overload 101

FEBRUARY 2011 £3

Queue with Position Reservation

We introduce an interesting data structure, particularly useful for multithreaded applications.

Floating Point Blues

We continue our series on numerical computing, this time looking at rationals.

The Agile 10 Steps Model

A look at how Requirements Management fits into Agile processes.

Overused Code Reuse

It's tempting to (re)use someone else's code rather than write it yourself. We look at the hidden pitfalls of repurposing code.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 101

February 2011

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 102 should be submitted by 1st March 2011 and for Overload 103 by 1st May 2011.

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.

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

Ah! The fog is lifting!

Futurology has a dismal track record. Ric Parkin looks at the history of technology predictions.

Prediction is very difficult, especially about the future.

This such a great quote, and yet its providence is uncertain – possible coiners include Mark Twain, Yogi Berra, or Niels Bohr. Whoever actually said it, I do think it it holds a deep truth – we just don't know what's going to happen. It also captures a healthy humility that even if we use our knowledge and expertise to make as good a prediction as we can, reality has an almost perverse delight in proving us wrong, just to keep our hubris in check.

Examples abound in several fields: politics is a fine example – partly because at its core, people are interviewed at short notice on fast moving situations they have limited information on, and yet they feel they have to sound authoritative. A great recipe for putting your foot in it and getting things completely wrong.

And technology has an equally rich seam of such faux pas. I'd like to present a few, give their historical background (or lack of), and consider their deeper truth.

Everything that can be invented has been invented.

– Charles H. Duell, Commissioner US Patent Office

This is trotted out to mock the idea that people think science and society have reached a pinnacle, and there's nothing new to know. Unfortunately this quote appears to be a complete fabrication. There's a similar example from the late 1800s about all of physics being within their grasp, just as Quantum Mechanics and Relativity burst upon the scene. This too is most likely an exaggeration, as the problems that led to those breakthroughs were well known.

The goose that laid the golden eggs, but never cackled

– Churchill

This one is true, although is more of a description than a prediction. This is how Churchill described the Bletchley Park codebreakers, and sums up both of their great achievements – how they achieved the amazing and broke the codes, and yet despite the industrial scale of the work there, the secret was kept until the 70s – probably much longer than even Churchill could have imagined. An example of how useful it was turned up this week, involving the decrypted cables that showed that the Germans had fallen for the D-day bluffs [Fooled]. Just knowing the deception was working allowed the invasion to go ahead with greater confidence and far less loss of life. Golden eggs indeed.

I think there is a world market for maybe five computers

– Thomas John Watson, President of IBM

Again there is no evidence that he actually said it, although interestingly, if he had he'd have been right for around a decade! Recall that until the 70s, computers were huge, expensive machines that relatively few companies could afford, let alone individuals, generally used for quite specialised tasks – calculations for nuclear weapons research, some scientific modelling, and eventually business tasks [LEO] so it was not actually that ludicrous a prediction, until the cost dropped to the point where computers could become ubiquitous, and started to be used for things that couldn't even be imagined back then – think of the advent of computer graphics, and wireless networking. This illustrates that our predictions are shaped by what we know at the time, and that our ideas about what is possible are going to be limited by that.

640K ought to be enough for anybody

– Bill Gates

Again this doesn't appear to have been uttered. But I think the reason we'd like it to be true is because that 640K limit caused so much pain over the years, and people want somebody to blame (and feeling superior to the wildly successful Gates is a bonus). While the jump from 64K or so of memory to 640K in the IBM PC must have seemed like a big leap, an increase of x10 would not last long in the face of ever more inventive tasks for computers to do, and even the first PCs shipped with a large fraction of this limit, so Moore's Law would allow the limit to be reached within a year or two. Getting around this limit did support a small industry of companies inventing various tricks to increase the usable memory though. One clever ruse I remember was the use of extended memory, where blocks of memory could be mapped in and out of that usable 640K – known as conventional memory [Conventional] – via sophisticated 'pointers'. One interesting consequence of this was that a `segment:offset` 'pointer' might not be valid and cause a hardware fault if you tried to load it into the appropriate registers. Note that you didn't even have to dereference the pointer, just load an invalid segment value into the segment register. See for example [MIT]. This is one example of why in C and C++ even looking at an invalid pointer value is Undefined Behaviour. Many people still think that you have to dereference it to trigger UB, but this example shows you don't even need to go that far. One consequence was that you had to be careful with pointer arithmetic so that intermediate values were valid, and avoid falling off the end of an array.

Even when 32-bit flat addressing relieved us from that particular problem, it wasn't to last – fairly recently memory has got cheap enough that you can install more than a 32-bit pointer can address (in practice, operating system limitations have made the limit even lower, so for example 32-bit Windows can only use a maximum of 3GB). Are we in for another round of painful segmented pointer pain? Thankfully not – chips and operating systems have been developed for years to be able to use 64-bit pointers which can use much more memory with emulators to run legacy 32-bit

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.

programs, so the transition should be much smoother. Still, 64bit pointers are twice as large so there are potential data size and storage issues too. Even in this day and age, it's worth being mindful of storage and communication size – a reason that the complex UTF-8 character encoding is commonly used even though the UTF-32 format is much simpler to program with.

One lesson to be learned from these examples is that decisions on the representation of things can have long term consequences, and can cause a lot of pain and effort to fix or transition to the next stage. Even good choices at the time can eventually become a problem. Consider for example, the Y2K problem – a space saving representation of two digit years was a good choice back in the 60s and 70s, but the code and data persisted for longer than anyone really expected, and a lot of time and effort was expended to fix the problem – rather successfully I might add, although that did lead to wild accusations that it was never really a problem. The unix 2038 problem is similar but has been spotted a long way in advance, so solutions are already in progress to avoid many of the problems [2038].

A current problem is just hitting now though – the central internet address system is about to hand out the last blocks of IP numbers to the regional authorities [APNIC]. They in turn will continue to issue addresses until they run out, with estimates of when this will happen being as close as September of this year. See figure 1 for a graph showing the unassigned addresses and figure 2 for assignment rates (note how the number being assigned have shot up recently – looking carefully this looks likely to be due to a combination of the surge in popularity of internet enabled smartphones in North America and Europe, and growth in demand in the Asia-Pacific region).

However, this problem has been anticipated and some systems are already starting to use the next version [IPv6] which will have room for vastly more addresses (2^{128} as opposed to IPv4's 2^{32}). There are concerns that this is not happening fast enough, with many people not even realising they may need to do something. It's not totally clear to me what sort of problems could be expected – there are several proposed strategies that could be used as the final numbers are allocated, perhaps by reallocating no longer used blocks – but it might become more difficult to get addresses for new businesses from providers that are not IPv6 ready. But I doubt that the internet will break – there are much easier ways of doing that [ITCrowd].

References

[2038] http://en.wikipedia.org/wiki/Year_2038_problem

[APNIC] https://www.apnic.net/publications/news/2011/delegation

[Conventional] http://en.wikipedia.org/wiki/Conventional_memory

[Fooled] http://www.bbc.co.uk/news/magazine-12266109

[IPv6] http://en.wikipedia.org/wiki/IPv6

[ITCrowd] http://www.youtube.com/watch?v=wrQUWUfmR_I

[LEO] http://en.wikipedia.org/wiki/LEO_%28computer%29

[MIT] http://pdos.csail.mit.edu/6.828/2006/readings/i386/s06_03.htm

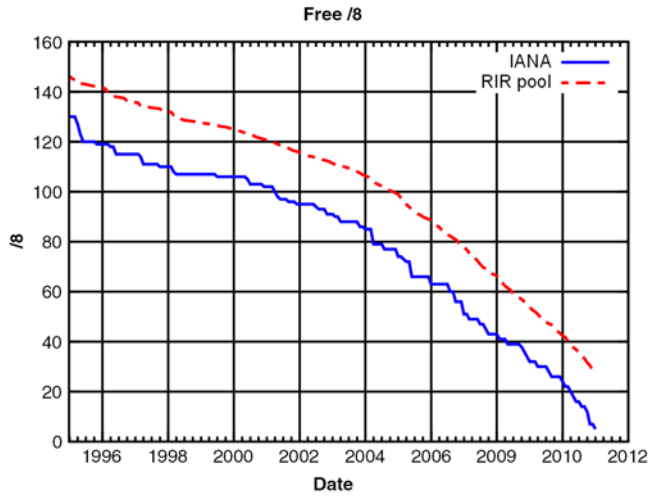


Figure 1

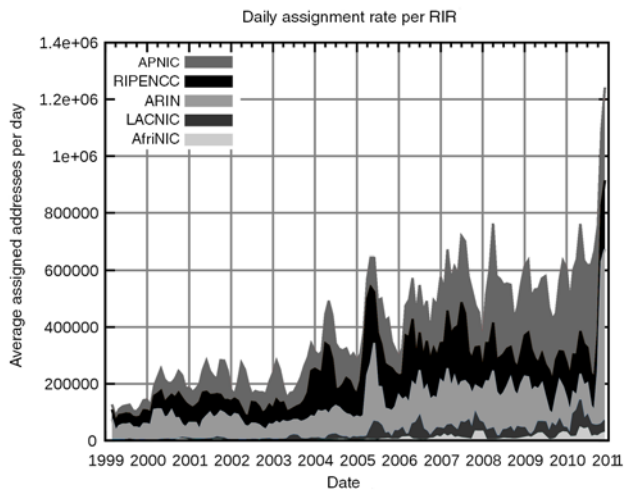


Figure 2

Queue with Position Reservation

Multiple threads can make processing a message queue faster. Eugene Surman needs the right data structure.

