overload 96

APRIL 2010 £3

Bug Elimination – Defensive Agile Ruses

We take a fresh look at the economics of software faults

The Predicate Student

We apply logic to problem solving, looking at a very familiar numbers game

The Model Student

We play a game of six integers

Using Design Patterns to Manage Complexity

Simpler programs are more reliable. Omar Bashir shows us how to make improvements by applying design patterns.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 96

April 2010

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

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

Copy deadlines

All articles intended for publication in Overload 97 should be submitted by 1st May 2010 and for Overload 98 by 1st July 2010.

ACCU

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

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

Copyrights and Trade Marks

Dealing with Growing Pains

Expanding your team is never easy. Ric Parkin experiences the recruiting process.

Just recently I've been heavily involved in recruiting two new developers. Although I've helped out in the past with technical interviews, this is the first time I've actually been involved the whole way through the recruitment process. And it is remarkable just how much work it is.

First of all you have to realise you need someone. This could be blindingly obvious, for example if someone has just left and needs replacing, but sometimes is more tricky with many factors involved. Perhaps a new contract has been awarded leading to an upcoming block of work, using a new technology needs someone with relevant experience, or organic growth has allowed a bigger headcount to tackle that backlog of features. In turn, the reasons why you need someone often influences the next two factors – what skills require, and when do you need them by. This can be quite complex, especially when you realise quite how disruptive taking on a new developer and integrating them into a team can actually be. In fact whole books and many Patterns have been written on how to cope, for example SACRIFICIAL LAMB, where you dedicate a team member to get the new people up to speed so that in the short term you only lose their productivity and the rest of the team can continue to develop without being distracted. So you need to balance getting the right skills in the right order with the disruption that can cause. Sometimes the answer can be quite simple, but often you're looking for such a range of skills that it is impossible for a single person to have them all, and so you end up juggling possible subsets to find more realistic roles, eg C++ and unix scripting; html, css and javascript. Depending on upcoming work, there may be obvious roles and a natural order so it that makes sense hiring one at a time to avoid taking a big hit.

Now we know what sort of things we need, we can write a job specification. These usually involve what level of experience in essential skills, and desirable or optional skills, and an idea of what sort of remuneration would be appropriate. You do have to be careful here as employment law changed a few years ago, and some common practices are now illegal, eg asking for a number of years of experience. Your HR department will be involved at this stage, but make sure you get a final check on the wording – I've heard stories of the skills part being edited so that they made no technical sense at all, which is hardly going to encourage people to want to work for you! This last point is too often overlooked – a large part of the recruitment process is selling the company and the role to the candidate. So make sure there are no spelling mistakes.

Next we have to get that job spec out to get applications. But where? Depending on the job, different avenues can be excellent or a waste of time. So for example, I'd probably not bother with Job Centres for a C++ job, and perhaps not even the local papers even in a hi-tech area such as Cambridge. The reasons are that you're potentially looking further afield and for someone who is probably already in a job. A good agency can be an excellent route, but the problem here is that sadly many are not very good at all. But if they do their job properly and learn what you're really looking for and care about, and what their candidates are really interested in, then they do a fantastic job of matching up very suitable people and jobs. There is one particular agency I know that always stood out as the CVs they sent were always worth reading through, in stark contrast to other agencies where they always sending completely unsuitable CVs – eg just last week I got sent a CV for a RF Engineer CV. We're a web services company – that's a no then. Job web pages seem pretty mixed – some seem to work, but some are just too big and unfocused. In my area we have a good 'networking' site and jobs email that has been very useful over the years. And of course, targeted mailing lists such as accu-contacts are good at getting to a lot of very good people (although they therefore tend to be well looked after and settled in their current job...)

Now we get the CVs flooding in. Sadly they will be of variable quality, and you need to filter them so you don't waste too much time. Some will be badly written and full of typos which could be taken as an indicator of sloppiness – it is their most important document after all, and everyone should get someone to proof read it or at least run a spell checker over it. In this age of the internet and international mobility, you can often get CVs from many parts of the world. Interestingly different cultures seem to have different approaches to CV writing – some seem to list every tiny detail of the technologies they've used (or just heard about!), others are remarkably terse and shy away from 'selling' themselves. Whatever the style, what you're actually trying to do is get an idea of the person behind the CV, what their abilities actually are and can they do the job and contribute to the team. This is a non-trivial exercise, and at this stage you should really be just be cutting down the numbers of applicants to a manageable amount.

From here on, it is very important to do several things. One is to try very hard to be fair to each applicant so they have an opportunity to be seen in their best possible light. This includes things such as having interview 'scripts' so we ask roughly similar questions, have multiple people involved at each stage, and have the same people involved for each of the candidates. This tries hard to achieve consistency between candidates. It is also important to keep notes, partly so that you can remember what people said, as when you're doing five interviews, it easy to lose track of who said what. It's also important as it leaves a paper trail of what happened so your decisions can be justified.

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

At this stage there are many different techniques for finding out more about the applicants. We sometimes send them a quick programming question or two for them to answer. The idea is that it should be nothing very onerous, and is more about finding out their attitude and understanding rather than syntax details. An example we've used is a simple string class with poor resource handling, and get them to comment on it. This is mostly useful for weeding out those people who aren't really serious about their applications – the sort who use a scatter-gun approach to sending out CVs.

Then we have a quick 10–15 minute telephone chat. This shouldn't go into much detail, but allows us to explain what the company does, what sort of thing we're looking for and find out a bit more about the person, perhaps by getting them to talk about something on their CV. This last can really show whether they truly understand what they've done, and can communicate it to someone else (after all, I say that a large part of software development is communication, whether it's finding out requirements, writing a good bug report, or expressing a design in code). At the end of this you know whether the applicant is still interested in the role and whether it's worth going ahead.

At this point we sometimes set a quick home programming task (or get them in if they haven't the facilities at home). This should be self-contained, take no more than a few hours, and yet be complex enough that they have to make some interesting design decisions and use various language and library facilities. This is remarkably hard to set well, so make sure you get feedback from them about it, and try it yourself. When you get the results, get several people to go over them to make comments on the coding style, bugs, alternative approaches etc. This can very very revealing about the person's background and programming habits, for example whether they learnt C++ in the 90s before techniques such as RAII and exception-safety were widely understood. Unless the code is really bad, we then get them in for a face-to-face interview.

This usually lasts a couple of hours – more then that and people tend to get too tired (including the interviewer – it is quite hard work listening carefully and taking notes), so if you want to do more, a second interview may be the best option. In this we have various parts, with a general chat with someone from HR, a development manager to talk about methodologies, some technical discussions, and a session discussing their programming test to find out the 'why' of their decisions, what their attitudes are, and what sort of things they thought about but rejected. It's very important here to not treat this as an opportunity to catch the candidate out, or to show off your own technical knowledge – you're here to find out about them.

Once that's done, it's a good idea to have a quick meeting immediately with the people who have been involved in the interviewing. Collect people's opinions, and see if there's a consensus. It's usual pretty obvious if it's a Yes or a No, but sometimes there will be a split. If there's even a single strong No vote then that is probably a good reason to reject – a bad hire can be extremely costly, so better to reject the occasional good candidate. But if it's more nuanced, then perhaps a further interview may clarify – a good question to ask is what else people would like to find out and what questions would help. If people can't answer that, it can show that there may not be any point going further.

Finally, after HR has negotiated on salary and an offer been made and accepted, you must then make sure that there's the right equipment and a place to sit. I strongly recommend having a rethink of everyone's locations at this stage, to make sure they will be near the people that they will talk to most, and can help them get up to speed. Again, this is an issue of arranging your team so that you foster the desired communications channels, as per Conway's Law [Conway]. And make sure you've decided on a suitable project to help them find their feet and become productive as soon as possible, ideally something small enough that they don't have to understand too much too soon.

This whole procedure takes up a lot of effort. I'd estimate to fill each post it took about a third of my time over a three week period, and a couple of days input each from the rest of the development team (and we're a small company where we can make decisions pretty quickly as we don't need to get input from lots of people.) But because of that effort we now have two good new team members working productively – well worth it in my eyes.

Attention!

We have received word from station X: After the successful decryption of their mechanical rotary cipher at last year's conference, our enemy has adopted an electronic encryption system.

One of their machines has fallen into our hands and our boffins are currently busy examining its circuitry. At this year's conference we shall once again call upon the ACCU to assist in our decryption efforts. Those who bravely step forward with a donation to Bletchley Park and the Museum of Computing, will compete for the kudos of being crowned this years Crypto Champion in these very pages.

Dismissed!

The C++0x Standard draft

In a major piece of news, the final committee draft for the upcoming C++0x has been accepted. This has taken longer than originally hoped, and has only been achieved by taking out some controversial features and a lot of effort. What remains now is a few months of review and minor fixes, voting by national bodies, and the final text should be voted on next March. It will be interesting to see how fast compilers start to implement the new features. For full details, see Herb Sutter's blog and the links from there [Sutter].

References

[Conway] http://www.melconway.com/law/index.html

[Sutter] http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/

The Model Student: A Game of Six Integers (Part 2)

What are the properties of the Numbers Game? Richard Harris continues his analysis.

In the first part of this article we introduced the Countdown numbers game [Countdown]. Part of the longest running game show on UK television and one of the longest running in the world, it involves picking 6 numbers from a choice of 4 large and 20 small numbers, being the integers 25, 50, 75 and 100 and 2 of each of the integers from 1 to 10 respectively.

The contestants must then seek to construct a formula using the binary operators of addition, subtraction, multiplication and division with each of these 6 numbers no more than once, and with no non-integer intermediate values, that evaluates to a randomly selected target between 1 and 999.

Reverse Polish Notation

In order to simplify the automatic generation of formulae, we introduced Reverse Polish notation, or RPN, a notation for arithmetic formulae supremely suited to programmatic manipulation.

Unlike the more familiar infix notation, RPN places operators immediately to the right, rather than between, their arguments. For example the infix expression 3+4 would be written as 3 4 + in RPN.

RPN utilises a stack to keep track of the intermediate values in a calculation, and in doing so makes explicit the order in which operators are applied, making parentheses redundant.

Every time a number appears in the input sequence it is pushed on to the stack. Every time an operator appears it pops the arguments it requires off of the stack and pushes its result back on to it. If there are not enough arguments on the stack for an operator, or if there is more than one value on the stack at the end of the RPN formula, an error occurs.

For example, Figure 1 illustrates the state of the stack after each term in the RPN formula 3 4 × 2 +

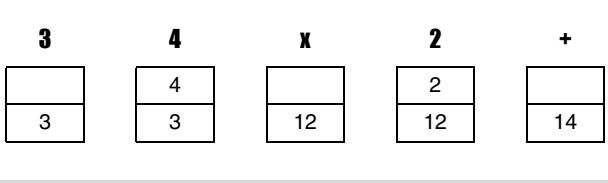

Figure 1

Formula templates

Rather than enumerate the set of valid formulae in any possible Countdown numbers game directly, we instead decided to generate templates into which we could substitute operators and arguments. Acting as place-holders for the operators and arguments we used the *o* and *x* symbols respectively.

x	xxo	xxoxo	xxxoo	xxoxoxo	xxoxoxoo
xxxooxo	xxxoxoo	xxxxooo	xxoxoxoxo	xxoxoxxoo	xxoxxooxo
xxoxxoxoo	xxoxxxooo	xxxooxoxo	xxxooxxoo	xxxoxooxo	xxxoxoxoo
xxxoxxooo	xxxxoooxo	xxxxooxoo	xxxxoxooo	xxxxxoooo	

Figure 2

We described an algorithm for generating the set of all formula templates with up to a given number of arguments in which, starting with a single *x* symbol, recursively replaced *x* symbols with the symbols *xxo*. Since both the former and the latter result in a single value on the stack, we cannot therefore create an invalid formula by doing so.

In general, this approach could generate some formula template more than once, so we kept track of the generated templates with a `set` of `string`s. The declaration of the `all_templates` function that implements this algorithm is:

```
std::set<std::string> all_templates(
  size_t arguments);
```

Since each substitution introduces 1 additional argument, we can ensure that we terminate the recursion when we have exhausted the available arguments by subtracting 1 from their number with each substitution and stopping when we reach 0.

Figure 2 gives the complete set of formula templates for up to 5 arguments in order of increasing length, as calculated using these functions.

Evaluating RPN formula templates

We concluded by implementing a mechanism by which we could evaluate a given formula template for a given set of operators and arguments. The first step was to implement an abstract base class to represent arbitrary RPN operators and concrete classes derived from it to represent the four valid binary operators of the Countdown number game. The base class and the declarations of the operators are provided in Listing 1.

Note that since the standard `stack` class does everything we require of an RPN stack, we very sensibly, or very lazily, opted to use it rather than implement our own.

