on your own computer while reading.

No Visual Studio is needed, a command window, simple text editor and the C# SDK installed will do just fine.

The plan of the book is recognizable from different "beginners programming books", from very basic concepts to more advanced at the end. Personally I appreciate the ending chapters in this book a bit extra, because they deal with all that stuff I am missing in Java from it's C/C++ heritage; structs, enum, Pointers (they are back in some sense at least) and operator overloading.

Also the premier chapter "Introducing .NET" is one of the best briefs of .NET I have read so far. A plus for Mr. Mok also because he is not wasting any time with ridiculous "religious" aspects of the two languages.

It's from the text context only impossible to find out which "side" he eventually has taken stand on. Despite the authors balanced, humourous writing style it still becomes a little dry and boring at the end to read it through. But as a handy companion for Java to C# converts it fits its purpose well.



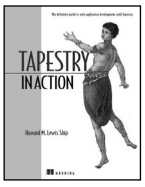
Programming with Managed Extentions for Visual C++ .NET by Richard Grimes. ISBN: 0-7356-1782-1 Microsoft Press
Reviewed by Paul Thomas

The major difficulty I have with this book is actually the subject matter. The managed extensions are a collection of ugly keywords with the double leading underscores Microsoft seems to favour. Anyone unfortunate enough to have used MFC would know what to expect. Let me just say that the managed extensions make MFC look positively engineered by comparison.

To make it even harder to work up an interest, the subject is now dead. It seems that someone was listening to the sound of retching that followed the language's birth and it was promptly drowned. The replacement C++/CLI makes a lot more sense, and you don't have to hold your nose to use it. If you find yourself forced into a position where you have to read this book, quit.

Microsoft press published the book, so it was never going to contain an honest appraisal of the merits of the extensions. Instead, the author's distaste is apparent in the lack of any feeling in the text. Richard Grimes, once a prominent writer on all matters .NET, famously disassociated himself from the cause and I wonder if this book wasn't a factor. Anyone interested in writing might find this book interesting as a study of what happens to style when the love is gone.

Not Recommended.



Tapestry in Action by Howard M Lewis Ship. ISBN: 1-932394-11-7 Manning Publications, 2004,536pp
Reviewed by Andrew Marlow

Highly Recommended

Tapestry is a component-based web development framework.

The book is organised in three sections followed by appendices. The three sections are "Getting Started", "Creating Tapestry Components", "Building Complete Tapestry Applications". The appendices are various references such as building the examples with Ant and do not have to be read as part of the main text. I rate it as highly recommended, not because there is little competition, neither because the author also happens to be the main designer and developer, but because it is well written, attractively presented and very practical in its use of example projects and code fragments.

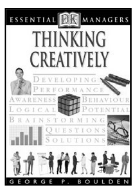
There is a large amount of material to cover. The organisation and presentation of the book as a whole helps here and is is well done. Code fragments are extremely relevant and bear close examination. The balance between prose and code fragments is about right. Algorithms are shown using UML sequence diagrams, giving the book a modern feel. The style is formal without being stuffy and is reasonably concise despite its size.

The book fails to introduce new information in small, easily understandable chunks that build up the learner's knowledge gradually. This can cause the reader to be quickly overwhelmed. For example, each section ends with a section summary and the sections are rather long. A better way would be to start the section with a top-level description of what is to be covered. The author avoids using completely trivial example applications but does not introduce them gently, particularly the first one (the hangman game). This compounds the problem and makes the first section particularly hard to grapple with.

Tapestry makes use of a separate open source project, OGNL (Object Graph Navigation Language). However, the book does not properly introduce it. The book refers to the OGNL web site and provides virtually no other information on what it is or how Tapestry uses it. This has to be gleaned by seeing OGNL in the context of the Tapestry examples. This is hard work for newcomers to Tapestry and OGNL.

It is worth overcoming the hurdle of the first section as the material in section two seems lighter and serves to consolidate material in section one. Some of the examples in section two refer back to the hangman application but in a simple way that makes it easy to see what feature is being presented and why.

Section three introduces a new example application, the virtual library. This application is presented more clearly than the hangman example, so generally the book does get easier as one progresses through it.



Thinking Creatively by George P. Boulden. ISBN 0-7513-3844-3 Dorling Kindersley, £5.99
Reviewer : Ian Bruntlet

I approached this book asking "What is creativity? Am I creative? How has this affected my relationships with co-workers?".

What is creativity? "the process of challenging accepted ideas and ways of doing things" or "seeing ideas or objects in a different context". Looking back at when I've been working, I've tended to be excessively creative, with scant insight or consideration for the consequences.

According to this book, if you have past experience of the class of problem you are tackling, logical/convergent thinking taps our personal experiences. When dealing with new problems, creative/divergent thinking must be employed.

Later on, this book provides a process so that even the most unimaginative reader has a decent chance to be creative and it will improve the creativity of the most radical reader.