For the past five years I have mostly been developing multi-threaded messaging applications. While they were all quite different, there was one particular situation that kept recurring: sometimes it was required to maintain the sequential order of incoming and outgoing messages, even though they were being handled by multiple threads concurrently, and not necessarily in the same exact order they were received. I searched for a solution in many ready-made messaging libraries, but did not find anything satisfactory. So, I had to resort to developing a solution of my own: the **PRQueue** – a Queue with Position Reservation (or 'seat reservation').

PRQueue is implemented in C++ using two STL **deques** and the pthread library. Two simple classes – **Mutex** and **Lock** are used in the example to demonstrate the logic. A sample message is represented by the **StringMsg** class, and the **QueueTest** class is used as a test-bed application.

I chose **deque** as a main building block of the design because it has all necessary operations (including **operator[]**) to implement **PRQueue**. In particular, it's important that the **push_back()** and **pop_front()** operations do not invalidate pointers and references to other elements of the **deque**.

Here is a simple example of how **PRQueue** can be utilized. Let's say we need to log a stream of large multi-field messages. Converting numeric fields to text strings is a slow process that is not mission-critical, so we decided to offload this task to dedicated threads that will generate the log.

Initially, the processing diagram may look like figure 1.

Since the core processing of the messages takes place in multiple threads, the messages may be ready in an order that is different from the original input queue order: if, for example, one thread takes a message off the input queue and goes to sleep, while another thread takes the next message, runs to completion and places the processed message in the output queue, ahead of the first thread. As a result, the log entries may appear out of order. We assume that logging must be done after the messages are processed by the core routines.

Listing 1 is an example illustrating this point. I use the standard STL **queue** and 3 threads. This generates the output shown in Figure 2.

Using **PRQueue** the above scenario will be avoided. It will make sure the order of messages in the output queue matches the order that existed in the input queue, regardless of the order in which the core routines finish processing the messages.

The basic logic behind **PRQueue** is simple: when the next message is taken off the input queue, still inside the lock, the next push-back position, or 'seat', for the output queue is acquired. The lock is then released and the processing continues. After a message is fully processed the previously acquired position is used to place the message into the output queue.

Eugene Surman received a degree in Radio Engineering from Moscow College of Electro Communication. He has been programming C/C++ for over 20 years and currently is a senior software developer for Knight Capital. His personal software interests are scripting languages. He can be reached at esurman@inch.com

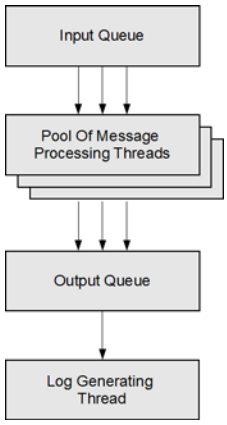


Figure 1

Figure 3 shows the previous example re-written using **PRQueue**. The order of the messages in the log is now perfectly preserved.

PRQueue is constructed using two deques: 'data' and 'filled'.

An element of 'filled' **deque** is an indicator showing that the position is filled with data and can be popped from **PRQueue**. A wrapper class **DataQueue** is a holder of 'data' and 'filled' **deque**s. The **PRQueue**

```
...
QueueTest quetest(3);
int i1 =0;
for( int i =10000; i; i--) {
  quetest.push( "| %d", i1++ );
  quetest.push( "- %d", i1++ );
}
...
```

Listing 1

```
Th# Time-stamp                    Msg#
1:  101108 15:04:49.576167  -   5243
3:  101108 15:04:49.576170  |   5244
1:  101108 15:04:49.576174  -   5245
3:  101108 15:04:49.576177  |   5246
3:  101108 15:04:49.576182  |   5248   // out
2:  101108 15:04:49.571945  |   4338   // of
1:  101108 15:04:49.576179  -   5247   // order
3:  101108 15:04:49.576188  -   5249
2:  101108 15:04:49.576189  |   5250
1:  101108 15:04:49.576191  -   5251
```

Figure 2

the messages may be ready in an order that is different from the original input queue order

```
Th#  Time-stamp                    Msg#
2:   101108 15:04:49.571945  |    4338
...
...
1:   101108 15:04:49.576167  -    5243
3:   101108 15:04:49.576170  |    5244
1:   101108 15:04:49.576174  -    5245
3:   101108 15:04:49.576177  |    5246
1:   101108 15:04:49.576179  -    5247
3:   101108 15:04:49.576182  |    5248
3:   101108 15:04:49.576188  -    5249
2:   101108 15:04:49.576189  |    5250
1:   101108 15:04:49.576191  -    5251
```

Figure 3

```
// The function 'process_msg' is executed by every
// spawned input thread. The signature corresponds
// to the pthread_create 'start_routine'
// File prqueue.cpp

void* process_msg( void* arg)
{
  int thidx = ++Thidx;
  QueueTest* quetest =(QueueTest*)arg;
  Msg* msg;
  PRQueue< Msg*>::position pos;

  cout << "Input thread=" << thidx <<
     " started" << endl;

  for(;;)
  {
    // Wait for the next available message in
    // input queue and pop it up, get the next
    // push position reserved in output queue
    quetest->input_que.pop(
       msg, quetest->output_que, pos);

      // Process message
      msg->process( thidx);

      // Push processed message into output queue
      // using reserved position
      quetest->output_que.push( msg, pos);
  }
  return NULL;
}
```

Listing 2

methods are for the most part 'mutexed' wrappers of **DataQueue** methods.

The design allows us to separate/hide thread safety code from the actual implementation, so the user shouldn't be concerned with writing any locking/unlocking logic.

Let's discuss **PRQueue**'s functionality in a bit more detail.

The **PRQueue pop** method does two things: it pops data from the input queue and reserves a push position in the output queue. The **push** method uses the previously reserved position to save data into the output queue.

For testing **PRQueue** with multiple threads a function **process_msg** is executed by every spawned thread. It pops a **StringMsg** from the input queue, processes the message by calling the **StringMsg::process()** method, and pushes the message out. (See Listing 2.)

The **pop** method is not only waiting for the next message to arrive in the input queue, it also checks if the message is ready to be popped by looking at the element of the 'filled' queue. If data is not filled yet, **pop** will go back to sleep and wait.

Pop logic (Listing 3):

- Lock input queue
- If input queue is not empty and top element is filled with data, **pop** it (otherwise release lock and go to sleep)
- Lock output queue
- Reserve bottom position in output queue.
- Unlock output queue
- Unlock input queue

The **push** method copies data to the reserved position of the output queue and sets the 'filled' indicator to true. It also releases threads waiting on a condition variable by sending a notification signal (`prqueue.hpp`) – see Listing 4.

Now, the messages are arriving in the output queue in order. If we want to extend the chain of our processing conveyor further, another **PRQueue** can be added to the end. In the test case above we don't do it: we use a single output thread simply to read processed messages from the output queue and print them out. In that final step, a 'simple **pop**' method was used without its second and third arguments (references to the output queue and position value). See Listing 5.

Now, let's take a look at the auxiliary class **DataQueue**.

As was mentioned before, **DataQueue** is a holder of two STL **deque**s: 'data' and 'filled'. The **DataQueue** also defines 'structure position' and methods where the key steps of position reservation and data popping happen.

The **DataQueue** is included in **PRQueue** as a data-member **m_que** (see Listing 6).

To compile and run **PRQueue** test, use the commands in Figure 4.

the order of messages in the output queue exactly matches the order that existed in the input queue

```
// Pop data from input queue and reserve position
// in output queue file prqueue.hpp
void PRQueue::pop( DATA& data, PRQueue& outque,
   PRQueue::position& pos)
{
  Lock lk( m_mux);

  // Waiting for the message in input queue - pop
  // message
  while( true) {
    if( m_que.pop( data))
      break;
    // either message has not arrived or position
    // is not filled
    wait_while_empty();
  }
  // Reserve position in output queue
  outque.reserve_pos( pos);
}

//
void PRQueue::reserve_pos(
   PRQueue::position& pos) {
  Lock lk( m_mux);
  m_que.reserve( pos);
}
```

Listing 3

```
// Push data using reserved position into output
// queue (prqueue.hpp)
void PRQueue::push( const DATA& data,
   const PRQueue::position& pos)
{
  Lock lk( m_mux);
  m_que.fill( data, pos);
  notify_not_empty();
}
```

Listing 4

```
// The function 'print_msg' executed by final
// single output thread file prqueue.cpp
void* print_msg( void* arg)
{
  QueueTest* quetest =(QueueTest*)arg;
  Msg* msg;

  cout << "Output thread started" << endl;
  for(;;)
  {
   // pop-up message from output queue and print it
    quetest->output_que.pop( msg);
    msg->print();
    delete msg;
  }
  return NULL;
}
```

Listing 5

Conclusion

The queue with Position Reservation (**PRQueue**) presented here could be useful in multi-threaded applications when the order of streaming messages should be preserved. **PRQueue** will make sure that the order of messages in the output queue exactly matches the order that existed in the input queue, because the next push-back position in the output queue is reserved synchronously with taking the message off the input queue. The reserved spot is later filled with data when the message is done processing and ready. ■

Reference

A zip file containing the code is available at:
http://accu.org/content/journals/ol101/prqueue.zip

```
   c++ -I. prqueue.cpp -lpthread                # PRQueue test
   c++ -I. prqueue.cpp -lpthread -DSIMPLE_QUE   # SimpleQueue test

   # Try long message
   c++ -I. prqueue.cpp -lpthread -DLONG_MSG
   c++ -I. prqueue.cpp -lpthread -DLONG_MSG -DSIMPLE_QUE

   a.out [number-of-messages]
```