The definitions of the four operator classes are, with the exception of their names, identical, so we provide just the definition of `rpn_divide` in Listing 2.

Recall that we gave the operators themselves the responsibility for issuing errors in the event that there are not enough arguments on the stack since this allows us to implement operators that require any particular number of arguments should we have cause to do so.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

we gave the operators themselves the responsibility for issuing errors in the event that there are not enough arguments on the stack

```
template<class T>
class rpn_operator
{
public:
  typedef std::stack<std::vector<T> >  stack_type;

  virtual ~rpn_operator();
  virtual bool apply(stack_type &stack) const = 0;
};

template<class T> class rpn_add;
template<class T> class rpn_subtract;
template<class T> class rpn_multiply;
template<class T> class rpn_divide;
```

Listing 1

```
template<class T>
class rpn_divide : public rpn_operator<T>
{
  public:
    virtual bool apply(stack_type &stack) const;
};
```

Listing 2

The **rpn_divide** operator is something of a special case in that it is undefined for some arguments. Specifically, for floating point calculations the second argument must be non-zero and for integer calculations it must wholly divide the first.

It is, therefore, the only one of the four operators that might return **false** from its **apply** member function, indicating that an invalid intermediate value resulted from its application.

We then implemented a simple structure to represent both the validity and the value of the result of a calculation. Given in Listing 3, the **rpn_result** structure is considered valid if it is constructed with a value and invalid if it is not.

```
template<class T>
struct rpn_result
{
  rpn_result();
  explicit rpn_result(const T &t);

  bool valid;
  T    value;
};
```

Listing 3

```
template<class T>
rpn_result<T>
rpn_evaluate(const std::string &formula,
  const std::vector<rpn_operator<T> const *>
  &operators,
  const std::vector<T> &arguments);
```

Listing 4

Finally, we implemented a function that, given a formula template represented by a string of **o** and **x** characters, a **vector** of pointers to **rpn_operators** and a **vector** of arguments, would substitute the latter pair into the former to produce a result.

The **rpn_evaluate** function, the declaration of which is shown in Listing 4, iterated over the formula template string, pushing the current argument on to the stack of the current term is an **x**, and applying the current operator if it is an **o**. We assert the correctness of the calculation by throwing exceptions if there aren't enough arguments or operators, or if there is not exactly one value on the stack at the end.

Whilst this approach isn't particularly useful as a general purpose RPN calculator due to the separation of the formula template, the operators and the arguments, it is particularly well suited to the programmatic evaluation of formulae for precisely the same reason.

Counting the number of formulae

We noted that in order to fully investigate the statistical properties of the Countdown numbers game we should need a means to enumerate every possible formula that might be expressed.

Recall that we proved that the total number of formulae that it is possible to construct with the 4 binary operations and 6 of the 24 possible arguments was

$$ {}^{24}C_6 \times \sum_{i=1}^{6} T_i \times 4^{i-1} \times {}^{6}P_i $$

where ${}^{24}C_6$ is the number of ways in which we can choose 6 from 24 items when their order is not important, T_i is the number of formula templates with i arguments, $4i$-1 is the number of sets of binary operators that can appear in a formula with i arguments, and ${}^{6}P_i$ is the number of ways in which we can choose i from 6 items when their order *is* important.

In light of this result, we concluded that in order to generate every possible formula we would need to enumerate the ${}^{24}C_6$ combinations of selected numbers, the set of templates with up to 6 arguments, the 4^{n-1} sets of operators required by formulae with n arguments and the ${}^{6}P_n$ permutations of n out of 6 arguments.

We closed with the suggestion that the standard **next_permutation** function might be an excellent example to follow and it is at this point that I mean to resume our study.

We can improve matters somewhat by first checking whether or not these elements are already sorted

```
template<class BidIt, class T>
bool
rotate_state(BidIt it, const T &lb, const T &ub)
{
  bool last = ++*it==ub;
  if(last) *it=lb;
  return last;
}

template<class BidIt, class T>
bool
next_state(BidIt first, BidIt last,
   const T &ub, const T &lb = T())
{
  BidIt it = last;
  while(it!=first && rotate_state(--it, lb, ub));
  return first!=last && (it!=first || *it!=lb);
}
```

Listing 5

Evaluating every formula for a given template

Fortunately for us, we solved the second part of this task during our analysis of knots in a previous study [Harris08]. The `rotate_state` and `next_state` functions which we used to iterate through every possible state of a set of sequential integer-like values are presented again in Listing 5.

Since iterators behave in a sequential integer-like fashion, we can place instances of our four RPN operators in a container and use `next_state` with the iterators ranging over it to generate every possible set of operators.

The third part is a little more complicated. The standard library already provides us with a function to iterate through every permutation of a strictly ordered set of values in `next_permutation`. Unfortunately, it does not give us a mechanism for iterating through the permutations of the subsets of such a set, so we shall have to knock one together ourselves.

Since `next_permutation` enumerates the permutations of the elements in the iterator range passed to it in lexicographical order, we can trick it into jumping past permutations of the elements not in the subset of interest. We do this by ensuring that they are in their lexicographically final order, thus forcing the next permutation to be the one that enumerates the next subset. Specifically we sort the spare elements in decreasing order, as illustrated in Listing 6.

This new version of the `next_permutation` function thus enumerates the permutations of `mid-first` elements from the iterator range `first` to `last` in lexicographical order. Listing 7 shows how we would use this function to enumerate the set of permutations of two elements from a vector containing the integers 0 to 3.

The output of this code snippet is given in Figure 3 and, as you can see, it correctly lists the permutations of two elements in lexicographically increasing order.

```
template<class BidIt>
bool
next_permutation(BidIt first,
   BidIt mid, BidIt last)
{
  std::sort(mid, last);
  std::reverse(mid, last);
  return std::next_permutation(first, last);
}
```

Listing 6

```
std::vector<long> seq;
for(size_t i=0;i!=4;++i)  seq.push_back(i);

std::vector<long>::iterator first = seq.begin();
std::vector<long>::iterator mid   =
seq.begin()+2;
std::vector<long>::iterator last  = seq.end();

do
{
  std::vector<long>::iterator it = first;
  while(it!=mid)  std::cout << *it++ << " ";
  std::cout << std::endl;
}
while(next_permutation(first, mid, last));
```

Listing 7

Note that since we have implemented our new `next_permutation` function in terms of the standard one, we are subject to its restrictions. Specifically, the sequence cannot contain two or more elements with equivalent ordering. This condition shall not necessarily be met by the numbers we pick in the Countdown numbers game since there are two of each of the integers from 1 to 10. We shall sidestep this by instead

```
0 1
0 2
0 3
1 0
1 2
1 3
2 0
2 1
2 3
3 0
3 1
3 2
```

Figure 3

subsets that are distinguished only by the elements that they contain and not by the order of those elements

```
template<class BidIt>
bool
sorted(BidIt first, BidIt last)
{
  BidIt next = first;
  if(next!=last)  ++next;

  while(next!=last && !(*next<*first)) {
     ++first; ++next;}
  return next==last;
}

template<class BidIt>
bool
next_permutation(BidIt first, BidIt mid,
   BidIt last)
{
  if(!sorted(mid, last))  std::sort(mid, last);
  std::reverse(mid, last);
  return std::next_permutation(first, last);
}
```

Listing 8

considering permutations of iterators from a sequence of the selected numbers, which will guaranteed to be unique.

The major issue I have with this new overload is that it sorts the spare elements at every step, despite the fact that in a series of calls to it they are guaranteed to be sorted for all but the first call.

We can improve matters somewhat by first checking whether or not these elements are already sorted. Assuming we have *n* unused elements, this is at worst an *n*-step operation; hardly ideal, but certainly better than the worst case of the call to **`sort`** which is greater by a factor of the base 2 logarithm of *n*. We shall do this using another function, **`sorted`**, as illustrated in Listing 8.[1]

Whilst I'm sure that with some thought we could come up with something more efficient, the relative simplicity of this approach is enough to convince me to stick with it.

Enumerating the choice of numbers

Well, we're nearly ready to start investigating the properties of the Countdown numbers game in detail.

All that remains to do is to implement a mechanism for enumerating every way in which we might pick 6 numbers from the available 24. We don't care about their order since we shall be iterating through the permutations of the chosen numbers when we evaluate each function template. Hence we shall need to enumerate the combinations of 6 out of the 24 numbers.

1.The upcoming new C++ standard library will in fact have an **`is_sorted`** function, but for now we must write our own.

We might as well shoot for a general purpose implementation along the lines of our **`next_permutation`** overload, as illustrated by the following declaration:

```
template<class BidIt> bool next_combination(
   BidIt first, BidIt mid, BidIt last);
```

This function should transform the range **`first`** to **`mid`** to the lexicographically subsequent combination of **`mid-first`** elements from the range **`first`** to **`last`**. Furthermore, like **`next_permutation`**, it should, upon having exhausted the set of combinations, leave the range in a sorted state and return **`false`**.

An algorithm for enumerating combinations

Having trawled through my personal library for an algorithm to enumerate combinations in such a fashion, I unfortunately emerged empty handed. We shall therefore have to don the mantle of such luminaries as Knuth and Cormen *et al* and come up with an algorithm of our own.

The first thing we should note in seeking an algorithm to enumerate the combinations of a set is that combinations are subsets that are distinguished only by the elements that they contain and not by the order of those elements, as permutations are. To simplify their enumeration it therefore makes sense to keep the elements of the combinations in a particular order; specifically in increasing order.

To describe the algorithm, I shall use the example of enumerating the combinations of four of the integers from 1 to 6. Since we want the algorithm to act as much like the standard **`next_permutation`** function as possible, and hence work for any elements that can be strictly ordered, we must avoid all operations other than assignment, or strictly speaking swapping, and the less than operator.

We shall represent a permutation as a sequence of elements, with a bar separating the elements that are in the combination from those that are not. For example, the lexicographically first combination of four from six integers shall be represented as

1 2 3 4 | 5 6

The clue that nudges us in the direction of an effective algorithm is that fact that if we rotate the last three elements of the sequence one to the left, we generate the lexicographically subsequent combination

1 2 3 5 | 6 4

Doing so again generates the next combination

1 2 3 6 | 4 5

At this point applying the same operation will return us to a lexicographically earlier combination; the very first as it happens. Fortunately, we can identify that this is about to happen since the element after the bar is now less than the element before it.

What we need to do next is return the sequence to its original order and rotate the last four elements, yielding

1 2 4 5 | 6 3

English is a pretty damn inconvenient language in which to describe algorithms, which is of course why Algol was invented

And herein lays the basic principal by which our algorithm shall algorithmicate.

If the rotation of the last element in the combination would yield a lexicographically earlier sequence, we iterate backwards through the combination seeking an element for which we can rotate the following elements to return it and them to their lowest lexicographical order. We then rotate both it and them together to yield the lexicographically subsequent combination.

As an example, let us consider the combination

2 4 5 6 | 1 3

As we iterate back through the elements of the combination we find that the last element for which we can rotate subsequent the elements into an increasing order is in fact the first. Choosing the lexicographically earliest sequence we can thusly generate we have

2 3 4 5 | 6 1

Now rotating the first and subsequent elements yields

3 4 5 6 | 1 2

which is, as can easily be confirmed, the lexicographically next greatest combination.

The full enumeration of our example combinations would proceed as illustrated in Figure 4, in which we differentiate the valid combinations from the intermediate steps with a bold font and we identify the elements that we shall need to rotate by underlining them.

1 2 3 4 | 5 6
1 2 3 5 | 6 4
1 2 3 6 | 4 5
1 2 3 4 | 5 6
1 2 4 5 | 6 3
1 2 4 6 | 3 5
1 2 4 5 | 6 3
1 2 5 6 | 3 4
1 2 3 4 | 5 6
1 3 4 5 | 6 2
1 3 4 6 | 2 5
1 3 4 5 | 6 2
1 3 5 6 | 2 4
1 3 4 5 | 6 2
1 4 5 6 | 2 3
1 2 3 4 | 5 6
2 3 4 5 | 6 1
2 3 4 6 | 1 5
2 3 4 5 | 6 1
2 3 5 6 | 1 4
2 3 4 5 | 6 1
2 4 5 6 | 1 3
2 3 4 5 | 6 1
3 4 5 6 | 1 2
1 2 3 4 | 5 6

Figure 4

Implementing our algorithm in C++

Now English is a pretty damn inconvenient language in which to describe algorithms, which is of course why Algol was invented. That said, this is ostensibly a C++ article and so, Algol be damned, we shall formally describe our algorithm in C++.

Before we can proceed to an implementation of the `next_combination` function, however, we shall need some helper functions.

