overload 85

JUNE 2008 £3

The Model Student

A continuing look at knotty problems. Can we cut through the complexity?

RSA Made Simple

An introduction to public key cryptography in Java

Quality Manifesto

Is software quality a systems engineering job?

Performitis

Patterns try to solve problems. We look at one from the medical perspective.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 85

June 2008

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 86 should be submitted to the editor by 1st July 2008 and for Overload 87 by 1st September 2008.

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

Plus ça change

Our job titles seem to be dictated as much by fashion as by anything else. Does it matter? It does to some. Oh, and Overload has a new editor.

Hello, and welcome to Overload 85.

Here I am, a newish face at the helm, and this time not as a one-off guest editor. Alan is bowing out after quite a few years, leaving Overload in excellent shape. I'd like to thank Alan for all his work – being editor is one of those jobs where people don't really know exactly what's involved, so it's not given the recognition deserved. But after talking to many people at the Conference about my taking over, it was clear just how much people enjoy the journals and rate them highly, even more so in a world where many other computing magazines have stopped publishing. He can be rightly proud of what he has achieved.

What's in a name?

Talking of jobs, I had an embarrassing moment on accu-general the other week. A couple of people had been mailed by an agency about a contracting job, but part of the description amused one so much that they posted it for discussion. It read:

> This is not a Web Application Developer role, nor a Programming role. This is a position for a Software Engineer.

What a strange turn of phrase! Even more intriguingly, it was in my local area, so I asked those who'd received it who on earth it was for. The embarrassing part turned out that this was not just about a job at my current employers, it was for a contractor doing something pretty much identical to my job. Looks like I'm not a programmer after all.

The phrase had actually come about because our recruiter was trying to explain to an agency what sort of role it was, and was having trouble getting them to understand that it wasn't a web 2.0 application or whatever – the above was the final frustrated attempt after they still hadn't grasped it. Unexpectedly the agency just wrote it down and used it verbatim in their mail shots.

But it did get me thinking about how we describe our jobs, in the software industry in particular, and how it has changed over the years. For example, in my first job, I was a plain old 'Programmer'. This was fair enough as it was the main part of the job, but it wasn't the only thing I did. There was also talking to customers to gather requirements, architectural decisions, software design, programming, build systems, testing, on-site installation, graphic design, and digitising maps (at which I freely admit that I was rubbish). This wide range was helped by it being a small firm, so everyone just mucked in to do what was needed – in a larger place you'd tend to have different people dedicated full time to these different tasks.

Since then I've been a Software Developer, a Software Engineer, a Risk Developer (which sounds excitingly edgy, but in fact referred to the sort of software – Financial Risk Models – that was very much *not* exciting until the sub-prime crisis happened), and currently back to a Software Developer. Other job names I've seen include things like Architect, Analyst, Field Application Engineer, Release Engineer, Build Manager, Scientist, Consultant, Web Developer, Technologist, and even Evangelist. Then these root names often get tagged with some sort of grade or title, and I've seen ones like Graduate, Junior, Senior, Principal, Lead etc. and sometimes a tag to describe the product or field, such as my Risk Developer, or Materials Scientist.

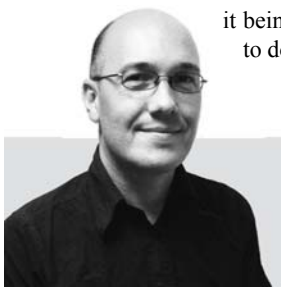
So what are these names trying to describe?

Well, those Junior/Senior type labels are trying to indicate what level of experience, and more importantly, responsibility, people have. One thing to be careful of is that they can't refer to age or years of experience – that's now illegal. Instead I think of them as describing what sort of tasks, decisions, and responsibility the role demands. They can sometimes be a rough label, or come from a more detailed and formal grading system where these labels describe a certain level.

Some try to describe the product, science, or technology area you're working in, hence things like 'Risk Developer', or 'Materials Scientist'. These tend to occur when the area is rather unusual and a particular expertise is needed. Other names could describe a particular area of responsibility, for example 'Build Engineer', but this only works when you're concentrating on just that one niche, whereas most jobs cover a range of technology and roles.