Figure 4

```
// An auxiliary class DataQueue - holder of 'data'
// and 'filled' deques
template< typename DATA> class DataQueue
{

public:
  typedef typename
     deque< DATA>::pointer data_pointer;
  typedef typename
     deque< bool>::pointer filled_pointer;

  // Structure to hold pointers of reserved
  // position
  struct position {
    position() : data_pnt(0), filled_pnt(0) {}
    data_pointer data_pnt;
    filled_pointer filled_pnt;
  };

  // Check if data deque is not empty and front
  // element is 'filled'.
  // Copy front data out, pop-up front elements
  // of both deques
  bool pop( DATA& out) {
    if( m_data_que.empty() ||
       ! m_filled_que.front())
      return false;
    out = m_data_que.front();
    m_data_que.pop_front();
    m_filled_que.pop_front();
    return true;
  }
  // Add dummy elements to the back of both
  // deques.
  // Save pointers of both elements to the output
  // position
  void reserve( position& pos) {
    m_data_que.push_back( m_dummy);
    m_filled_que.push_back( false);
    pos.data_pnt =
       &m_data_que[ m_data_que.size() -1];
    pos.filled_pnt =
       &m_filled_que[ m_filled_que.size() -1];
  }
  // Copy data and set 'filled' indicator by
  // position
  void fill( const DATA& data,
     const position& pos) {
    *pos.data_pnt = data;
    *pos.filled_pnt = true;
  }
```

Listing 6

```
  void push( const DATA& data) {
    m_data_que.push_back( data);
    m_filled_que.push_back( true);
  }

private :
  deque<DATA> m_data_que;
  deque<bool> m_filled_que;
  DATA m_dummy;
};//DataQueue
```

Listing 6 (cont'd)

Why Rationals Won't Cure Your Floating Point Blues

Numerical computing is still proving hard to do accurately. Richard Harris considers another number representation.

In the first article in this series we described floating point arithmetic and noted that its oft criticised rounding errors are relatively inconsequential in comparison to the dramatic loss of precision than results from subtracting two almost equal numbers. We demonstrated that the order in which we perform operations, whilst irrelevant for real numbers, can affect the result of a floating point expression and that consequently we must be careful how we construct expressions if we wish their results to be as accurate as possible.

In the second article we discussed the commonly proposed alternative of fixed point numbers and found that, although it is supremely easy to reason about addition and subtraction when using them, they can suffer even more greatly than floating point numbers from truncation error, cancellation error and order of execution.

Rationals

So, can we do any better?

Perhaps if we were to implement a rational number type, in which we explicitly maintain both the numerator and the denominator, rather than declare by fiat that we are working to some fixed number of decimal places or significant figures.

The rules of rational arithmetic are pretty straightforward. Given two rationals a_0/b_0 and a_1/b_1 we have

$$\frac{a_0}{b_0} + \frac{a_1}{b_1} = \frac{a_0 b_1 + a_1 b_0}{b_0 b_1}$$

$$\frac{a_0}{b_0} - \frac{a_1}{b_1} = \frac{a_0 b_1 - a_1 b_0}{b_0 b_1}$$

$$\frac{a_0}{b_0} \times \frac{a_1}{b_1} = \frac{a_0 a_1}{b_0 b_1}$$

$$\frac{a_0}{b_0} \div \frac{a_1}{b_1} = \frac{a_0 b_1}{b_0 a_1}$$

One enormous advantage of rational numbers is that, provided we do not overflow the integers representing the numerator (the top of the fraction) and the denominator (the bottom) the order of execution of these arithmetic operations is irrelevant; the answer will always be the same. Given that we have gone to great lengths to create an integer type that cannot overflow, this behaviour will prove rather useful.

The only thing we need to watch out for is the fact that there are many ways of writing down the same number; 1/2, 2/4 and 3/6 all represent the same number, for example. We shall ensure that our representation is unique by insisting that the numerator and the denominator are the smallest numbers that yield the same rational, or equivalently have no common factors, and that the denominator is positive.

The latter condition is relatively straightforward to maintain. The former requires an algorithm to determine the highest common factor, or HCF, of a pair of numbers, the greatest positive integer that wholly divides both. We can subsequently divide out that factor and return any rational to its simplest form. Fortunately one such algorithm has been handed down to us from antiquity, courtesy of the great Euclid and it proceeds as follows.

Euclid's algorithm

If the two numbers are equal, their value is the HCF.

If the smaller exactly divides the larger, the smaller is the HCF.

Otherwise, divide the larger by the smaller, and make note of the remainder. The HCF of the original numbers is equal to the HCF of the smaller number and the remainder.

In mathematical notation this can be expressed as

$$\begin{aligned} &\textit{if} && x_1 = a \times x_0 + b \\ &\textit{where} && a > 0 \wedge 0 < b < x_0 \\ &\textit{then} && HFC(x_1, x_0) = HFC(x_0, b) \end{aligned}$$

Recursively applying these rules is guaranteed to terminate and we can thus determine the HCF.

For example, applying the Euclidean algorithm to 2163 and 1785 yields the following steps

$$\begin{aligned} 2163 &= 1 \times 1785 + 378 \\ 1785 &= 4 \times 378 + 273 \\ 378 &= 1 \times 273 + 105 \\ 273 &= 2 \times 105 + 63 \\ 105 &= 1 \times 63 + 42 \\ 63 &= 1 \times 42 + 21 \\ 42 &= 2 \times 21 \end{aligned}$$

and hence the HCF of 2163 and 1785 is 21, a fact that is clear if we look at their prime factorisations.

$$\begin{aligned} 2163 &= 3 \times 7 \times 103 \\ 1785 &= 3 \times 5 \times 7 \times 17 \end{aligned}$$

As it happens, this is simply a special case of the more general result that for *any* integers x_0, x_1, a and b where

$$x_1 = a \times x_0 + b$$

then x_0 and b must have the same highest common factor as x_0 and x_1, as shown in derivation 1.

As a consequence, it should not be surprising that the algorithm converges faster if we round the result of the division to the nearest integer rather than round down, consequently admitting negative remainders, and use the absolute value of the remainder in the following step.

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.

one such algorithm has been handed down to us from antiquity, courtesy of the great Euclid

Applying this optimisation to the same pair of numbers yields the same result in fewer steps.

$$2163 = 1 \times 1785 + 378$$
$$1785 = 5 \times 378 - 105$$
$$378 = 4 \times 105 - 42$$
$$105 = 2 \times 42 + 21$$
$$42 = 2 \times 21$$

A rational class

Now that we have described the various arithmetic operations, and the scheme that we shall use to ensure that each rational has a unique representation, we are ready to actually implement it. Listing 1 illustrates the class definition of our rational number type.

The first thing we shall need is a helper function to compute the HCF of a pair of positive integers as given in listing 2.

Note that we capture both termination conditions by checking whether the absolute remainder, now stored in x_1, is equal to 0. This will be true both if the smaller number is equal to or wholly divides the larger.

We implement the more efficient version of the algorithm by checking whether the remainder is greater than half the divisor. If it is, then the absolute value of the remainder of the rounded closest, rather than rounded down, division is simply the divisor minus the remainder.

Proof of shared common factors

First, let us assume that x_0 and x_1 share a common factor of c. We can therefore rewrite the equation as

$$cx_1' = a \times cx_0' + b$$

for some x_0' and x_1'.

Now since the left hand side is wholly divisible by c then so must the right hand side and furthermore since the first term on the right hand side is wholly divisible by c then so must be the second term, allowing us to express the equation as

$$cx_1' = a \times cx_0' + cb'$$

Second, let us assume that x_0 and b share a different common factor of d. We can now rewrite the equation as

$$x_1 = a \times dx_0'' + db''$$

for some x_0'' and b''.

But now the right hand side is wholly divisible by d and so therefore must be the left hand side.

Hence any factor shared by x_0 and x_1 must be shared by x_0 and b, and any factor shared by x_0 and b must be shared by x_0 and x_1 and that they must consequently have the exactly the same highest common factor.

Derivation 1

```
template<class T>
class rational
{
public:
  typedef T term_type;

  rational();
  rational(const term_type &x);
  rational(const term_type &numerator,
           const term_type &denominator);

  const term_type & numerator() const;
  const term_type & denominator() const;
  int        compare(const rational &x) const;
  rational & negate();
  rational & operator+=(const rational &x);
  rational & operator-=(const rational &x);
  rational & operator*=(const rational &x);
  rational & operator/=(const rational &x);

private:
  rational & normalise();
  term_type numerator_;
  term_type denominator_;
};
```

Listing 1

```
template<class T>
T
hcf(T x0, T x1)
{
  if(x0<=0 || x1<=0)
    throw std::invalid_argument("");

  if(x0<x1)
    std::swap(x0, x1);

  do
  {
    const T div = x0/x1;
    const T rem = x0 - div*x1;

    x0 = x1;
    if(rem+rem<x1)  x1  = rem;
    else            x1 -= rem;
  }
  while(x1!=0);

  return x0;
}
```

Listing 2

we must multiply the numerators and denominators during comparison which, for fixed width integer types, introduces the possibility of overflow

We can see that this is true by considering the implications on the remainder of increasing the result by 1. In mathematical notation, the initial step is

$$d = \lfloor x_0 / x_1 \rfloor$$
$$r = x_0 - d \times x_1$$

where the odd looking brackets mean the largest integer less than or equal to their contents.

The new remainder is equal to

$$x_0 - (d+1) \times x_1 = r - x_1$$

which is guaranteed to be negative, meaning that the absolute value of the new remainder must be $x_1 - r$.