As we did in our `next_permutation` overload, we shall need to check whether or not the input sequence represents a valid combination and if not, rearrange its elements so that it does. Again, the reason for performing the check first is that it is computationally less expensive than rearranging the elements and the sequence will always be a valid combination after the call to the function.

Specifically a valid combination will be sorted between `first` and `mid`, and from `mid` to `last` will be a rotation of the sorted elements such that either `mid` contains the next largest element than the one that precedes it, or that every element in the range is smaller than it.

Listing 9 illustrates the function that we shall use to check whether or not the sequence is a combination.

We make a sequence into a combination by sorting the elements between `first` and `mid` and the elements between `mid` and `last` before finally rotating the latter so that the smallest of them that is greater than the last of the former is in `mid`.

```
template<class BidIt>
bool
is_combination(BidIt first, BidIt mid, BidIt
last)
{
  if(mid==first || mid==last) return sorted(
     first, last);
  if(!sorted(first, mid))     return false;

  BidIt prev = mid; --prev;
  BidIt next = mid; ++next;

  while(next!=last && !(*next<*mid)
     && *prev<*mid) {++mid;++next;}

  if(next==last) return true;
  if(*mid<*prev) return false;
  ++mid; ++next;
  while(next!=last && !(*next<*mid)
     && !(*prev<*mid)) {++mid;++next;}

  return next==last;
}
```

Listing 9

I'm not entirely convinced that this is the most efficient possible algorithm

```
template<class BidIt>
void
make_combination(BidIt first, BidIt mid,
   BidIt last)
{
  std::sort(first, mid);
  std::sort(mid, last);

  BidIt prev = mid;
  if(prev!=first)
  {
    std::rotate(mid, std::upper_bound(
       mid, last, *--prev), last);
  }
}
```

Listing 10

Listing 10 provides the definition of the function that rearranges the elements into valid a combination.

The final helper function that we shall need is one to find the smallest element in the range **mid** to **last** that is larger than the last in the range **first** to **mid** that is smaller than any of them. We cannot use **upper_bound** this time since the range will not, in general, be sorted. Listing 11 illustrates the function that we shall use to locate the element in question.

We are now ready to implement the **next_combination** as shown in Listing 12. As you can clearly see, this is simply our algorithm directly translated into C++, with a few extra conditions to cover the corner cases of combinations of none or all of the elements in the sequence.

Note that, like **next_permutation**, this function can't cope with sequences that contain multiple elements with equivalent orderings and will terminate before having enumerated the full set of combinations for such sequences.

```
template<class BidIt, class T>
BidIt
min_greater(BidIt first, BidIt last, const T &t)
{
  BidIt result = last;
  while(first!=last)
  {
    if(t<*first && (
       result==last || *first<*result))
       result = first;
    ++first;
  }
  return result;
}
```

Listing 11

```
template<class BidIt>
bool
next_combination(BidIt first, BidIt mid,
   BidIt last)
{
  if(!is_combination(first, mid, last))
  {
    make_combination(first, mid, last);
  }

  if(mid==first || mid==last) return false;

  BidIt next = mid;
  BidIt prev = mid; --prev;

  if(!(*prev<*next))
  {
    BidIt target = last;
    while(target==last && prev!=first)
    {
      target = min_greater(mid, last, *--prev);
    }

    if(target==last)
    {
      std::rotate(first, mid, last);
      return false;
    }

    next = prev; ++next;
    std::rotate(next, target, last);
  }

  std::rotate(prev, next, last);
  return true;
}
```

Listing 12

Once again, I'm not entirely convinced that this is the most efficient possible algorithm. However, given that checking whether or not an input sequence of *n* elements is a combination requires at best *n* operations, and that our algorithm requires at worst a fixed multiple of *n* operations, I suspect that it could only be bettered by some constant factor.

Figure 5 illustrates the result of enumerating the set of combinations of four from the integers 1 to 6 using this function.

But does it actually work?

Now, it's all well and good describing, implementing and demonstrating our algorithm, but we cannot be certain that it will work for all possible sets of combinations until we get on with the rather more tedious business of proving it. Fortunately for us, it's not too onerous a task.

this has been a somewhat handwaving attempt at a proof

```
1 2 3 4
1 2 3 5
1 2 3 6
1 2 4 5
1 2 4 6
1 2 5 6
1 3 4 5
1 3 4 6
1 3 5 6
1 4 5 6
2 3 4 5
2 3 4 6
2 3 5 6
2 4 5 6
3 4 5 6
```

Figure 5

The key point is in fact addressed by the `is_combination` function. If we can demonstrate that whenever a sequence satisfies this condition, our algorithm generates the lexicographically subsequent combination, and that the sequence representing it also satisfies the condition, then our work is done.

Representing a combination of r from n elements as

$$x_0\, x_1\, x_2 \ldots x_i \ldots x_{r-1} \mid x_r \ \ldots x_j \ldots x_{n-1}$$

where the ith element is the last that is smaller than any of those in the range r to n-1 and the jth is the smallest in that range that is larger than the ith.

Now if this is the very last combination, the elements r to n-1 must be smaller than the elements 0 to r-1 and both ranges must be in increasing order. Rotating the sequence so that the latter precede the former therefore returns us to the first combination.

If i is equal to r-1, then since our condition is assumed to be satisfied, j must be equal to r. Applying our algorithm therefore results in the sequence

$$x_0\, x_1\, x_2 \ldots x_r \mid x_{r+1} \ldots x_{n-1} \ldots x_{r-1}$$

This is trivially the lexicographically subsequent combination and must satisfy our condition. Indeed, if it did not, it would imply a contradiction since the r-1th element would have to be greater than both the rth and the n-1$^{\text{th}}$.

Otherwise, we can be certain that the i+1th to the r-1th elements must all be larger than the jth to the n-1th and that both sequences must be sorted in increasing order.

The first rotation we perform must therefore return the sequence to the lexicographically smallest with the ith element in its given position. Since this previous combination could not have been the lexicographically last, there must now be at least 1 element larger in the range r to n-1 that is larger than those in the range 0 to r-1. Most importantly, the rth element must be larger than the r-1th. The second rotation therefore results in the lexicographically subsequent combination. Furthermore, it too must satisfy our condition, since the before the second rotation the elements after the ith were in increasing order up to some element after the r-1th and the elements that follow those, if any, must be smaller than the ith.

QED.

I *think*.

I'll freely admit that this has been a somewhat handwaving attempt at a proof, and that I wouldn't be terribly surprised to learn that I've missed a corner case or two. Nevertheless, I believe that we can be reasonably confident that our algorithm will correctly enumerate any set of combinations and so we can happily use it to analyse the Countdown numbers game.

Putting it all together

Note that our formula template evaluation function expects `vector`s of pointers to `rpn_operator` objects and `long` integers, and we shall be using our choice enumeration functions with sequences containing iterators into these in order that their elements exhibit the strict ordering and integer-like behaviour that they require.

We shall therefore need a pair of functions to convert to and from `vector`s of values and `vector`s of iterators, as provided in Listing 13.

To simplify the gathering of various forms of information regarding the numbers game, we shall take a leaf out of the STL's book and build a function template like the standard `for_each` to do the actual iteration and forward on the calculation to a function passed in as an argument.

We shall build this in two parts, the first of which iterates through the combinations of selections of numbers to work with, and the second of

```
template<class FwdIt>
void
fill_iterator_vector(FwdIt first, FwdIt last,
                     std::vector<FwdIt> &vec)
{
  vec.resize(std::distance(first, last));
  std::vector<FwdIt>::iterator out = vec.begin();

  while(first!=last)  *out++ = first++;
}

template<class FwdIt, class T>
void
fill_dereference_vector(FwdIt first, FwdIt last,
                        std::vector<T> &vec)
{
  vec.resize(std::distance(first, last));
  std::vector<T>::iterator out = vec.begin();

  while(first!=last)  *out++ = **first++;
}
```

Listing 13

we shall be able to examine the properties of both every possible game and any specific game

which iterates through every possible game that can be played with those numbers. By doing so we shall be able to examine the properties of both every possible game and any specific game.

Illustrated in Listing 14, the first of these functions expects four arguments. The **first** and **last** arguments represent the numbers from which we can choose our arguments, the **args** argument the number that we shall choose and finally the **f** argument the function we wish to apply to the result of every calculation.

This function simply iterates through the combinations of **args** from our available numbers, passing the iterator range representing each of them together with the function to be called on to the second version of the function.

The second, significantly longer, function is illustrated in Listing 15. Note that this enumerates the permutations of arguments for each formula template using the iterator range representing the current choice of numbers. Since the **next_permutation** function will eventually return it to lexicographical order, we can be sure that at the end of this function, the full range into which they point will once again represent a valid combination.

The operation of this second function is reasonably straightforward. Firstly, it constructs the full set of formula templates and a collection of the available operators. It then iterates through every formula template, argument choice and operator choice, calculating the result of each function thus generated using temporary buffers into which the argument and operator choices are dereferenced, and calls our statistics gathering function for each valid result.

Unfortunately, however, I have spent so much time developing the tools with which we shall perform our analysis that I have run out of time in which we might actually do it.

So we shall have to wait until the next instalment before we can answer the question that we originally posed.

Until then, dear reader, farewell. ■

```
template<class BidIt, class Fun>
Fun
for_each_numbers_game(BidIt first, BidIt last,
   size_t args, Fun f)

{
  std::vector<BidIt> number_choice;
  fill_iterator_vector(first, last,
     number_choice);

  if(args>std::distance(first, last))
  {
    throw std::invalid_argument("");
  }

  std::vector<BidIt>::iterator first_choice
     = number_choice.begin();
  std::vector<BidIt>::iterator mid_choice
     = first_choice+args;
  std::vector<BidIt>::iterator last_choice
     = number_choice.end();

  do
  {
    f = for_each_numbers_game(first_choice,
       mid_choice, f);
  }
  while(next_combination(first_choice,
     mid_choice, last_choice));

  return f;
}
```

Listing 14

Acknowledgements

With thank to Keith Garbutt for proof reading this article.

References and further reading

[Countdown] http://www.channel4.com/programmes/countdown

[Harris08] Harris, R., The Model Student: A Knotty Problem, *Overload 84*, 2008.

```
template<class BidIt, class Fun>
Fun
for_each_numbers_game(BidIt first_number_choice,
                      BidIt last_number_choice,
                      Fun f)
{
  typedef
     rpn_operator<long> const * operator_type;
  typedef
     std::vector<operator_type> operators_type;
  typedef
     operators_type::const_iterator
     operator_iterator;

  const size_t args =
     std::distance(first_number_choice,
     last_number_choice);
```

Listing 15

```
operators_type operators(4);

const rpn_add<long>      add;
   operators[0] = &add;
const rpn_subtract<long> subtract;
   operators[1] = &subtract;
const rpn_multiply<long> multiply;
   operators[2] = &multiply;
const rpn_divide<long>   divide;
   operators[3] = &divide;

std::vector<operator_iterator> operator_choice(
   args-1, operators.begin());
const std::set<std::string>    templates(
   all_templates(args));

operators_type    used_operators;
std::vector<long> used_arguments;

std::set<std::string>::const_iterator
   first_template = templates.begin();
std::set<std::string>::const_iterator
   last_template  = templates.end();

while(first_template!=last_template)
{
  const size_t template_args = (
     first_template->size()+1)/2;
    do
    {
      fill_dereference_vector(first_number_choice,
         first_number_choice+template_args,
         used_arguments);

      do
      {
        fill_dereference_vector(
           operator_choice.begin(),
           operator_choice.begin()
              +template_args-1,
           used_operators);

        const rpn_result<long> result =
           rpn_evaluate(*first_template,
           used_operators, used_arguments);
        if(result.valid)  f(result.value);
      }
      while(next_state(operator_choice.begin(),
         operator_choice.begin()+template_args-1,
         operators.end(), operators.begin()));
    }

    while(next_permutation(first_number_choice,
       first_number_choice+template_args,
       last_number_choice));
    ++first_template;
  }
  return f;
}
```

Listing 15 (cont'd)

Using Design Patterns to Manage Complexity

Simpler programs are more reliable. Omar Bashir sees how to make improvements.

Conditional statements are used to alter the flow of control. Increase in the number of variables in these statements, the number of alternative branches or the depth of conditional statements add to the complexity of sections of programs where these statements exist. Design patterns can help manage this complexity by dividing these conditional statements between different participants of the patterns used.

Introduction

All programming languages contain conditional statements that are used to alter the flow of control. Based on the values of certain variables, these conditional statements determine the branch of the program that is executed. Popular conditional statement constructs are IF, IF ELSE, IF ELSE IF and SWITCH. While IF determines whether a certain branch of the program is executed or not, alternative branches to be executed are determined using IF ELSE, IF ELSE IF and SWITCH statements.