We could improve performance a little for **bignum**s by overloading this function to exploit the fact that their division helper function also calculates the remainder. However, since our division algorithm is $O(n^2)$ in the number of bits and our multiplication algorithm is only $O(n^2)$ in the number of digits, it would probably not make that much difference in most cases.

```
template<class T>
rational<T> &
rational<T>::normalise()
{
  if(denominator_==0)
    throw std::invalid_argument("");

  if(denominator_<0)
  {
    numerator_   = -numerator_;
    denominator_ = -denominator_;
  }

  if(numerator_==0)
  {
    denominator_ = 1;
  }
  else
  {
    const T c = hcf(abs(numerator_),
                    denominator_);

    numerator_   /= c;
    denominator_ /= c;
  }

  return *this;
}
```

Listing 3

```
template<class T>
rational<T>::rational()
: numerator_(0), denominator_(1)
{
}

template<class T>
rational<T>::rational(const term_type &x)
: numerator_(x), denominator_(1)
{
}

template<class T>
rational<T>::rational(const term_type &numerator,
                      const term_type &denominator)
: numerator_(numerator), denominator_(denominator)
{
  normalise();
}
```

Listing 4

Next we shall implement the **normalise** member function which we shall use to ensure that our **rational**s are always represented in a standard form, as shown in listing 3. In this form, common factors are removed, the denominator is always positive and shall furthermore be equal to 1 when the numerator is 0.

We shall first call this function in one of the constructors, as given in listing 4. Specifically, we shall not entrust the correct representation to the user when construction from numerator and denominator.

The remaining member functions are equally straightforward which should come as no surprise given the simplicity of rational arithmetic.

The data access member functions, **numerator** and **denominator**, together with the **compare** and **negate** member functions are shown in listing 5.

Note that we must multiply the numerators and denominators during comparison which, for fixed width integer types, introduces the possibility of overflow and, for **bignum**s, unfortunately makes it a relatively costly operation.

The arithmetic operators, given in listing 6, are similarly sensitive to overflow when using fixed width integers and similarly expensive when using **bignum**s. Most irritating is that fact that addition and subtraction are now more sensitive to these factors than multiplication and division.

The problem with rationals

Recall that I mentioned that the square root of 2 is irrational and hence cannot be equal to any integer divided by another. A demonstration of this fact is given in derivation 2.

We cannot therefore exactly represent any such number with our **rational** type. However, it is also true that for every irrational number

```
template<class T>
const rational<T>::term_type &
rational<T>::numerator() const
{
  return numerator_;
}

template<class T>
const rational<T>::term_type &
rational<T>::denominator() const
{
  return denominator_;
}

template<class T>
int
rational<T>::compare(const rational &x) const
{
  const term_type lhs = numerator_ *
     x.denominator_;
  const term_type rhs = denominator_ *
     x.numerator_;

  if(lhs<rhs)  return -1;
  if(lhs>rhs)  return  1;
  return 0;
}

template<class T>
rational<T> &
rational<T>::negate()
{
  numerator_ = -numerator_;
  return *this;
}
```

Listing 5

there are an infinite number of rationals to be found within any given positive distance, no matter how small.

Perhaps we could represent an irrational with one of its rational neighbours?

Proving that the square root of 2 is not rational

Let us assume that there are integers *a* and *b* such that

$$\frac{a}{b} = \sqrt{2}$$

and that we have cancelled all common factors so that their HCF is 1.

Trivially, we have

$$\frac{a^2}{b^2} = 2$$

$$a^2 = 2b^2$$

Now any odd number multiplied by itself results in another odd number, so *a* must be even and hence equal to $2a'$ for some a'. Hence

$$(2a')^2 = 2b^2$$

$$b^2 = 2a'^2$$

But this similarly means that *b* must be even and that consequently *a* and *b* have a common factor of 2; a contradiction.

The square root of 2 cannot, therefore, be rational.

Keep it to yourself though; you might get drowned.

Derivation 2

```
template<class T>
rational<T> &
rational<T>::operator+=(const rational &x)
{
  numerator_    = numerator_   * x.denominator_ +
                  denominator_ * x.numerator_;
  denominator_ *= x.denominator_;
  return normalise();
}

template<class T>
rational<T> &
rational<T>::operator-=(const rational &x)
{
  numerator_    = numerator_   * x.denominator_ -
                  denominator_ * x.numerator_;
  denominator_ *= x.denominator_;
  return normalise();
}

template<class T>
rational<T> &
rational<T>::operator*=(const rational &x)
{
  numerator_   *= x.numerator_;
  denominator_ *= x.denominator_;
  return normalise();
}

template<class T>
rational<T> &
rational<T>::operator/=(const rational &x)
{
  numerator_   *= x.denominator_;
  denominator_ *= x.numerator_;
  return normalise();
}
```

Listing 6

Well, yes we could, but we'd have to decide exactly how distant that rational should be and, whatever distance we choose, we could match its accuracy with a floating point representation of sufficient precision.

So, whilst rational number types are supremely accurate for addition, subtraction, multiplication and division and are consequently not sensitive to the order of execution of these operations, they require no less care and attention than floating point number types the instant we start mucking about with non-linear equations.

I am reluctant to categorise this capable numeric type as a lame duck, but am compelled to observe that, so far as general purpose arithmetic is concerned, it does seem to have a pronounced limp.

Quack, quack, quack. ■

Further reading

[Boost] http://www.boost.org/doc/libs/1_43_0/libs/rational/index.html

Overused Code Reuse

It's tempting to use someone else's code rather than write it yourself. Sergey Ignatchenko reports that 'No Bugs' Bunny recommends caution.

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

First of all, I want to congratulate all fellow rabbits on the Year of the Rabbit, which started on 3rd February. I wish all rabbits all the best in this year, but want to remind you that it is not only a year of great opportunity, but also of great responsibility. Let us try to make the Year of the Rabbit as bug-free as possible! One of the important steps on this road will be understanding the pitfalls of code reuse.

Since ancient times, using pre-existing code from somewhere else has been seen as a Holy Grail by project management. 'Why develop functionality ourselves if we can buy it?' With open source software becoming ubiquitous, the temptation became even stronger: the argument 'Why spend money if we can get it for free?' is as strong as it can possibly get for a manager. On the developers' side the temptation to reuse is also rather strong: 'Hey, we can use this neat 3rd-party class and get this very cool feature!' (It often happens that nobody has actually asked for this feature, but that rarely stops this kind of reuse.)

While I certainly don't want to claim that all code reuse is inherently evil, it does have significant drawbacks which are often overlooked. In this month's column I will try to describe these issues, which might cause lots of problems down the road, and to provide some reflections on the question 'when to reuse'.

Toll of code reuse: real-life disasters

For some examples where things went really badly, I will describe two well-known cases when code reuse has significantly contributed to disasters which happened because of software bugs.

In 1982, a new radiation therapy machine Therac-25 [Therac-25] was developed by Atomic Energy of Canada Ltd. Therac-25 was a further development of two previous models, Therac-6 and Therac-20, and (*naturally*) it had been decided to reuse some software from the previous models [Leveson]. The hardware was a bit different though, and in particular, while the Therac-20 had hardware interlocks to prevent the software from activating the beam in the wrong position, the Therac-25 didn't have such interlocks relying instead solely on the software. The software that was reused actually had a difficult to find bug, which had never manifested itself on Therac-20 because of the hardware interlocks. Reusing it on Therac-25, where hardware wasn't available to prevent an overdose, resulted in at least 6 confirmed cases of massive (100x) radiation overdose, and in 3 to 5 (estimates vary depending on source) deaths due to this bug. Some may argue that 3 deaths is nothing compared to the number of deaths in car accidents every day, but would *you personally* like to be responsible for them? I hope not.

In 1996, an Ariane 5 rocket self-destructed 37 seconds into its first launch, with estimated damages at least in the order of several hundred million euros [Robinson]. An investigation [*Ariane Inquiry*, Robinson] has shown that it had been caused by re-using a subsystem of the software of Ariane 4. The bug was within a piece of code which wasn't necessary for Ariane 5 but which was (*naturally*) maintained 'for commonality reasons' [Ariane Inquiry]; the bug had never manifested itself in Ariane 4 because of different flying dynamics.

I cannot tell if it was a mistake to reuse any code in these two cases (though there are indications it was – for example, [Leveson] says 'The reuse of Therac-6 design features or modules may explain some of problematic aspects of the Therac-25 software design'), but what is clear is that *careless* reuse is what essentially caused both of these disasters. Two observations can be made out of these two cases.

The first one is the following rule of thumb:

If reusing, one needs to carefully consider the new environment where the code is moved; failure to do so can be catastrophic.

Another is the following (based on the Ariane 5 failure above, but is also supported by personal experience):

When reusing, it is often difficult to understand how much code you've just added.

Resource bloat

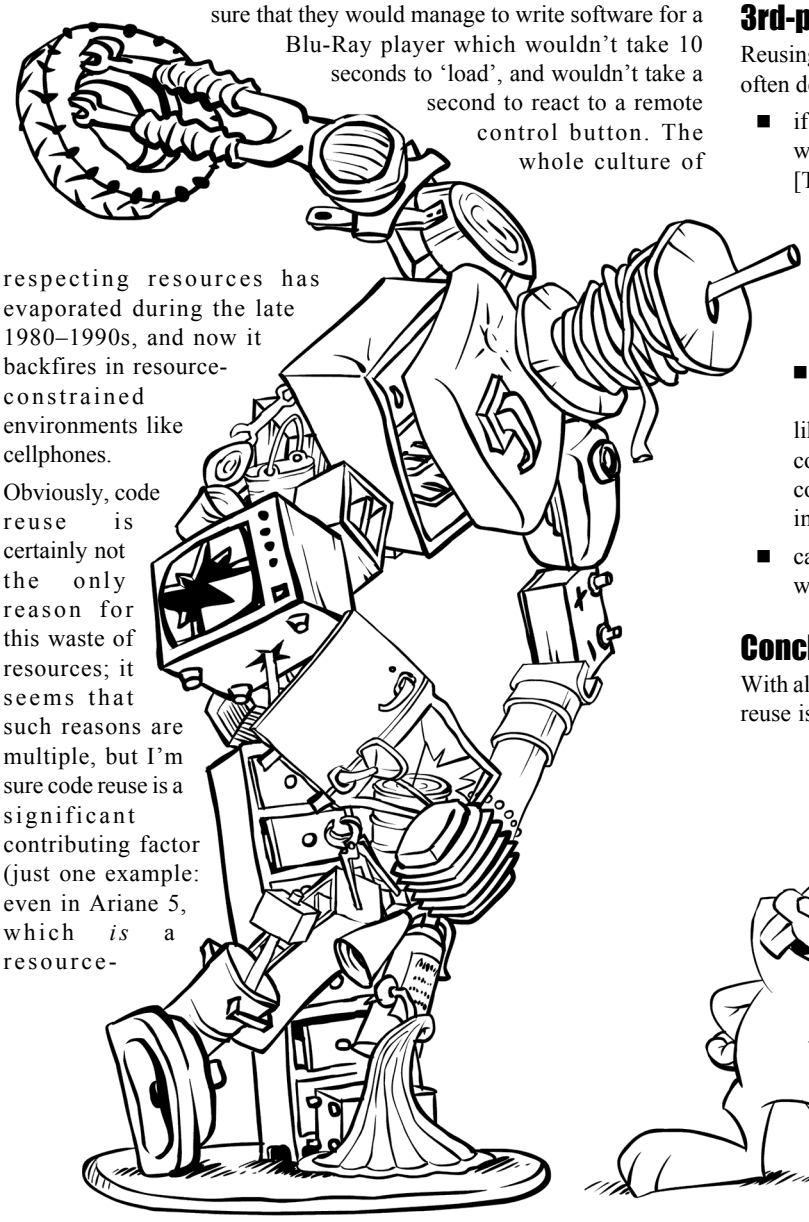
In his presentation [Martin], Robert Martin stated that since the time of the PDP8, hardware has been improved by 27 orders of magnitude. While I can't comment on the exact numbers, it is obvious that improvements in hardware within last 20 years were HUGE. Does anybody remember the ZX Spectrum home computer? It had 3.5MHz Z80 CPU (this is not only without floating point at all, this is without hardware multiplication!), and 48KB RAM (of which 7KB was video RAM); no HDD to swap to, not even a floppy disk, everything needed to be loaded from tape into RAM. And still, developers were able to do wonders with this hardware. One game of the time, *Elite*, contained an inter-planetary trading system (with price based on supply and demand), real-time 3D space fights (OK, it was contour-only 3D, but keep in mind the restrictions), several special missions, and a galaxy map of a few thousand planets – all within 41KB of RAM (code and data combined), on a CPU which is 1000+ times slower than today's. There were also compilers, word processors and spreadsheets. One starts to wonder – if it was possible to do these kind of things in 41KB, how much better should be software which can use 41MB! Unfortunately, it is not the case. Modern software can do (as a rule of thumb) absolutely nothing in 41K and just a few minor things in 41MB;

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

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

Reusing 3rd-party code introduces dependencies. Such dependencies are often detrimental for several reasons

for example, the Eclipse IDE requires as much as 512MB RAM by default, this is 10000 times more than ZX Spectrum had. No doubt that Eclipse has more capabilities than ZX Spectrum era development tools, but is it *four orders of magnitude* more? I don't think so. One can argue that these days RAM is cheap, so who cares about all this bloat? The answer is: I do, at least, because if the guys who wrote Elite were around to develop for the more constrained modern environments, I'm pretty sure that they would manage to write apps for a cellphone which wouldn't feel sluggish on a mere 1GHz CPU (it is still 300 times faster than the ZX Spectrum had), or 512MB RAM (which *is* four orders of magnitude more). Also I'm pretty sure that they would manage to write software for a Blu-Ray player which wouldn't take 10 seconds to 'load', and wouldn't take a second to react to a remote control button. The whole culture of respecting resources has evaporated during the late 1980–1990s, and now it backfires in resource-constrained environments like cellphones.



Obviously, code reuse is certainly not the only reason for this waste of resources; it seems that such reasons are multiple, but I'm sure code reuse is a significant contributing factor (just one example: even in Ariane 5, which *is* a resource-constrained environment, they decided to keep useless code 'for commonality reasons'; even if it wouldn't crash the whole thing, it would still be a waste of CPU resources). But usually the reason is simpler than that: as we have seen above, *it is often difficult to understand just how much code you've added*. Therefore, it often happens that just one tiny DLL/.so is used, which in turn calls a dozen other DLLs/.so's, and so on. Did you know, for example, that loading `MFC42.DLL` implicitly loads not only OLE/COM, but also loads the print spooler, even if you never use any of it? Or how many DLLs depend on `SHDOCVW.DLL`?

3rd-party code and dependencies

Reusing 3rd-party code introduces dependencies. Such dependencies are often detrimental for several reasons:

- if there is a bug in the 3rd-party code, it is still *your* application which will crash, and your users will blame *you* (see also [ToDllOrNotToDll]).
- if there are changes to the 3rd-party code, you are at their mercy to keep the APIs stable. Moreover, it is not only their understanding of the APIs that should not be affected by change, it is also *your* understanding (and they are not always the same thing).
- if a 3rd-party API does not exactly correspond to your requirements (which is almost always the case), relying on it will likely lead to lower cohesion and higher coupling, increasing overall code rigidity. While these effects can be mitigated by creating 'glue code' to isolate 3rd-party code, it is an extra cost and is rarely done in practice.
- careless reuse of 3rd-party code can easily lead to 'licence hell', with a need to handle lots of potentially incompatible licences.

Conclusion

With all those problems with code reuse outlined above, does it mean code reuse is always wrong? Not at all, there are perfectly legitimate uses for it. For instance, I don't mean that if you're writing a business application, you should start by writing your own operating system or database. The reason why I listed all those problems is to illustrate that reusing code from different projects or (even worse) from 3rd-parties SHOULD NOT be taken lightly, but only with a full understanding of all the implications and consequences. Individual analysis is required in each case, but there are several rules of thumb I and many of my fellow rabbits use, which can be a reasonable starting point:

- If reusing, one needs to carefully consider the new environment where the code is moved; failure to do so can be catastrophic.

reused code can be free, integration with your own code is never free

One example of reuse from non-software field would be to reuse bridge piers when building a new bridge in place of the old one. While such reuse is possible and sometimes undertaken, it is always preceded by a very careful analysis; such analysis also often reveals that reuse will be dangerous, or more expensive than using new ones, and a new bridge is often built completely separately. Why should software be any different?

- All decisions about reuse of 3rd-party code (this includes code from within the same company, but perhaps from a different project) must only be made after careful consideration at project level; both architectural and legal analyses should be performed before making a decision about reuse.

  If you need to make some major decisions (and as discussed, the decision to reuse 3rd-party code is a major one), it requires some formalities; licence issues are a contributing factor too.

- When making decisions about reuse, remember integration costs.

  This rule of thumb is of special importance for managers. While reused code can be free, integration with your own code is never free, and in some cases can exceed the costs of writing code from scratch.

- As a rule of thumb, the lower the level of API, the more chances that it will be suitable for your needs.

  For example, the chances that 'JPEG library' will be exactly what you're looking for, are much higher than that 'business flow handling' will suit your needs; the longer-term chances of the latter are further decreased by likely changes to the business flow logic.

- To avoid increasing code rigidity, if reusing 3rd-party code, think about adding 'glue code' around it. Note that 'dumb' wrappers (wrapping every function 1-to-1) don't tend to help with it, and are essentially useless.

  This can be a tough exercise, but unless you're building something which has a 100% dependency on 3rd-party code, having 'glue code' is paramount to keep software maintainable in the long run. Over the time all kinds of things can happen: 3rd-party code can go out of circulation, a competing product can become better, new management can strike a deal with another vendor. Proper 'glue code' can save you from rewriting the whole program (or at least to reduce the amount of work significantly), but the trick here is to find what kind of 'glue code' is appropriate. As a rule of thumb, it is better to specify 'glue' APIs in terms of 'what we need to do' (as opposed to 'what *this code* can do for us'). It essentially rules out 'dumb' wrappers (where the 'glue' API merely mirrors the functionality of the API being wrapped), which are indeed pretty much useless.

- If you're not writing something inherently reusable, like an OS or public API, don't write for reuse – reuse existing code instead.

  It has been mentioned by both [Brooks] and [Kelly], that writing code aimed for reuse is three times more expensive than writing single-use code. It is in line with practical observations by fellow rabbits: among other things, when writing code which is aimed for code reuse it can be not so easy to adhere to the 'No Bugs' Axe' principle, and deviations from it are likely to lead to 'creeping featuritis' [NoBugsAxe].

- Know what exactly you're including, what resources it takes ,and what are the implications of the code being reused.

  Maybe reused code includes a call which is specific to Win7, and you're required to support XP? Or maybe it will not run unless some specific version of some other library is installed? Or maybe you're writing an Internet application, and the reused code issues 100 successive RPC calls which you won't notice over your LAN but which will cause delay of several seconds across a transatlantic link? If you are re-using code, it is *your* responsibility to make sure it is suitable for *your* purposes.