Complexity due to conditional statements depends on at least three factors,

1. Number of variables examined to determine which branch is to be executed,
2. Number of alternative branches to be selected from,
3. Depth of nested conditional statements.

In each of the above mentioned cases, complexity arises due to the increase in the number of conditions a developer needs to consider to implement an algorithm correctly. Attempting to reduce complexity arising due to the number of variables examined by rearranging conditional expressions may, in certain cases, increase complexity by increasing the depth of conditional statements. For example, consider the following,

```
if (A && B){
  doThis();
} else if (A && !B){
  doThat();
} else {
  throwAnException();
}
```

where `A` and `B` are conditional expressions that may also be considerably complex. This can, however, be rearranged as follows, introducing nesting,

```
if (A){
  if (B){
    doThis();
  } else {
    doThat();
  }
} else {
  throwAnException();
}
```

Historically, complexity in programs arising due to the number of conditional and iterative statements has been measured using the ***cyclomatic complexity*** metric [McCabe1976]. As explained later in the article, cyclomatic complexity of a program module is directly related to the number of conditional and iterative statements within that module. It has also been argued that higher cyclomatic complexity leads to higher defects [McCabeEtAl1989], [Sharpe2008], lower cohesion [SteinEtAl2005] and larger number of unit tests for greater code coverage [WatsonEtAl1996].

This article uses a simple example to demonstrate the management of cyclomatic complexity through the application of design patterns [GammaEtAl1995]. As conditional statements affect program behaviour, some behavioural patterns can help design and implement modules with lower complexities. These lower complexity modules are then connected appropriately to provide the required behaviour.

After discussing cyclomatic complexity and its use in software development, the article proceeds with an example of a class with moderate cyclomatic complexity. The class in the example is refactored to the CHAIN OF RESPONSIBILITY design pattern and later to the STRATEGY design pattern to show reduced cyclomatic complexity in the participating classes.

Cyclomatic complexity

Cyclomatic complexity measures the amount of decision logic in a program module as a function of the number of edges and nodes in a control flow graph representing that module. The following rules provide a simple mechanism to determine cyclomatic complexity of a module,

- Each module without any control flow statements is assigned a complexity measure of 1. For example, the following method,

```
public static void main(String[] args){
  int operand1 = Integer.parseInt(args[0]);
  int operand2 = Integer.parseInt(args[1]);
  int sum = operand1 + operand2;
  System.out.println(
     operand1 + "+" + operand2 + "=" + sum);
}
```

- If there are only p binary predicates in a module, its cyclomatic complexity is p+1. For example, the cyclomatic complexity of the following method is 2.

```
public static int abs(int in){
  int out = in;
  if (out < 0){
    out = -1 * out;
  }
  return out;
}
```

Omar Bashir had his first experiences with programming trying to interface devices in avionics systems over 15 years ago. His interests in device interfacing have evolved from networking to distributed systems and also architectures and patterns. He is currently working as a software developer for a financial services company..

complexity in programs arising due to the number of conditional and iterative statements has been measured using the cyclomatic complexity metric metric

Complexity is incremented for every boolean operator (e.g., AND, OR) in a predicate. For example, the cyclomatic complexity of the following method is 3.

```
public static boolean isSingleDigitPositive(
    int in){
  boolean singleDigitPositiveFlag = false;
  if ((in >= 0) && (in < 10)){
    singleDigitPositiveFlag = true;
  }
  return singleDigitPositiveFlag;
}
```

- Cyclomatic complexity of multiway decisions is one less than the number of edges out of the decision node. For example, the cyclomatic complexity of the following method is 8. This includes 1 for method itself, 4 for the multiway if block (5 edges including the last else, i.e., the default edge) and 3 for logical AND operators.

```
public static double calculatePayment(
    int hoursParked){
  double parkingCharge;
  if ((hoursParked >= 1) && (hoursParked < 3)){
    parkingCharge = 1.25;
  } else if ((hoursParked >= 3) && (
     hoursParked < 6)){
    parkingCharge = 3.0;
  } else if ((hoursParked >= 6) && (
     hoursParked < 12)){
    parkingCharge = 7.5;
  } else if (hoursParked >= 12){
    parkingCharge = 15.0;
  } else {
    parkingCharge = 0.75;
  }
  return parkingCharge;
}
```

Because this measure is based on the decision structure of the code, it is completely independent of text formatting and is nearly independent of programming languages as they usually contain the same fundamental decision structures. Therefore, it can be applied uniformly across projects [McCabeEtAl1989].

Cyclomatic complexity fundamentally focuses on the complexity of a module. It has been ascertained that number of tests required to test all the paths in a module is equal to the cyclomatic complexity of the module. It has been widely suggested that cyclomatic complexity greater than 10 results in modules with higher defect rates thus implying this as a threshold beyond which modules are considered complex [Glover2006].

Integration complexity is a measure that provides the number of independent integration tests through an entire program but it depends on the cyclomatic complexities of component modules [McCabeEtAl1989], [WatsonEtAl1996]. While cyclomatic complexity of a particular module can be lowered by further modularisation, overall complexity may rise when considering integration complexity unless the process of modularisation produces a considerable number of common, reusable modules [McCabeEtAl1989]. Inheritance and polymorphism in object oriented programs result in implicit control flow which is used to resolve dynamic method invocation (i.e., deciding the concrete implementation of an abstract method to be executed). This is considered to increase the cyclomatic complexity and the number of tests required to perform structured testing of a module containing a reference to an instance of a class [WatsonEtAl1996]. However, in most circumstances, a program may reference only a segment of a particular class hierarchy, thereby only exercising a segment of the overall implicit control flow. It may, therefore, be practical to only consider cyclomatic complexity arising from the relevant segment of the implicit control flow and perform testing accordingly.

Lower cyclomatic complexity as a consequence of modularisation is desirable despite possibly increased integration complexity resulting in higher overall number of tests. This is because lower cyclomatic complexity results in higher cohesion and fewer anticipated defects within modules [SteinEtAl2005]. Therefore, this article discusses, through the application of design patterns, refactoring code with higher cyclomatic complexity into a structure where constituents have lower individual cyclomatic complexity.

Design patterns and their impact on cyclomatic complexity

A design pattern describes a solution to a recurring software engineering problem at the design level. Design patterns aim to promote reusability of successful designs and architectures. A design pattern is identified by a name, solves a particular problem, provides a general arrangement of components in a representative solution and describes the consequences of applying that pattern. Gamma *et al* have documented a catalogue of general purpose design patterns which are often referred to as GoF (Gang of Four) patterns [GammaEtAl1995]. They classify these patterns on the basis of their purpose as well as scope. Purpose reflects what patterns do whereas scope specifies whether the pattern applies primarily to classes or objects.

Gamma *et al* suggest that patterns can have a creational, structural or behavioural purpose. Creational patterns provide mechanisms for object creation. Structural patterns suggest the composition of classes or objects. Behavioural patterns describe the interaction of classes or objects and the division of responsibilities among them.

Metsker *et al* provide an alternative and a finer grained classification of GoF patterns on the basis of their purpose. They suggest classifying GoF patterns on the basis of their intent into interface, responsibility, construction, operation and extension pattern classes [MetskerEtAl2006]. Interface patterns provide convenient or appropriate interfaces to their respective implementations or to a collection or composition of objects. Responsibility patterns assign responsibility of operations to various objects and also define mechanisms to determine the objects in the system having the responsibility. Construction patterns define mechanisms to

A design pattern describes a solution to a recurring software engineering problem at the design level

Pattern	Classification on Purpose	Classification on Intent
ABSTRACT FACTORY	Creational	Construction
BUILDER	Creational	Construction
FACTORY METHOD	Creational	Construction
PROTOTYPE	Creational	Construction
SINGLETON	Creational	Responsibility
ADAPTER	Structural	Interface
BRIDGE	Structural	Interface
COMPOSITE	Structural	Interface
DECORATOR	Structural	Extension
FAÇADE	Structural	Interface
FLYWEIGHT	Structural	Responsibility
PROXY	Structural	Responsibility
CHAIN OF RESPONSIBILITY	Behavioural	Responsibility
COMMAND	Behavioural	Operation
INTERPRETER	Behavioural	Operation
ITERATOR	Behavioural	Extension
MEDIATOR	Behavioural	Responsibility
MEMENTO	Behavioural	Construction
OBSERVER	Behavioural	Responsibility
STATE	Behavioural	Operation
STRATEGY	Behavioural	Operation
TEMPLATE METHOD	Behavioural	Operation
VISITOR	Behavioural	Extension

Table 1

construct various objects. Operation patterns address the contexts in which more than one method is needed in a design to perform similar operations or variations of the same operation. Finally, extension patterns extend or vary the fundamental operation of an object.

Other classifications of GoF patterns are also possible. Table 1 lists GoF patterns with the two classifications mentioned above.

Although design patterns focus on reusability and extensibility, most design patterns, particularly behavioural patterns, can assist in the reduction of cyclomatic complexity. This is accomplished by organising participants such that each solves a part of the problem resulting in simpler control flows within each participant hence lowering their cyclomatic complexity. Overall, each is more cohesive resulting in simpler testing. The GoF's description of some of the design patterns specifically mentions simplification of the control flow in the resulting solution (e.g., STRATEGY).

Example – arithmetic calculator

Listing 1 shows a Java class that performs basic arithmetic calculations. The `main` method of the program takes two operands separated by an operator. These are used by the `calculate` method to perform the operation related to the operator provided. The `calculate` method employs a `switch` statement to determine the branch that contains the statement to perform the required arithmetic operation. Cyclomatic complexity of the `calculate` method is 6.

Refactoring arithmetic calculator using chain of responsibility

CHAIN OF RESPONSIBILITY consists of a chain of loosely coupled objects with responsibilities between objects kept specific and minimal. These include knowing the task they can perform and the next object in the chain to which a message can be propagated if they cannot process the message. Client objects need not know which object in the chain has to be sent a message to perform a particular task. Instead, clients only know of the object at the head of the chain and they send the message to it. That object processes the message if it can otherwise it passes it on to the next object in the chain.

The calculator shown in Listing 1 can be refactored into a CHAIN OF RESPONSIBILITY as shown in Figure 1. Four classes, `Adder`, `Subtractor`, `Multiplier` and `Divider` perform respective arithmetic operations (Listing 3). These classes inherit from an abstract base class `Processor` (Listing 2). An instance of a `Processor`'s subclass is an element in the CHAIN OF RESPONSIBILITY and maintains a reference (`successor`) to another object of one of its subclasses. An object of the `Calculator` class (Listing 4) acts as a client to an instance of a `Processor`'s subclass, which is the head of the chain.

In the `main` method of the `Calculator` class, a chain of `Adder`, `Subtractor`, `Multiplier` and `Divider` instances are created with the `Adder` instance at the head of the chain. The `Adder` instance is then assigned to the `processor` attribute of the `Calculator` instance by passing it as a parameter to the `setProcessor` method on the `Calculator` instance. To perform a calculation, a `Calculator` instance calls the `calculate` method on the object referenced by its `processor` attribute. The `calculate` method calls the `canProcess` method to determine if that instance can perform the calculation. If it can, it calls the `process` method to perform the calculation and return the results. If it cannot and a successor exists in the chain (i.e., the value of the `successor` attribute is not null), it calls the `calculate` method on the successor.

In addition to CHAIN OF RESPONSIBILITY, `Processor` and its subclasses also implement the TEMPLATE METHOD pattern. `calculate` method is a concrete method in the `Processor` base class. However, both `canProcess` and `process` methods are abstract methods, specialised implementations of which exist in the subclasses of `Processor`.

lower cyclomatic complexity results in higher cohesion and fewer anticipated defects

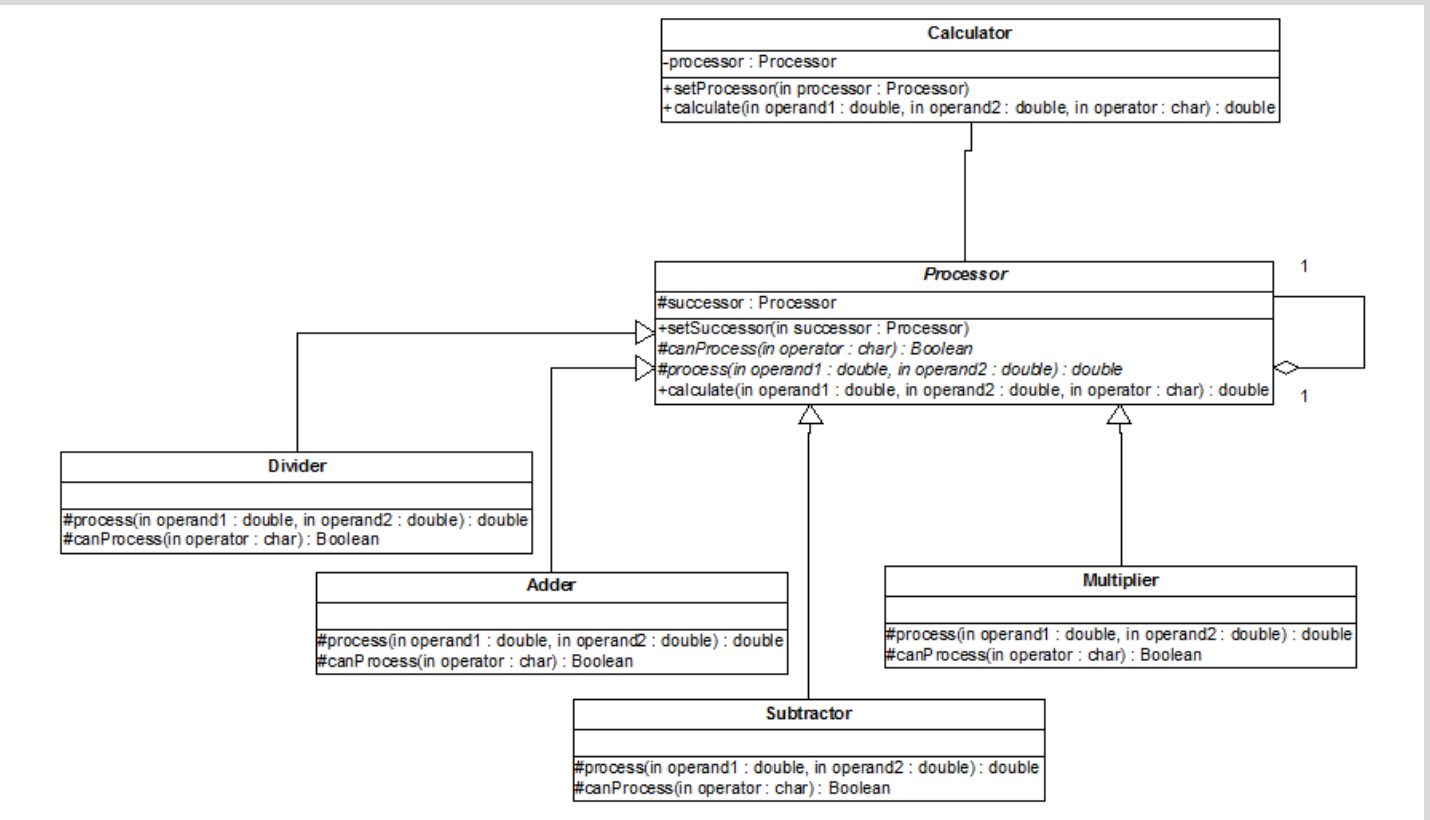

Figure 1

The code in listings 2 to 4 results in more classes than the code in listing 1 but these classes are more cohesive and have lower individual cyclomatic complexities. The cyclomatic complexities of all methods of `Adder`, `Subtractor` and `Multiplier` classes are 1 whereas that of the `process` method of the `Divider` class is 2. The cyclomatic complexities of all instance methods of the `Calculator` class are also 1. The `main` method of the `Calculator` class and the `calculate` method of the `Processor` class have the highest cyclomatic complexities of 3.

Although it may be argued that this exercise has increased the number of classes in the application, each of these classes is far less complex than the original program. Specialised behaviour of these classes allows developers to conveniently focus on the functionality of the class being developed and an extension to the application (e.g., a class to provide the remainder, i.e., modulus operation) only requires developing a new class and adding its instance to the end of the chain in the `main` method of the `Calculator` class. Techniques like dependency injection [Fowler2004] allow creation of such chains via application configuration rather than directly in code. Furthermore, each class can be tested independently with simpler unit tests. These advantages, especially in more complicated applications (e.g., message handling in a communications application or update handling in a GUI based on the OBSERVER pattern), may offset the increased integration complexity due to an increased number of classes in the application.

Refactoring arithmetic calculator using Strategy

The STRATEGY pattern allows the definition of a family of algorithms such that they are interchangeable within an application. Thus it lets algorithms vary independently from the clients that use it. STRATEGY is typically used to configure a class with one of many behaviours, when different variants of an algorithm are needed or when a class defines may behaviours and these appear as multiple conditional statements in its operations [GammaEtAl1995].

Figure 2 is the class diagram of the calculator application implemented using a variation of the STRATEGY pattern. The `Processor` interface represents the abstract strategy in the application:

```
package calculator;
public interface Processor {
  double process(double operand1,
                 double operand2);
}
```

refactoring code with higher cyclomatic complexity using design patterns can lead to flexible, configurable and extensible systems

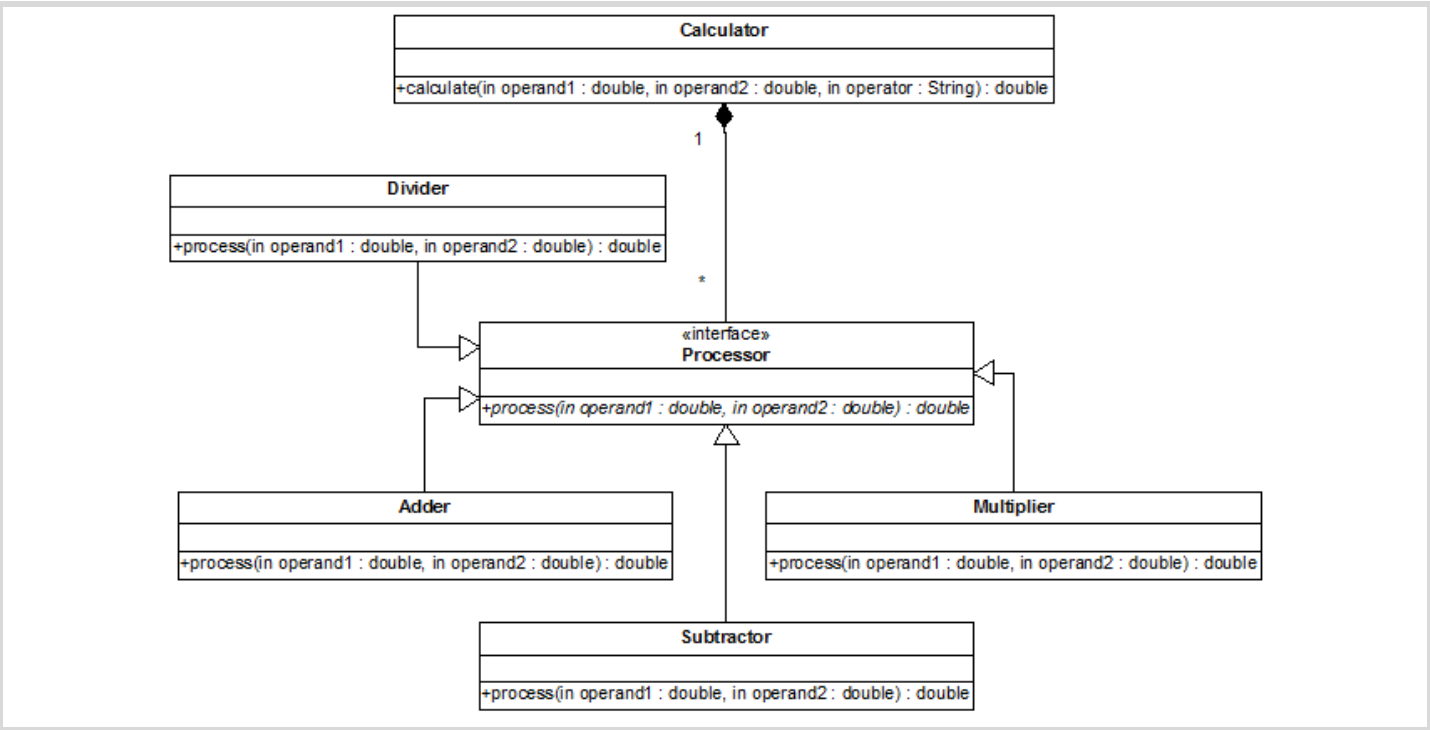

Figure 2

Its implementations, `Adder`, `Subtractor`, `Multiplier` and `Divider` (Listing 5) form concrete strategies for the application. `Calculator` class (Listing 6) is the context. However, it differs from the context in the GoF's description of the STRATEGY pattern as it contains a collection of concrete strategy instances from which it selects the appropriate strategy instance on the basis of the operation requested.

`Adder`, `Subtractor`, `Multiplier` and `Divider` (Listing 5) contain only one method, `process`. For `Adder`, `Subtractor` and `Multiplier`, this method has a cyclomatic complexity of 1 whereas for the `Divider`, it has a cyclomatic complexity of 2. The constructor of the `Calculator` class creates a mapping between arithmetic operators and instances of the corresponding implementations of the `Processor` interface (the `processors` attribute of the `Calculator` class). The `calculate` method determines, using the `operator` parameter, if an instance of an implementation of the `Processor` interface has been mapped to this operator. If such a mapping exists, that instance of the implementation of `Processor` interface is accessed and its process method called to perform the requested operation. If such a mapping does not exist, an exception is thrown. Cyclomatic complexity of the `calculate` method of the `Calculator` class is 2 whereas that of the `main` method is 3.

Like the CHAIN OF RESPONSIBILITY implementation of the calculator, this implementation also requires an initialisation step, which is performed in the constructor of the `Calculator` class. As mentioned earlier, this can also be delegated to a dependency injector. Individual classes are more cohesive than the original implementation of calculator which, as mentioned earlier, is usually preferred even at the cost of possible higher integration complexity.

Conclusions

The cyclomatic complexity of a program module is determined by examining the control flow within the module. It has been suggested that complicated control flows in a single module may also have an adverse impact on developers' productivity through cognitive overload [Klemola2000]. Thus, quantification of cyclomatic complexity provides an opportunity to reduce module complexity by delegating parts of control flow to other modules. This leads to lower individual cyclomatic complexities with possibly higher integration complexity. However, reduction of cyclomatic complexity also leads to higher cohesion which is proven to be a key aspect of well designed software.

This article discusses the reduction of cyclomatic complexity in a method of a class through the application of design patterns. It has been demonstrated, with a simple example, that refactoring code with higher cyclomatic complexity using design patterns can lead to flexible, configurable and extensible systems. However, it can also be seen that such refactoring increases the structural complexity of the solution due to more

some behavioural patterns can help design and implement modules with lower complexities

```
package calc;
import java.lang.IllegalArgumentException;
public class Calculator {
  public double calculate(double operand1,
     double operand2, char operator){
    double result = 0.0;
    switch(operator){
    case '+':
      result = operand1 + operand2;
      break;
    case '-':
      result = operand1 - operand2;
      break;
    case '*':
      result = operand1 * operand2;
      break;
    case '/':
      if (Math.abs(operand2) > 0.0){
        result = operand1 / operand2;
      } else {
        throw new ArithmeticException(
           "Numerator is zero.");
      }
      break;
    default:
      throw new IllegalArgumentException(
         operator + " unknown.");
    }
    return result;
  }
  public static void main(String[] args){
    double operand1
       = Double.parseDouble(args[0]);
    double operand2
       = Double.parseDouble(args[2]);
    char operator = args[1].charAt(0);
    double result = new Calculator().calculate(
       operand1, operand2, operator);
    System.out.println(operand1 + args[1]
       + operand2 + "=" + result);
  }
}
```

Listing 1

number of resulting classes with possible coupling between them. Therefore, the application of these techniques may be more appropriate in systems where conditional statements allow selection from several elaborate and complex branches with a possibility of variation in these branches in the future versions of the system. This variation can be in the number of these branches as well as functionality within each branch. A typical example may be of a message receiver in a data communications application communicating many different types of messages. Each

```
package calculator;
public abstract class Processor {
  protected Processor successor;
  public Processor(){
    successor = null;
  }
  public void setSuccessor(Processor successor){
    this.successor = successor;
  }
  protected abstract boolean canProcess(
     char operator);
  protected abstract double process(
     double operand1, double operand2);
  public double calculate(double operand1,
     double operand2, char operator)
     throws Exception {
    double result;
    if (canProcess(operator)){
      result = process(operand1, operand2);
    } else {
      if (null != successor){
        result = successor.calculate(
           operand1, operand2, operator);
      } else {
        throw new Exception("No successor set");
      }
    }
    return result;
  }
}
```

Listing 2

message type may be handled by a separate message handler. These message handlers can be arranged in a CHAIN OF RESPONSIBILITY or a STRATEGY. Not only is each message handler more cohesive but the resulting application is also extensible in handling newer message types. ■

References

[Fowler2004] M. Fowler (2004), 'Inversion of Control Containers and the Dependency Injection Pattern', http://martinfowler.com/articles/injection.html.