It is important to note that while some of these points do not apply to 'internal reuse' (such as placing code in functions and calling them from many different places), some of these 'rules of thumb' are still essential regardless of reuse being 'internal' or 'external'. In particular, 'new environment', 'integration costs', and 'know what exactly you're including' points stand even for 'internal reuse'. If reusing internal small well-defined functions, these points may be trivial to address, but as the complexity of reused code grows, analysis can become more complicated and the lack of such analysis may cause significant problems down the road. ■

References

[Adams] http://en.wikipedia.org/wiki/Lapine_language

[Ariane Inquiry] *Ariane 501 Inquiry Board Report*, http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[Brooks] *The Mythical Man Month and Other Essays on Software Engineering*, Frederick P. Brooks Jr.

[Kelly] 'Reuse Myth – can you afford reusable code?', Allan Kelly, 2010, http://allankelly.blogspot.com/2010/10/reuse-myth-can-you-afford-reusable-code.html

[Leveson] *Medical Devices: The Therac-25*, Nancy Leveson, http://sunnyday.mit.edu/papers/therac.pdf

[Loganberry] David 'Loganberry', *Frithaes! – an Introduction to Colloquial Lapine!*, 'Unit 14: Feelings and Emotions; Parts of the Body (2)', http://www.loganberry.furtopia.org/bnb/lapine/unit14.html

[Martin] 'The Language Stew', Robert Martin, ACCU 2010 Conference

[NoBugsAxe] 'From Occam's Razor to No Bugs' Axe', 'No Bugs' Bunny, *Overload* #100

[Robinson] *Ariane 5 Flight 501 Failure – A Case Study of Errors*, Ken Robinson, 1996, http://www.cse.unsw.edu.au/~se4921/PDF/ariane5-article.pdf

[Therac-25] http://en.wikipedia.org/wiki/Therac-25

[ToDllOrNotToDll] 'To DLL or Not to DLL', 'No Bugs' Bunny, *Overload* #99, Oct 2010

The Agile 10 Steps Model

Technical processes have tended to dominate agile thinking. Allan Kelly looks at the wider picture.

In the beginning, when Agile first hit the headlines, it was mainly a story about developers doing technical practices. Weird things like pair-programming, writing tests first and other 'extreme' stuff. In time Agile has become a story about processes – iterations, stand-up meetings and such. Extreme went way and it became respectable to 'Scrum'.

Agile has a message for developers and project managers but less so for Business Analysts and Produce Managers. Agile as we know it currently rests on two pillars: technical practices and iterative processes.

Requirements Management form the missing third pillar of Agile. Yet this element hasn't received the same attention as the first two. Teams can achieve some element of Agile with only one or two pillars but for maximum effect and greatest stability all three are required.

These three pillars provide the operations base on which organizations can push to portfolio and strategic Agile. (See Figure 1 and [Kelly10b] for more about Agile at the portfolio and strategy level.)

So far the requirements pillar has one big idea and a whole host of small ideas. The big idea is User Stories. The hinterland includes things like roles and Mike Cohn's INVEST mnemonic (Independent, Negotiable, Valuable, Estimate-able, Small and Testable) but not a lot more.

Then there are the small ideas – small because unlike iterations, TDD or even User Stories, they are not so widely adopted or even well known. Many of these exist in isolation; they don't link up to each other or User Stories.

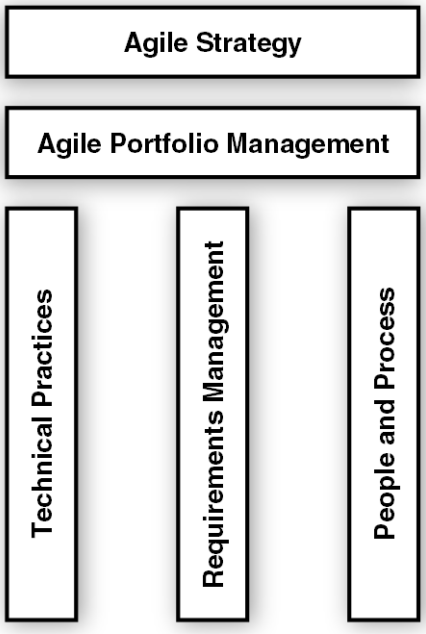


Figure 1

Use of the word 'project'

The PRINCE2 project management guide provides one narrow definition of a project: 'A temporary organisation that is needed to produce a unique and predefined outcome or result at a pre-specified time using predetermined resources.'[Commerce05].

Strictly speaking then a 'project' is a well defined piece of work. Still, it has become commonplace to refer to almost any piece of software development work as a 'project'. For the purposes of this document this wider definition is used liberally in keeping with the industry norms.

That said, I am contributing to an industry problem. Again and again in software development teams I see a 'project' can be anything from a single bug fix lasting a few days up to major change initiatives with a timeline measured in months if not years.

The more I thought about this problem the clearer it seemed to me: these bits weren't joined up. In 2009 I had sketched out a model I call the 'Agile 10 Step', a model I'd like to present here. Explaining the model in depth is beyond the scope of this article – here I will confine myself to a brief overview.

In future articles I hope to elaborate on this model to better link the requirements process up with the rest of the Agile world. Indeed, some of the points raised by the model are already addressed in pieces I have already published.

The model can't position every single requirements technique or tool ever created – that would be asking too much – but it can highlight some of the key ones.

Principles for requirements in an Agile world

The good news is Agile doesn't invalidate whole swathes of requirements engineering the way it can project management. There are many good books on requirements: how to find requirements, understand requirements, capture requirements and so on. The analysis side of this stands up: read a good requirements or business analysis book and most of it is still valid, e.g. [Cadle10], [Alexander09].

In order to discuss requirements more fully it helps to set down some overarching principles:

- **Business value focused**: requirements are a means to an end. The overall objective is to deliver business value. Creating requirements

Allan Kelly has held just about every job in the software world. Today he provides training and coaching to teams and companies in the use of Agile and Lean techniques to develop better software with better processes. He is the author of *Changing Software Development: Learning to become Agile*, numerous journal articles and is currently working on a book of Business Strategy Patterns.
Contact him at http://www.allankelly.net.

With a goal, clear requirements can be discovered by working backwards

is an analysis activity that helps identify and understand business value creation so it can be communicated to development teams.

- **Goal directed projects**: because requirements are emerging, being completed and disappearing, and because the environment is changing, projects cannot be measured on 'scope complete' criteria. Another measure is needed. Instead projects need to measure the progress towards some overarching goal, not fraction complete. (More on goal directed projects in [Kelly10d]).

  With a goal, clear requirements can be discovered by working backwards. Requirements are the things that will move the organization from where it is today towards its goal. Thus it makes sense to start with the desired outcome and work back.
  Of course it is much easier to say 'goal directed' than to realise it. Many projects start with vaguely defined goals. In these cases discovering the goal is a little like panning for gold. In amongst all the ideas circulating some goal needs to be found. Naturally, this makes working backwards to find the requirements even harder.
- **Customer/end user involvement**: those who will actually use the end product need to have a voice in how it is built, and need to have early sight of what is being created. There are two good reasons for this: as the people who perform the work they are best placed to know how things should work and describe the real-life environment to the development team. Second, as the people who will need to use the software their willingness to use the end product is critical. Involving them early and often is the simplest way to do this.
- **Iterative**: in common with the rest of Agile requirements require an iterative approach. One look will not find all the requirements, multiple passes are required and things will change (Previous writing [Kelly08], [Kelly04] contains a discussion of why requirements change.) Thus all aspects of requirements analysis are on going and in parallel with construction.

  Identification, capture and communication starts before the first line of code is cut and continues at least as long as development continues. Modern market economies do not stop while software is created so requirements continue to change and evolve. Priorities and values change.
  Consequently the collection, organization, prioritization needs to be cheap and individual requirements statements disposable. If individual requirements are expensive to create they will take on a life of their own. If they are cheap then there will be less agonising about disposing of them when things change.
- **Just in time**: there is little point in creating and storing masses of requirements in anticipation of the day they will be built. Requirements sitting on the shelf go off – the market and business environment changes. Since requirements are an ongoing process there is simply no need to create a store so we adopt a just-in-time principle instead.
- **Dialogue over document**: communication of requirements is primarily a dialogue rather than an exhaustive document. Documentation can play several useful roles in the requirements process but it should not attempt to be the definitive word on what needs doing.
- **Analysis not synthesis**: the process of deciding what needs doing is primarily one of analysis. The process of building something is primarily synthesis. No amount of analysis will create synthesis, the individuals who are best suited to analysts are usually different to those who are best at synthesis.

  Requirements should not specify what is to be built, or how it is to be built, i.e. solutions, only what is needed to move towards the goal. Of course some requirements specify constraints on the construction rather than functionality. Similarly the individuals who know most about the needs may work with the development team to design a review a proposed solution.

Who manages requirements?

The subject of just who is responsible for managing the requirements is worth an article in itself. One of the main reasons for IT project failure has been lack of user involvement. I'm sure many readers have seen it: a customer asks for a system, some requirements are written and the IT department disappear for six months. When they resurface with the finished system it might bear little resemblance to what was actually wanted – assuming of course that what was actually wanted hasn't changed in the meantime.

Agile's answer to this was to involve the customer, make them central to the development process.

In Extreme Programming the role of the person who specified what was wanted was actually called 'the customer'. Yet while many XP advocates seek to fill this role with an actual customer this is not always possible. Indeed, the original XP case study, the Chrysler C3 project, staffed this role with a Business Analyst.

Scrum calls for the person who really wants the system to get involved and play the Product Owner role. But there are, at least, two problems with this model. Firstly the person who plays the Product Owner may not have the skills and experience necessary to play the role – just because they want the software doesn't mean they know how to work with a development team.

Second, and a common complaint of Scrum teams, is a lack of time from the person playing the Product Owner. If the person in question is important enough to want the software they probably have other things to do. Spending their days working with unwashed developers may not be high on their priority list.