[GammaEtAl1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995), *Design Patterns – Elements of Reusable Object Oriented Software*, Pearson Education.

[Glover2006] A. Glover (2006), 'In Pursuit of Code Quality: Monitoring Cyclomatic Complexity', http://www.ibm.com/developerworks/java/library/j-cq03316/

[Klemola2000] T. Klemola (2000), 'A Cognitive Model for Complexity Metrics', *Proceedings of the 4th International Workshop on Quantitative Approaches in Object Oriented Software Engineering*.

reduce module complexity by delegating parts of control flow to other modules

```
public class Adder extends Processor {
  protected boolean canProcess(char operator) {
    return operator == '+';
  }
  protected double process(double operand1,
     double operand2) {
    return operand1 + operand2;
  }
}
public class Subtractor extends Processor {
  protected boolean canProcess(char operator) {
    return operator == '-';
  }
  protected double process(double operand1,
     double operand2) {
    return operand1 - operand2;
  }
}
public class Multiplier extends Processor {
  protected boolean canProcess(char operator) {
    return operator == '*';
  }
  protected double process(double operand1,
     double operand2) {
    return operand1 * operand2;
  }
}
public class Divider extends Processor {
  protected boolean canProcess(char operator) {
    return operator == '/';
  }
  protected double process(double operand1,
     double operand2) {
    double result;
    if (Math.abs(operand2) > 0.0){
      result = operand1/operand2;
    } else {
      throw new ArithmeticException(
         "Divide by zero.");
    }
    return result;
  }
}
```

Listing 3

[McCabe1976] T. J. McCabe (1976), 'A Complexity Measure', *IEEE Transactions on Software Engineering*, SE-2(4), December 1976, pp 308–320.

[McCabeEtAl1989] T. J. McCabe, C.W. Butler (1989), 'Design Complexity, Measurement and Testing', *Communications of the ACM*, December 1989, 32(12), pp 1415–1425.

```
package calculator;

public class Calculator {
  private Processor processor;
  public Calculator(){
    processor = null;
  }
  public void setProcessor(Processor processor){
    this.processor = processor;
  }

  public double calculate(double operand1,
     double operand2,
     char operator) throws Exception{
    return processor.calculate(operand1,
       operand2, operator);
  }

  public static void main(String[] args){
    Adder adder = new Adder();
    Subtractor subtractor = new Subtractor();
    Multiplier multiplier = new Multiplier();
    Divider divider = new Divider();
    adder.setSuccessor(subtractor);
    subtractor.setSuccessor(multiplier);
    multiplier.setSuccessor(divider);
    Processor processor = adder;
    Calculator calculator = new Calculator();
    calculator.setProcessor(processor);
    if (args.length != 3){
      System.out.println(
         "USAGE: java calculator operand1
         operator operand2");
    } else {
      double operand1
         = Double.parseDouble(args[0]);
      double operand2
         = Double.parseDouble(args[2]);
      char operator = args[1].charAt(0);
      try{
        double result
           = calculator.calculate(operand1,
           operand2, operator);
        System.out.println(operand1 + args[1]
           + operand2 + "=" + result);
      } catch (Exception exp){
        System.out.println(exp.toString());
      }
    }
  }
}
```

Listing 4

Not only is each message handler more cohesive but the resulting application is also extensible in handling newer message types

```
public class Adder implements Processor {
  public double process(double operand1,
     double operand2) {
    return operand1 + operand2;
  }
}

public class Subtractor implements Processor {
  public double process(double operand1,
     double operand2) {
    return operand1 - operand2;
  }
}

public class Multiplier implements Processor {
  public double process(double operand1,
     double operand2) {
    return operand1 * operand2;
  }
}

public class Divider implements Processor {
  public double process(double operand1,
     double operand2) {
    double result;
    if (Math.abs(operand2) > 0.0){
      result = operand1/operand2;
    } else {
      throw new ArithmeticException(
         "Divide by zero");
    }
    return result;
  }
}
```

Listing 5

[MetskerEtAl2006] S. J. Metsker, W. C. Wake (2006), *Design Patterns in Java*, Pearson Education.

[Sharpe2008] R. Sharpe (2008), McCabe 'Cyclomatic Complexity: The Proof in the Pudding', http://www.enerjy.com/blog/?p=198

[SteinEtAl2005] C. Stein, G. Cox, L. Etzkorn, S. Gholston, S. Virani, P. Farrington, D. Utley, J. Fortune (2005), 'Exploring the Relationship Between Cohesion and Complexity', *Journal of Computer Science*, 1(2), pp 137–144.

[WatsonEtAl1996] A.H. Watson, T.J. McCabe (1996), *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST (National Institute of Standards and Technology) Special Publication 500-235.

```
package calculator;
import java.util.Map;
import java.util.TreeMap;

public class Calculator {
  private Map<String, Processor> processors;

  public Calculator(){
    processors = new TreeMap<String,
       Processor>();
    processors.put("+", new Adder());
    processors.put("-", new Subtractor());
    processors.put("*", new Multiplier());
    processors.put("/", new Divider());
  }

  public double calculate(double operand1,
     double operand2,
     String operator) throws Exception{
    double result;
    if (processors.containsKey(operator)){
      result = processors.get(operator).process(
         operand1, operand2);
    } else {
      throw new Exception(
         "No processor for " + operator);
    }
    return result;
  }

  public static void main(String[] args){
    if (args.length != 3){
      System.out.println("Usage:java Calculator
         <operand1> <operator> <operand2>");
    } else {
      double operand1
         = Double.parseDouble(args[0]);
      double operand2
         = Double.parseDouble(args[2]);
      Calculator calculator = new Calculator();
      try{
        double result = calculator.calculate(
           operand1, operand2, args[1]);
        System.out.println(operand1 + args[1]
           + operand2 + "=" + result);
      } catch(Exception exp){
        System.out.println(exp.toString());
      }
    }
  }
}
```

Listing 6

The Predicate Student: A Game of Six Integers

How easily can you solve puzzles? Nigel Eke applies some logic.

Dum de dum de dum... boo doo, boo doo choo. Ah Countdown...

Thanks very much Richard Harris for bringing back memories of the Countdown TV show [Harris]. I too became fixated by this TV parlour game, back in the late nineties.

The mathematical section (finding an arithmetic formula using up to six blindly selected numbers to equate to a random goal) became another of my pet projects. Write a program which would be able to find an equation before Carol[1] did.

The approach taken was similar to Richard, but different enough that I thought it worth writing up so that *Overload* readers can make a comparison. The article also gives a taste of logic programming and, in particular, the logic programming language of Prolog. It is a very quick introduction though, and I would recommend *The Art of Prolog* [Sterling] for a definitive, and much more detailed, description of the language.

Prolog facts

The Prolog language is based around stating facts, and then making queries on those facts. The facts may be simple (yellow is a colour), or more complex (person A is the granddaughter of person B if one of A's parents is the child of B and A is female). The queries are used for finding unknowns, for example 'who is Elizabeth's father?' or 'let me know all descendants of Elizabeth who have fair hair and were born between 1800 and 1900'. This simple approach is remarkably powerful and expressive for a particular class of problems: those which require searching through many combinations of possible solutions.

Let us start with our first Prolog program – declaring one fact, but otherwise doing nothing:

```
colour(yellow).
```

yellow is called an *atom*. **colour** is known as a *fact* or a *predicate*. There are also built-in predicates, which we'll come across later. The name 'colour' selected here is chosen simply so that it does not clash with the built-in names.

How does this program get loaded and what does it mean to 'run' the program? All the examples here are demonstrated in a Prolog interpreter – SWI-Prolog [SwiProlog]. To load a program in SWI-Prolog from the file `colour.pl` we do the following:

```
?- [colour].
% colour compiled 0.00 sec, 492 bytes
true.
```

Simple – and so is running the program:

```
?- colour(yellow).
true.

?- colour(red).
false.
```

1. Carol Vorderman – Countdown, 1982–2008

```
colour(yellow).
colour(cyan).
colour(magenta).
colour(red).
colour(green).
colour(blue).
```

Listing 1

```
?- colour(X).
X = yellow ;
X = cyan ;
X = magenta ;
X = red ;
X = green ;
X = blue.

?- colour(_).
true ;
true ;
true ;
true ;
true ;
true.
```

Listing 2

Running the program is simply a matter of testing the stated facts or predicates. Additional facts can be declared dynamically, but we do not touch on this further here.

The first query confirms yellow is a colour by returning a **true** result. The second query, however, shows that red is not a colour – **false** was returned. This is because it has not been declared as a fact.

Let's expand the source file, as shown in Listing 1, to declare the fact that more than one colour exists, and make a query for all known colours (Listing 2).

The query **colour(X).** demonstrates the Prolog convention that variables start with an upper-case character. The output from this query shows **X** being bound to each of the possible facts.

In this instance we do not go on to use the variable, so it could also have been written as **colour(_)**. However, it can only be determined that the query found six results, but not what the results are.

Nigel Eke has been in the software engineering industry since stone-age man invented the wheel. He is currently a Senior BI Consultant working Down Under with open source software (Pentaho) in agile development environments. He can be contacted at me@nigel-eke.com

remarkably powerful and expressive for a particular class of problems: those which require searching through many combinations of possible solutions

Deeper facts

The above examples provide an introduction to syntax and some of the terminology used in the language. The next example shows facts with more than one atom, e.g. `mother(adam, allison)`, and conjoined predicates e.g.

```
grandmother(Child, Grandmother) :-
 parent(Child, X), mother(X, Grandmother).
```

Let's take these one at a time.

`mother(adam, allison)` means the fact that 'The mother of Adam is Allison'. So why not write it this way – `mother(allison, adam)` – and read it as 'Allison is the mother of Adam'. Well, you can do that too, *but not in the **same** program*. It is important to use consistent semantics throughout. For the remainder of the family tree examples the semantic meaning behind `relation(A, B)` is 'B is the <relation> of A'.

So what about the statement:

```
grandmother(Child, Grandmother) :-
  parent(Child, X), mother(X, Grandmother).
```

The `:-` means 'if' and the `,` is taken to mean 'and'. So the statement reads 'Grandmother is the grandmother of Child *if* someone (X) is the parent of the Child *and* Grandmother is the mother of that someone (X)'. This is known as a *conjoined predicate*.

To program the equivalent of 'or', e.g. the fact that a child's grand*parent* can be a grandmother *or* a grandfather, simply write each part of the 'or' as a separate statement.

```
grandparent(Child, Grandparent) :-
grandmother(Child, Grandparent).
grandparent(Child, Grandparent) :-
grandfather(Child, Grandparent).
```

Bounding along

One of the concepts in Prolog is *binding*. Variables get values *bound* to them. What do we mean by *binding* and being *bound*? A variable is either bound to a value, or it is not. The variable does not change its value once bound. At least, it does not change until Prolog wants to see if any further facts satisfy the predicate being queried. This happens when a predicate fails or a statement has been fully tested. At this point Prolog steps back through a tree of statement calls it has been recording, until such time it can start creating a new branch of the tree and bind the next value.

Consider the three simple facts:

```
fact(1,a).
fact(2,b).
fact(3,c).
```

The predicate can be tested with neither parameter bound, or with either one or both parameters bound (Listing 3). In the case where neither parameter is bound Prolog returns all facts. It initially returns the first fact, binding the variables to the values found in that fact. Once that search has been satisfied, i.e. all unbound variables have been bound to valid, self-consistent values, Prolog backtracks, and looks for the next fact that fits

```
?- fact(A,B).
A = 1,
B = a ;
A = 2,
B = b ;
A = 3,
B = c.

?- C=1,fact(C,D).
C = 1,
D = a.

?- C=4,fact(C,D).
false.

?- E=1,F=a,fact(E,F).
E = 1,
F = a.

?- E=4,F=a,fact(E,F).
false.
```

Listing 3

(indicated by the `;` in Listing 3), and so on. Similarly for one bound parameter. However, this time, it is possible that none of the facts fit the bound part, in which case `false` is returned. Finally with both parameters bound, a fact is either found or not found, in which case the original values or `false` is returned respectively.

Making a query

You've already seen examples of queries. One was the explicit query for all colours (`colour(X)`). The other was less obvious and embedded in the fact:

```
grandmother(Child, Grandmother) :-
  parent(Child, X), mother(X, Grandmother).
```

In this instance the query is to find X so that it satisfies the parent relationship to Child and Grandmother is the mother of X. The previous section on binding also shows simple queries being made with bound and unbound variables. Now another more complex, and less abstract, example is discussed around the subject of family trees.

Figure 1 shows a family tree, which is then defined in Listing 4 by declaring facts about the people, their genders and immediate relationships (only the first and last of each are shown for brevity). Listing 5 goes on to show the remaining relationship predicates.

So what actually happens when a query is made? Prolog tries to *bind* any *unbound* variables in a valid and self-consistent combination of the given facts.

Listing 6 shows a query for all father / child relationships by binding valid combinations to the variables `Father` and `Child`.

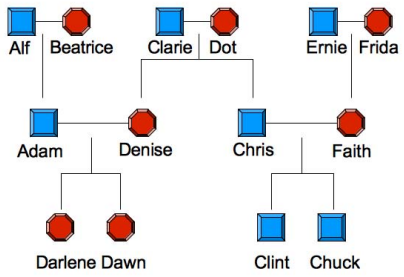

Figure 1

```
person(alf).
/* code removed */
person(chuck).

gender(alf, male).
/* code removed */
gender(chuck, male).

father(adam, alf).
/* code removed */
father(chuck, chris).

mother(adam, beatrice).
/* code removed */
mother(chuck, faith).
```

Listing 4

```
parent(Person, Parent) :- mother(Person, Parent).
parent(Person, Parent) :- father(Person, Parent).

daughter(Person, Daughter) :-
 gender(Daughter, female),
 parent(Daughter, Person).

son(Person, Son) :-
  gender(Son, male),
  parent(Son, Person).

child(Person, Child) :- parent(Child, Person).

grandmother(Person, Grandmother) :-
  parent(Person, Parent),
  mother(Parent, Grandmother).

grandfather(Person, Grandfather) :-
  parent(Person, Parent),
  father(Parent, Grandfather).

grandparent(Person, Grandparent) :-
  grandmother(Person, Grandparent).
grandparent(Person, Grandparent) :-
  grandfather(Person, Grandparent).

granddaughter(Person, Granddaughter) :-
  gender(Granddaughter, female),
  grandparent(Granddaughter, Person).

grandson(Person, Grandson) :-
  gender(Grandson, male),
  grandparent(Grandson, Person).

grandchild(Person, Grandchild) :-
  grandparent(Grandchild, Person).
```

Listing 5

```
?- father(Child, Father).
Child = adam,
Father = alf ;
Child = denise,
Father = clarie ;
Child = chris,
Father = clarie ;
Child = faith,
Father = ernie ;
Child = darlene,
Father = adam ;
Child = dawn,
Father = adam ;
Child = clint,
Father = chris ;
Child = chuck,
Father = chris.
```

Listing 6

```
?- P = chris, father(P, Father).
P = chris,
Father = clarie.

?- P = chris, father(Child, P).
P = chris,
Child = clint ;
P = chris,
Child = chuck.
```

Listing 7

Listing 7 binds the variable `P` with the value `chris` before using the bound value in the `father` query. This is done twice, once to find his father and once to find his children. Note that `P` must be bound in each query explicitly.

So far we have only used the simple facts stated in Listing 4, but none of the relationships from Listing 5. The final examples of queries (Listing 8) show all Clint's grandparents, whether Dot is one of Clint's grandparents (which she is) or whether Alf is Clint's grandparent (which he isn't). As you can see, the more complex predicates are used in just the same way as the simpler facts.

You may have noticed the `false` following the Clint / Dot query. Why is this? Even though Dot *is* Clint's grandparent *and it was even returned as a valid result*. Remember though that grandparent is either a grandmother or a grandfather. So the first of the `grandparent` facts successfully returns that Dot is a grandparent, i.e. when performing the `grandmother(Parent, Grandmother)` part. But Prolog also goes

```
?- P = clint, grandparent(P, Grandparent).
P = clint,
Grandparent = frida ;
P = clint,
Grandparent = dot ;
P = clint,
Grandparent = ernie ;
P = clint,
Grandparent = clarie.

?- P = clint, G = dot, grandparent(P, G).
P = clint,
G = dot ;
false.

?- P = clint, G = alf, grandparent(P, G).
false.
```

Listing 8

on to try the second fact, which is determining if Dot is Clint's grandfather – which she isn't – hence the `false`.

Is this a problem? The answer to that is 'It depends'. It depends on the predicate and if we know whether we need to check the remaining predicate statements.

Cuts

In the case of the `grandparent` predicate we know that, if the first test proves true then there is no point in checking further cases. This is indicated by introducing a cut (`!`):

```
grandparent(Person, Grandparent) :-
  grandmother(Person, Grandparent), !.
grandparent(Person, Grandparent) :-
  grandfather(Person, Grandparent).
```

Now, when we check if Dot is Clint's grandmother, we get the one expected result:

```
?- P = clint, G = dot, grandparent(P, G).
P = clint,
G = dot.
```

Tree hugging

As Prolog has been solving the queries given to it, it builds a tree of possible results. It is this tree building, walking and searching that makes it an ideal candidate language for the original problem – the Countdown equation finder.

The first search tree is to determine a list of all possible permutations of the six numbers. This is not just permutations on the six numbers, but also all combinations of five from the six, four from the six numbers and so on. (Not all six numbers need to be used when deriving an equation).

The program works through each of these permutations, building a tree representing an equation. Figure 2 depicts the trees for two equations (2 + (3 x 4) [a] and (2 + 3) x 4 [b]). The root of each tree is an operator and the left and right branches are the operands. Once a tree is built, the program then tests to see if the equation equals the required goal.

Before we look at the actual program however, one final Prolog concept needs to be introduced – lists.

Lists

Prolog's list syntax uses the `[` and `]` characters to define the beginning and end of a list, with the pipe `|` character delimiting one or more initial elements and separating them from the remaining tail of the list. Also by convention `[]` is the empty list.

Three predicates are shown in the program (Listing 9), all of which take a list as an argument, but access it differently. The first just prints the object (it could in fact be any object). The second and third extract one and two head elements and print them separately before printing the remaining tail.

Listing 10 shows the output from calling these predicates with a simple list, including the results of calling the predicates with a list shorter than two elements.

Finally the built-in predicate `findall` is demonstrated, which finds argument one, satisfying the predicate (argument two), and returns them

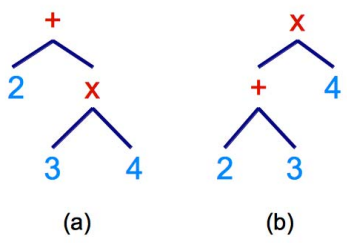

Figure 2

```
fact(a).
fact(b).
fact(c).
fact(d).
fact(e).
fact(f).

print_list(Ls) :-
  print(Ls),
  print('\n').

print_head1_list([H|Ls]) :-
  print(H),
  print('\n'),
  print(Ls),
  print('\n').

print_head2_list([H1,H2|Ls]) :-
  print(H1),
  print('\n'),
  print(H2),
  print('\n'),
  print(Ls),
  print('\n').
```

Listing 9

in a list (argument three). (For this example note the `fact` facts declared at the start of the program).

Countdown to Countdown

At last we have enough Prolog behind us to allow us to look at the Countdown program (Listing 11). The main two predicates that solve the problem are `operation` and `findTree`. That's seven statements, or

```
?- Xs=[1,2,3,4,5,6], print_list(Xs).
[1, 2, 3, 4, 5, 6]
Xs = [1, 2, 3, 4, 5, 6].

?- Xs=[1,2,3,4,5,6], print_head1_list(Xs).
1
[2, 3, 4, 5, 6]
Xs = [1, 2, 3, 4, 5, 6].

?- Xs=[1,2,3,4,5,6], print_head2_list(Xs).
1
2
[3, 4, 5, 6]
Xs = [1, 2, 3, 4, 5, 6].

?- Xs = [1], print_list(Xs).
[1]
Xs = [1].

?- Xs = [1], print_head1_list(Xs).
1
[]
Xs = [1].

?- Xs = [1], print_head2_list(Xs).
false.

?- findall(X, fact(X), Xs), print_head2_list(Xs).
a
b
[c, d, e, f]
Xs = [a, b, c, d, e, f].
```

Listing 10

eleven lines of code. This shows the power of Prolog for this type of problem solving.

In summary, we have the following predicates which form the entire program:

- **subsets** and **subsets2** – finds all subsets of all combinations of 1 from 6, 2 from 6 through to all 6 numbers.
- **operation** – performs each of the four arithmetic operations permitted in the final equation.
- **findTree** – finds all possible equation trees.
- **makeExpr** and **treeExpr** – helper predicates to create the equation text to be output.
- **delta_length** – helper predicate to enable the shortest list of numbers to get checked first and give the most efficient equation, i.e. the one using the minimum number of numbers.
- countdown – the real goal of the game.

Countdown and lift-off

So how do each of these predicates work?

subsets2

subsets2 enables us to find subsets of a set. **subsets2** should be seen as a 'private predicate' of subsets, and is not designed to be called by other predicates[1].

The first statement simply states a set (**Xs**) is a subset of itself.

The second statement gets all subsets (**Ys**) of length N-1 from set **Xs** (length N). This is done with the help of the built-in predicate **select**, which selects a member (not used, hence **_**) from **Xs**, giving the remaining set **Ys**.

The final statement gets all subsets (**Ys**) of length N-n, where n > 1, from set **Xs** (length N). First we get all subsets (**Y1s**) of length N-1 from set **Xs**, as before, but then pass this result, recursively into **subsets2**, to get the subsets of **Y1s**. It is as result of this recursion that we generate duplication – which is removed in our **subsets** predicate.

subsets

Here we return a 'set of sets' (**Xss**). The 'set of sets' returned is actually the set of *subsets* of **Xs**.

findall is used to find a temporary list of subsets (**Yss**) of all **Xs** which match the predicate **subset2**. **subsets2(Xs, Ys)** binds each subset of **Xs** to **Ys**. **findall** takes each result **Ys**, and adds it to its result list **Yss**.

We know, because of way **subsets2** is written, that the list **Yss** contains duplicate subsets. **list_to_set** is a built-in predicate that removes the duplicates in **Yss**, binding the result we are after to **Xss**.

operation

Each of of the four allowed mathematical operations are handled here, with **operation(operand1, operand2, resultantValue, operatorString)**. The operands are always bound when this predicate is tested.

resultantValue gets bound to the result of the calculating the operation. There is one circumstance when **resultantValue** is already bound – which is discussed further when we look at the main **countdown** predicate.

The string representing the operation (**operationString**) is also bound on return from the predicate, in order to be used when printing out the final equation.

The minus operation dismisses negative results. Although they could be used as interim results in finding the final goal we know that a) the final goal will always be positive, and b) the operands can be re-arranged with the plus operator to achieve the same – and seemingly cleaner, as far as

1.As far as I'm aware, Prolog has no way to make predicates private.

```
/* Helper function for subsets/2 */
subsets2(Xs, Xs).
subsets2(Xs, Ys) :- select(_, Xs, Ys).
subsets2(Xs, Ys) :- select(_, Xs, Y1s),
subsets2(Y1s, Ys).

/* Find all subsets 0 of N, 1 of N .. N of N for
the list Xs */
subsets(Xs, Xss) :-
  findall(Ys, subsets2(Xs, Ys), Yss),
  list_to_set(Yss, Xss).

/* Find value of each of the four arithmatical
operations */
operation(O1, O2, V, ' + ') :- V is O1 + O2.
operation(O1, O2, V, ' - ') :- V is O1 - O2, V >
0.
operation(O1, O2, V, ' * ') :- not(O1 = 1), not(O2
= 1), V is O1 * O2.
operation(O1, O2, V, ' / ') :- not(O2 = 1), M is
O1 mod O2, M = 0, V is O1 // O2.

/* Build equation trees from given list. If goal is
bound then only trees which match the goal are
deemed successful. */
findTree([], _, _) :- !, fail.
findTree([X|[]], value(X), X) :- !.
findTree(Xs, tree(O, TL, TR), V) :-
  append(L, R, Xs),
  not(compare(=, L, Xs)), findTree(L, TL, V1),
  not(compare(=, R, Xs)), findTree(R, TR, V2),
  operation(V1, V2, V, O).

/* Helper for building equation text. */
makeExpr(L, O, R, E) :-
  concat('(', L, E1),
  concat(E1, O, E2),
  concat(E2, R, E3),
  concat(E3, ')', E).

/* Return equation text for given operations tree.
*/
treeExpr(value(X), X).
treeExpr(tree(O, L, R), E) :-
  treeExpr(L, Le),
  treeExpr(R, Re),
  makeExpr(Le, O, Re, E).

/* Helper function to sort lists of subsets of the
choosen numbers based on list length. */
delta_length('<', Xs, Ys) :-
  length(Xs, XL),
  length(Ys, YL),
  compare(<, XL, YL), !.
delta_length('>', _, _).

/* Main predicate. Expects Vn and Goal to be bound
when called. */
countdown(V1, V2, V3, V4, V5, V6, Goal, E) :-
  subsets([V1, V2, V3, V4, V5, V6], Xss),
  predsort(delta_length, Xss, Yss),
  member(Ys, Yss),
  permutation(Ys, Y1s),
  findTree(Y1s, T, Goal), !,
  treeExpr(T, E).

countdown(_, _, _, _, _, _, _,
  'It''s impossible').
```

Listing 11

the final equation appears – result. So, for A - (-B) the equivalent A + B will also appear in the possible equation trees. Similarly for A + (-B), which becomes A - B, i.e. we never have to deal with a negative B value.