Indeed, the lack of time and consequent stress and pressure was highlighted in a study as long ago as 2004 [Martin04]. Equally worryingly is the focus on a single 'customer' can result in other stakeholder needs being overlooked – a point made by Tom Gilb. Even if customers are put above

the focus on a single 'customer' can result in other stakeholder needs being overlooked

and beyond stakeholders there is still a need to consider multiple customers.

For example, Microsoft Word has several million customers. While these customers may be segmented in various groups (Home, Business, Education, etc.) something needs to be done to understand competing needs, and priorities still need to be decided.

In short, the 'end user' as requirements gatherer and decider model – whether the XP or Scrum version – has problems. What is needed is a requirements professional who can take on these issues and be a proxy for the final customer/users.

Keeping with the Scrum model this article will use the generic Product Manager title to refer to the person(s) who decide what the software needs to do. It is common to find a Business Analyst or Product Manager performing the role, at least on a day to day basis. For more on these roles see my earlier *Overload* articles [Kelly10a], [Kelly09b], [Kelly09a]. Other titles like Requirements Engineer or Analyst are also used for those filling these roles too but the basic skills set remains the same.

One role which is frequently asked to work on requirements but probably should not is that of Project Manager. While some Project Managers move between Business Analysis or Product Manager roles others confine their role to delivery of projects. Sometimes these individuals are asked to take part in the requirements gathering process. The problem is that requirements elicitation is not part of most Project Manager training.

Take for example the UK PRINCE2 standard project manager qualification. PRINCE 2 assume that what is wanted is known, the method and techniques focus on breaking the work down, risk management, scheduling and the like. It does not cover requirements analysis or capture in any depth.

10-Step model overview

The 10 Step Model shown in Figure 2 outlines a framework for aligning the work of requirements engineers – usually a Business Analysts or Product Managers, often called a Product Owner – with Agile development. The model may be considered a process, a checklist or just an aide-memoire. The model attempts to relate various aspects of Agile requirements analysis advocated by different authors.

The model assumes the classic Agile/Scrum/XP iteration, or sprint, time boxed development episodes together with the product backlog / sprint backlog mechanism defined in Scrum [Schwaber02] and XP [Beck00], [Beck04]. These mechanisms can be seen in the lower right of the diagram. Since much has been written about this cycle already this description will focus on the wider requirements process.

1. **Objective**: the objective is given from outside the model – usually from higher up the management chain. It is the reason the team is brought into being, the reason the project is started, the goal the work is aimed towards. (See [Kelly10d] for a longer discussion of this.)
2. **Stakeholders**: stakeholders are those people, and groups of people, who have some interest in the work being undertaken. Stakeholders have their own objectives for the work which might, or might not, align with the objective. Some stakeholders have more stake than others, and some are more significant than others.

This step is not confined to stakeholder identification, it also includes analysis of stakeholder 'stake': what stakeholders want from the system, the constraints they impose, how it will create value for them, and more.

The stakeholders group includes more than just customers. To start with stakeholders can be split into two large groups: internal stakeholders and external stakeholders. Within corporate IT departments the former will be the larger group while in software companies the latter.

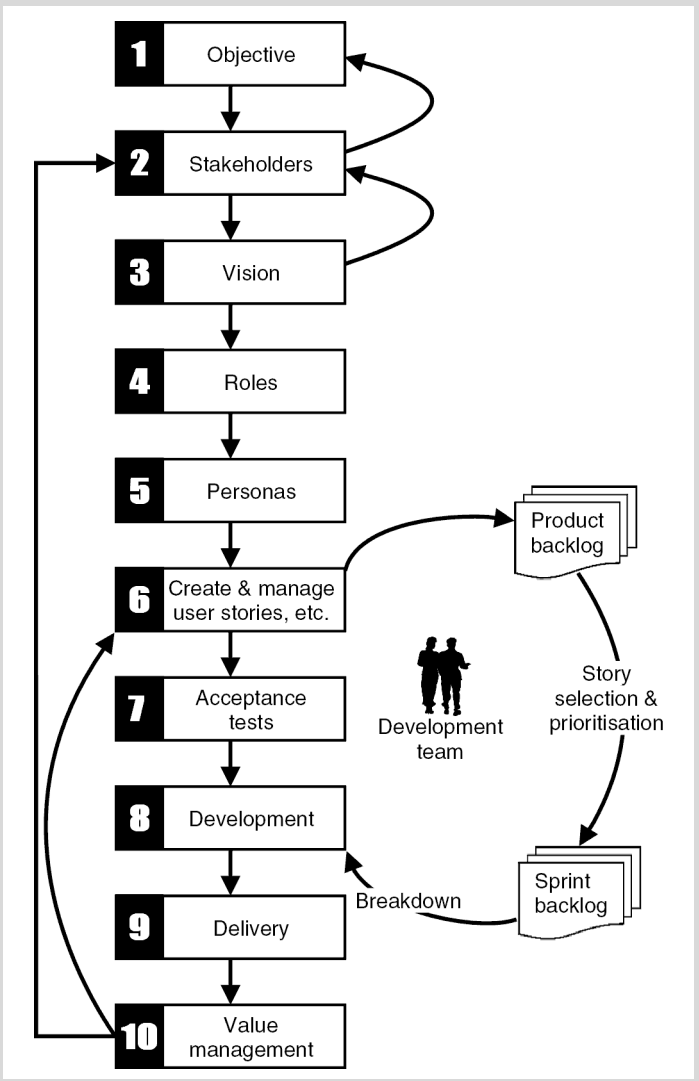


Figure 2

once a need is identified, understood and acceptance criteria specified, it is time to actually do the work and develop the software

Within the external stakeholders group will be the ultimate customer of the organization. More often than not this group will also benefit from segmentation into specific sub-groups.

3. **Vision**: while the objective is owned by the powers that created the team, the vision is created and owned by the team itself. The vision both expands on the objective and answers the objective. If the objective specifies a problem that needs solving the vision gives an answer.
4. **Roles**: roles narrow the stakeholder base to consider those who will actually interact with the system as envisaged by the vision. It is role holders who interact with the system and thus their needs that need to be considered when determining functionality.
5. **Personas**: personas expand and elaborate certain roles, adding texture so requirements analysts, user design specialists and software developers can better understand and empathise towards those who will use the system. Not all roles will be developed into full personas, and different personas will come to the fore at different times.

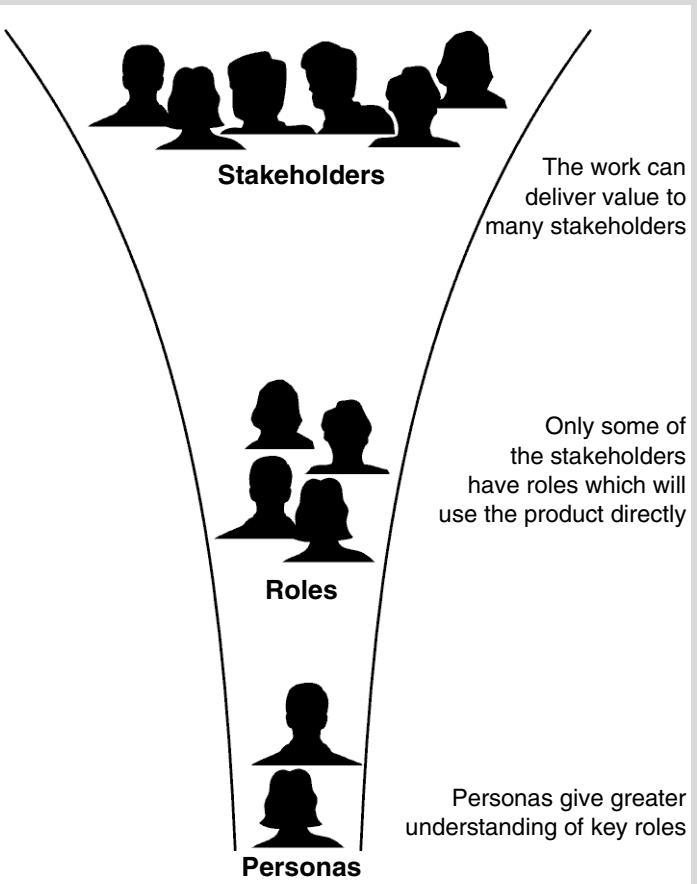


Figure 3

What's the Story?

The basic unit of requirements specification and thus development work, is termed a Story. The format and style of the story can vary widely. Many teams like to use the User Story format: 'As a [Role] I can [Action] So that [Reason]'. This format is commonly associated with Mike Cohn, although Cohn himself credits Rachel Davies [Cohn04], who in turn credits the Connextra development team collectively.

I like to widen this format to allow for Personas and Stakeholders: 'As a [Role|Stakeholder|Persona] I can [Action] So that [Reason]'. Without Stakeholders some User Stories become tortuous as the writer attempts to give a reason to a role. Personas help bring focus to story and add more background texture.

Although widely taken to be part of Scrum this format is absent from the original Scrum texts [Beedle98], [Schwaber03], [Schwaber02]. Nor are User Stories present in Beck's *Extreme Programming* [Beck00], [Beck04]. Beck discusses the idea of a 'development story' without specifying how it is written.

While User Stories are perhaps the most widely used format some teams still prefer to use Use Cases [Cockburn01] or make no attempt to follow a particular format or style. Still other teams use Planguage [Gilb05]. While particularly useful for non-functional requirements Planguage is not widely known and requires a particular skill to use effectively.

For the purposes of this discussion the term story will be used generically to cover all possible formats. Story is taken simply to mean: a small piece of development work to be undertaken.

As analysis proceeds from stakeholders through roles to personas there is a natural narrowing shown in Figure 3

6. **Create and manage stories**: when objectives and users are well understood it is time to start specifying what they system will do. Whatever the format used to describe the specifications something needs to be created. Once more than a few requirements have been captured there becomes a need to manage what has been created. This is the step into which much of the existing Agile literature fits: writing User Stories, Managing the Product Backlog and so on. If the 10-step model is being used as a process these process occur in tandem.
7. **Acceptance tests**: once the essence of a story is captured some description of what constitutes 'done' for the story needs be given. How will developers know to stop writing code? Testers know when to pass, or fail, functionality? And requirements specialists know something has actually been done? The answer to all these questions is a set of criteria that determines when a story is complete.
8. **Development**: once a need is identified, understood and acceptance criteria specified, it is time to actually do the work and develop the software. (Little needs to be said about this particular step because much has already been written about how development happens in Agile teams.)
9. **Delivery**: once a need is met the product needs to be delivered to the customer. For some systems this is a trivial step, for others it is complicated and involved. Delivering a system in multiple discrete

steps is very different from delivering a big-bang all or nothing. Delivering a system as a shrink-wrapped installable software on a CD is different to a software-as-a-service model.

10. **Value Management**: last but by no means least is the need to close the loop and check that value is actually delivered. The key here is linking the finished product back to stakeholders' needs and objectives. Few organizations can place a dollar amount on a single piece of functionality, for some it may be impossible; but since all requirements start with some stakeholder it should be possible to link return to the stakeholder and check whether the thing that is delivered creates value.

From stakeholders to value management

At first site it may seem odd for value management to appear at the end but this step is about closing the loop, ensuring value was delivered not just promised. There is a symmetry between the stakeholder and value management steps. Stakeholders are ultimately the root of all requirements, no matter how technical. At the end of the day someone, somewhere, must want something from the system. For this person, the stakeholder, there is value (perhaps not financial) to having this thing done.

Value can only be assessed if the stakeholders are known. If nobody wants anything doing to a system then nothing should be done. If value cannot be assigned to work then there is no reason to incur the cost.

The stakeholder might not know what work they want doing, and they are often oblivious to the technical aspects, but then, there is no reason why they should. The route between stakeholder and change may be complex and non-obvious but it must exist.

Stakeholder analysis and value management are perhaps the two most important steps and the two which certainly deserve more attention in future.

More tools and techniques

There is certainly no shortage of tools and techniques available to the contemporary business analyst or product manager for analysing needs. Whether it is stakeholder analysts, win-loss reports, business analysts modelling, UML diagrams or CATWOE the tools are available. This model does not try to show where each and every tool may be used: not only would it take too long but there are sometimes no clear answers.

What the model does do is, firstly, place outputs and expectations at the start of the process: objective and stakeholders should provide a way in here. Secondly it shows where these tools can be used: the stakeholders and roles steps are about understanding customers and needs and it is in these stages that most analysis tools come into play.

While some tools will work within single steps in this model other tools will span multiple steps. The truth is, requirements discovery is not a neat and tidy exercise that occurs in clear cut chunks. Like code development it involves intuition, insight and inspiration which cannot be scheduled.

As a result those charged with discovering, understanding and communicating activities are likely to have several different activity streams occurring at once, overlapping and informing one another.

For example, in tandem with this model forward looking plans and scenarios need to be maintained. Release plans and product roadmaps [Kelly10c] are both informed by the information gathered in the model and feed into the model.

Useful?

This is a deliberately brief explanation of the 10 Step model, I hope readers find the model useful and I would appreciate any feedback on the ideas. For me the model has already filled its original intention of helping explain different aspects of Agile requirements.

I certainly find it helps explain and pull together some of the ideas floating around the discussion on Agile Requirements. Although it is simplest to explain as a process I shy away from calling it that. Rather I prefer to think of it as a check-list and a guide

Perhaps it is better still to view this model as a starting point for your own model. Which steps would you remove? Which would you add? Would you reorder any? ■

References

[Alexander09] Alexander, I. & Beus-Dukic, L. 2009. *Discovering Requirements*, Chichester, John Wiley & Sons.

[Beck00] Beck, K. 2000. *Extreme Programming Explained*, Addison-Wesley.

[Beck04] Beck, K. & Andres, C. 2004. *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

[Beedle98] Beedle, M., Devos, M., Sharon, Y., Schwaber, K. & Sutherland, J. 1998. *Scrum: A Pattern Language for Hyperproductive Software Development*. Pattern Languages of Program Design 'PLoP' Allerton Park Monticello, Illinois.

[Cadle10] Cadle, J., Paul, D. & Turner, P. 2010. *Business Analysis Techniques: 72 Essential Tools for Success*, Swansea, BISL (BCS books).

[Cockburn01] Cockburn, A. 2001. *Writing Effective Use Cases*, Addison-Wesley.

[Cohn04] Cohn, M. 2004. User Stories Applied, Addison-Wesley.

[Commerce05] Commerce, O. O. G. 2005. *Managing Successful Projects with PRINCE2*, London, TSO (The Stationary Office).

[Gilb05] Gilb, T. 2005. *Competitive Engineering*, Butterworth-Heinemann.

[Kelly04] Kelly, A. 2004. 'Why do requirements change?' ACCU *Overload*.

[Kelly08] Kelly, A. 2008. *Changing Software Development: Learning to Become Agile*, John Wiley & Sons.

[Kelly09a] Kelly, A. 2009. 'On Management #5 – The Product Manager.' ACCU *Overload*.

[Kelly09b] Kelly, A. 2009. 'On Management #6 – The BA role'. ACCU *Overload*.

[Kelly10a] Kelly, A. 2010. '"I'm a BA get me out of here" – the role of the Business Analyst on an Agile team'. ACCU *Overload*.

[Kelly10b] Kelly, A. 2010. 'Objective Agility' [Online]. *Modern Analyst*. Available: http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/ArticleView/articleId/1502/Objective-Agility-what-does-it-take-to-be-an-Agile-company.aspx [Accessed December 2010].

[Kelly10c] Kelly, A. 2010. 'Three Plans for Agile' [Online]. Toronto: RWNG. Available: http://www.requirementsnetwork.com/node/2663 [Accessed December 2010].

[Kelly10d] Kelly, A. 2010. 'Time for Goal Driven Projects' [Online]. Toronto: RQNG. Available: http://www.requirementsnetwork.com/node/2597 [Accessed 23 December 2010].

[Martin04] Martin, A., Biddle, R. & Noble, J. Year. 'The XP Customer Role in Practice: Three Studies'. In: *Agile Development Conference, 2004* 2004 Salt Lake City, Utah.

[Schwaber02] Schwaber, K. & Beedle, M. 2002. *Agile Software Development with Scrum*, Addison-Wesley.

[Schwaber03] Schwaber, K. 2003. *Agile Project Management with Scrum*, Microsoft Press.

Rise of the Machines

Sometimes the world really is out to get you. Kevlin Henney identifies some culprits.

In the space of a few syllables, the word *resistentialism* is packed with humour, rhythm, profound insight, philosophy, multilingual wordplay and astute commentary on much irksome code. Formed from *res* (the Latin for *thing*), *resistance* and *existentialism*, what exactly does it mean? And what does it have to do with code?

- 'Resistentialism is a jocular theory in which inanimate objects display hostile desires towards human beings.' [Wikipedia]
- 'The theory that inanimate objects demonstrate hostile behavior toward us.' [AWordADay]
- 'The belief that inanimate objects have a natural antipathy toward human beings, and therefore it is not people who control things, but things which increasingly control people.' [WordSpy]

In a nutshell, it's about objects. Especially the ones that seem to thwart our efforts and intentions:

- The stateful and pervasive *singleton* object that increases the stealth coupling of a system.

  Complicated unit tests, reduced thread safety and tsunami rebuilds are just a few of the symptoms. Although undoubtedly introduced with the best of intentions – along with the multitude of other singletons in your application clamouring for your attention, initialisation and coordination – most singletons are short cuts and band-aids (and band-aids for short cuts, and band-aids for band-aids for...) that nurture a sense of resistentialism.

- Uncohesive objects suffering from scope creep and overeducation.

  They seem to know too much and do too much, greedily pulling in as many responsibilities and dependencies as they can. For example, value objects that also have service-like behaviour accessing an external such as the file system, a database or the network. Or classes that amass *statics* as well as instance-specific members, so there is an implied and missing object responsibility, such as an aggregating concept that collects or manages instances of the class. Sleepwalking into a state of resistentialism, one feature at a time.

- Anaemic objects, strangely free of any useful behaviour: constructors that result in meaningless and yet-not-usable objects; methods that fail to encapsulate common usage or reflect the actions and queries of the domain; a type vocabulary that barely drags its knuckles above strings, integers and collections. What you get instead is a public method interface that is little more than a pass-through for the private fields – for every field a `get`, for every `get` a `set`. You cannot understand the system by looking at these objects – they offer little more than object-oriented assembler. Although trivial (even pointless) to test, they are incomplete and free of meaning. Their behaviour is displaced and scattered around the codebase, often holed up in large procedural or controller classes, too difficult to test or to change with any confidence or comfort. Resistentialism is the proclaimed and upheld belief system of the code and those who work on it.

Perhaps the best thing about *resistentialism* is that you now have a blame-free philosophical framework whose name you can call on in times of code distress. And in those times of need you also have a word you can use in polite company. Of course, in a well-crafted system, resistentialism is futile. ■

References

[AWordADay] *A Word a Day*
http://wordsmith.org/words/resistentialism.html
[Wikipedia] http://en.wikipedia.org/wiki/Resistentialism
[WordSpy] *Word Spy*
http://www.wordspy.com/words/resistentialism.asp

Kevlin Henney is an independent consultant and trainer based in Bristol. His development interestests are in patterns, programming, practice and process. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*.