The multiplication operation dismisses the uninteresting operand one because 1 * N = N * 1 = N.

The division operation dismisses the uninteresting case where N / 1 = N. It also makes sure that the result is non-fractional.

findTree

This is the crux of the program. The part that does the real work in finding an equation to match the goal.

The format of the predicate is `findTree(listOfOperands, tree, value)`. The `listOfOperands` is always bound when the predicate is tested. The tree will be a representation of the operation tree (see Figure 2). value gets bound to the value of the result of the tree's operations being applied.

The first statement dismisses the empty list of operands, by failing straight-away.

The second statement deals with the leaf of the tree. When there is only one element in the list of operands (determined by `[X|[]]`) the 'leaf' gets returned in the tree as `value(X)`, and the actual value is clearly `X`.

The third statement deals with lists of more than one element, i.e. those where operations can be applied. Note that we know this predicates is being passed a list with more than one element, because the previous two statements used the cut (`!`), to indicate the zero-length and unary length cases have been fully dealt with.

We use the built-in predicate `append` to enable us to easily split the given list into left and right branches of the tree.

Given the statement `append(As,Bs,Cs)`, if `As` and `Bs` were bound then `Cs` would be the concatenation of the previous two. However when `Cs` is bound (as in this case with `Xs`) and the others are not, append returns all the possibilities of `As` and `Bs` which, when concatenated, would form `Cs`.

Given two lists (`L` and `R`), we have operands that can go on to be used in the left branch and right branch of the expression tree respectively. After dismissing the case when `L` is `Xs` (`R` is empty), which would cause an infinite recursion, the left tree (`TL`) is built up. This is done as a recursive call on `findTree`, but this time just using part of the original list of operands. Similarly the right tree (`TR`) is also built up.

Finally, given the two sub-trees `TL` and `TR`, all operations can be applied on them, and we return their value `V`, and the string of the equation (`O`) that is represented by that operation being applied on the subtrees. This all gets magically bound to the result tree in the bound response `tree(O, TL, TR)`.

makeExpr

This is used by `treeExpr` when the program comes to print out the final result. It concatenates a left expression string (`L`), operation string (`O`) and right expression string (`R`), together with wrapping brackets, to bind to a final expression, `E`.

treeExpr

This is called once the final result goal has been found, and will build up an expression string from the result tree. It takes the form `treeExpr(treeOrLeaf, expressionString)`.

The first statement deals with the leaf case, where the value is simply represented in string form.

The second statement deals with the tree. It calls itself recursively to find the expression for the left and right branches of the tree (`L` and `R` respectively). It then calls `makeExpr` to form the final expression from the left branch's expression (`Le`), the operation (`O`) at the root of the tree, and the right branch's expression (`Re`).

delta_length

`delta_length` is used at the start of the program so that the smallest sets of numbers are initially checked to see if they can be used to form the goal. It provides the `compare(Operator, Operand1, Operand2)` interface that is required by the built-in predicate `predsort` (see later).

`length(List, Length)` is a built-in predicate that binds `Length` to the length of `List`.

`compare` is a built-in predicate that, in this case, enables us to check if `Xs` is shorter than `Ys`.

The second statement for `delta_length` always returns a greater than operator, even in the cases when the lists are the same length, i.e. we are not worried about sort order for equal length lists.

countdown

This predicate forms the control of the game. The first statement tries to match the given goal. The second statement is the catch-all, if there is no solution.

Taking the lines in the first statement one by one, the following is performed:

1. Generate a list of all subsets (`Xss`) of the list of the six numbers provided (`[V1, V2, V3, V4, V5, V6]`).
2. Sort the list of subsets into order based on the length of the subset. The sorted list is `Yss`.
3. Take each subset (`Ys`) in turn.
4. For each of the subsets (`Ys`) we look at every permutation of numbers (`Y1s`). Note that `permutation` is another built-in predicate. We look at permutations of numbers because the order of numbers in an expression is important, for example we need to try A / B and B / A.
5. Now find a tree whose value matches the supplied goal. This is the point mentioned earlier, where `findTree` is called with the already bound `Goal` value. This step of the predicate fails when the generated tree does not match the goal. Prolog then backtracks to try the next tree. Failing to match all those trees the next permutation is tried and, failing all permutations, the next subset of numbers.
   If we do find a tree, whose value matches the goal we know the end result has been found. We declare this with the cut (`!`), which stops the program from trying the 'It's impossible' statement.
6. Finally, having found an expression tree which will match the goal, the expression string is generated so that the result can be seen in a more human readable form.

A final note

SWI-Prolog provides interfaces through to C programs. The original program provided a GUI interface enabling the user to enter the six numbers and goal, before calling the Prolog logic to find the equation.

The program was even sent to the TV program for a mention but, alas, they didn't want to replace Carol by a computer. But then again, who would? ■

Acknowledgements

With much thanks to Mingyuan Mu and Jacqui Alexander for proof reading this article, and to Ric Parkin and Richard Harris who provided valuable feedback.

References and further reading

[Harris] Harris, R., 'The Model Student: A Game of Six Integers [Part 1]', *Overload 95*, Feb 2010.

[Sterling] Sterling, L. & Shapiro, E., *The Art of Prolog*, MIT Press, 1986. ISBN 0-262-69105-1.

[SwiProlog] http://www.swi-prolog.org/

Bug Elimination – Defensive Agile Ruses

Everyone thinks they understand bug economics. Walter Foyle looks again.

A popular work concerning project management is Fred Brooks' *The Mythical Man-Month: Essays on Software Engineering*. While it has many interesting insights into the economics and process of creating software, it is most often remembered for two specific observations, and yet I believe both have been profoundly misunderstood. This is because people are looking at it from a project management point of view, and are missing the economic aspect where project budgets are far more important.

The most famous lesson people remember is that adding people to a late project makes it later. While this is true if you myopically focus solely on your current project, people tend to assume that this is necessarily a bad thing. In fact, a more rounded project manager realises that by increasing your developer count now, you will have a bigger project and hence budget, so putting yourself in a much stronger position to get a more prestigious project in the future. By a process of induction, it can be shown that by repeatedly adding more developers, you can make the project arbitrarily late until you reach the optimum size for you to get the next job.

The other main observation from Brooks is that the cost of fixing a bug increases dramatically the later it is found (see Figure 1). Many have jumped to the simplistic conclusion that this means we should strive to detect and fix the bugs as early as possible. And yet most don't realise the more sophisticated truth that this in turn implies that by introducing the bugs sooner, we have a better chance of detecting and fixing them when they're cheapest. In fact we can improve these chances further by moving all the coding to before the design phase, which normally delays bug detection and so can be thought of as a negative value activity. In addition, delaying the release as far as possible into the future further improves the opportunities for bug detection. Finally, moving testing to after the release frees valuable developer time and has the bonus of reducing the observed bug count.

This all assumes we are blessed with a team productively working on such bugs. Unfortunately, there will often be people who slow down projects rather than advance them. By assigning these people to the unproductive yet politically essential Design Role (or the more trendy role of On Site Customer), we can multiply a negative outcome (too much design delaying implementation and thus delaying the bugs), with a negative ability to do such a thing and thus produce a positive budget item. Just in case anyone objects, the canny Project Manager can use the appropriate spreadsheet magic and to package up these Customer Design Obligations and assign to multiple ongoing projects, so spreading the project risk and making sure that none of their project economies can possibly fail. Ideally these should be stored in Write Only Memory[WOM] along with other Documentation Artifacts to protect the project stability in the face of awkward questions.

Of course, sometimes the customer will insist on a meeting to ask when the delivery will actually be. Unlike normal meetings, which are designed to reduce risk by avoiding doing anything that might cause troublesome progress, these meetings require a swift and firm response. I can recommend sending a memo round immediately to reassure all interested parties that everything is as it ought to be. Ideally this should merely sound reassuring without actually promising anything, as in this masterful example:

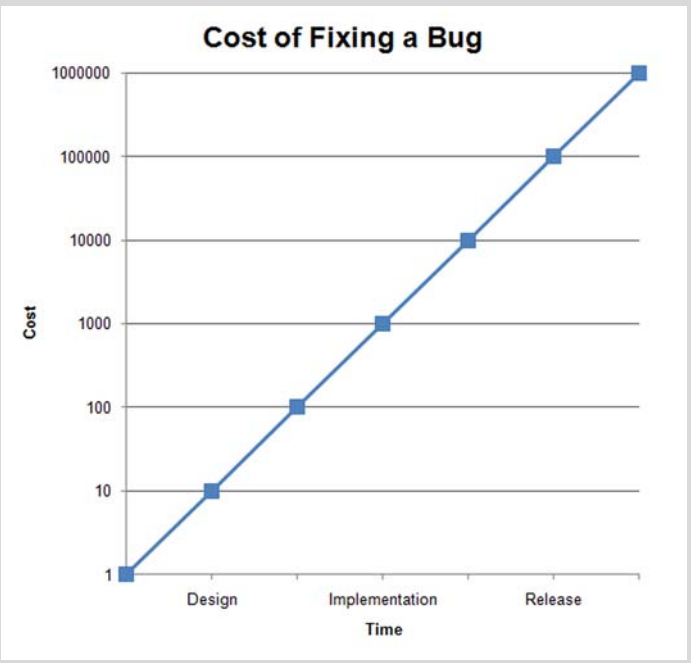

Figure 1

> The meeting was productive and actions have been agreed by both parties, commencing with early technical meetings this week, which if carried through, should lead to the resolution of the issues. Provided there is the expected progress during the coming weeks, both parties are hopeful that it will be possible to indicate by the middle of April the target date for trialling and then operating... [CEN]

By applying such techniques, we can ensure that our projects are large, long, and avoid suffering at the hands of troublesome users. ■

References

[CEN] http://www.cambridge-news.co.uk/Huntingdon-St-Ives-St-Neots/Date-for-opening-expected-after-busway-progress.htm

[WOM] http://en.wikipedia.org/wiki/Write_Only_Memory

Walter Foyle has been consulting on project management for longer than anyone has noticed. By applying his unique perspective on the project life cycle, he has a perfect record of never being on a failed project. He is looking forward to staying on one long enough to see the delivery phase.

A Practical, Reasoned and Inciteful Lemma for Overworked and Overlooked Loners

Popular movements need a rallying cry. Teedy Deigh offers a timely one.

It seems you can no longer promote an idea without framing it as a manifesto of some kind or other. The *Manifesto for Agile Software Development* [Agile] is ultimately to blame for this state of affairs. Many have followed in its wake and its form, some more notable than others, some more notorious. The *Declaration of Interdependence* [DecInd], the *Manifesto for Software Craftsmanship* [Craftmanship], the *SOA Manifesto* [SOA], the *FAIL Manifesto* [FAIL] and undoubtedly countless others have all aped the basic style of the Agile Manifesto and, similarly, without saying much that could be considered provocative.

Teedy Deigh sees herself as the lone voice of reason in the cacophonous landscape of office politics, a radical and revolutionary in the battle for the heart and mind of the knowledge worker, someone prepared to stand against both current and past thinking, or indeed any thinking that threatens to change the way she does things. She's not sure what her colleagues see her as, or that it matters.

Provocation, contradiction and taking a stand used to be what manifestos and declarations were all about. It is perhaps time for another proclamation, one that is likely to reach the core values of developers everywhere, one that is a counterpoint to more thoughtful and considered approaches. Individuals and interactions over processes and tools? This is about individuals without interactions! This is a battle cry from the trenches thrown together over coffee, refined during meetings as an alternative to buzzword bingo and published as an afterthought during a long build. ■

References

[Agile] Manifesto for Agile Software Development, http://agilemanifesto.org

[Craftmanship] Manifesto for Software Craftmanship, http://manifesto.softwarecraftmanship.org

[DecInd] Declaration of Independence, http://pmdoi.org

[FAIL] FAIL Manifesto, http://failmanifesto.org

[SOA] SOA Manifesto, http://soa-manifesto.org

We have been putting in overtime and the code face and, through non-reflective practice and missed deadlines, have come to value:

Coding over **writing any documentation whatsoever**

Debugging over **unit testing**

Singletons over **carefully reasoned, loosely coupled design**

Voice over **IP**

That is, while the items on the right look like hard work and sound quite boring, the items on the left offer identifiable short-term gains and the promise of surprise, mystery and continued employment in the longer term.