Fashion plays a part too, evidenced by the changes from 'Programmer' to 'Developer' to 'Engineer' and back to 'Developer'. I think I'm happiest with the term Developer – it's what I do, it doesn't describe the details of how I do it, or limit the tasks to just programming, and it doesn't try to elevate it to precision engineering, or worse, a science.

This theme crops up periodically on accu-general – is what we do 'Engineering', or is it a 'Craft', or even 'Art'. Personally, I tend to favour thinking of it as a 'Craft' with parts of engineering involved, based on some theoretical underpinnings that could be described as a science. I justify this view partly by comparing what we do to how people design and build buildings – you need structural engineers to make sure your foundations are good enough and so on, but the people who actually put things up, make the tweaks so that things fit, and the people who adjust the building over the years of use are not engineers – they are much closer



Ric Parkin has been programming professionally for nearly 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 and is now organising the ACCU Cambridge local meetings. He can be contacted at ric.parkin@gmail.com.

to the artisan craftsmen, with years of practical experience of getting things done, and enough knowledge of when to go back and check with the engineers. Craft doesn't imply 'slapped together' though – the best craftsmen build with great skill and elegance, making things that cope well with the inevitable adjustments and changes to requirements. They just accept that not everything has to be perfectly engineered, and in fact would argue that having it too perfect leads to over-optimized brittleness – things change and what was once just right is no longer adequate.

So how good are these names? Well it depends on what you are trying to describe, and who you are describing it to.

We started with a job spec. This is trying to describe two different things – it comes from what the company wants the employee to do; but also it's trying to describe the sort of people who might be suitable for the role, which is much broader and seems surprisingly hard to describe in practice. After all a job advert should be getting people to think 'I could do that', and 'That sounds interesting'. This is going to be a combination of selling the product or company, and roughly describing the role so that the reader thinks they're suitable – but beware of too detailed a description as you'll put off people because they no longer think that they fit.

Then there's the job title once you're in it. This is mainly so that other people know what your expertise and responsibilities are, and for HR to grade you so you get paid enough (although they'll most likely do that with a much more detailed scheme of which your job title is a reflection). As such I don't think the details are actually that important, although some people seem to take it very seriously.

And then there's on your CV, when you're trying to sell yourself to a prospective employer who is trying to match you to a role. Here's where your job title has some importance as you want to sound responsible and multi-talented, but not too specific which would make it sound like you'd only be able to do exactly the same again. And surely worse are those weird job titles where you haven't a clue what they did. You can go into details of the job in the rest of the CV, but the job title is the headline to get the reader's attention.

So how does the original job description phrase fare against these criteria? Well, it isn't developing web applications, so it would have been bad to describe it so incorrectly. A programming role? That's certainly a part of it, but not exclusively so this would be too limited a description. A 'Software Engineer'? Although I don't think software is really an engineering discipline – I would prefer 'Software Developer' (which is in fact the real job title) – this terminology is quite common for this type of wider role, and it works quite well describing it for advertising, HR, and CV purposes.

Not quite so embarrassing after all.

Back in the real world

But there's more to life than your job, and things have been happening in the outside world.

The OOXML standard finally got approved and renamed OXML and is now officially ISO/IEC 29500, which has caused some controversy in certain quarters. There's lots of comment online so I won't add to it, but did notice that one issue seemed to be the voting-in of lots of last minute changes to a very large document (one report mentioned thousands of changes to a document that is several thousand pages long), which has the odd result that the de facto file format that it was meant to standardize is now incompatible without lots of changes! See [Brown] for one such test. With something that big and important, I have to wonder whether fast-tracking was the best way of getting a good quality standard.

This is in stark contrast to the other big standardization process that will be of interest to many of you – the new version of C++ [WG21]. This has been proceeding steadily for the last few years and is now on the final stretch. There's a draft version of the new standard [N2588], which is now around 1254 pages – up from around 800 in C++98 reflecting the extra features. Searching the web will find various compilers that have some of the new features implemented eg ConceptGCC [Concept] and have been used for 'road testing' various ideas. My understanding of the timetable is that the standard is pretty much 'feature complete', then there will be revisions and 'bug fixes' for the following 18 months, and it is due to be finally approved around December 2009. I'm pleased to say that several ACCU members are heavily involved in this process and have been giving talks at conferences and local meetings to update people on what's in it. There seems to be a lot of interest, and I fully expect Overload will be covering many of these new features in depth in the run up to standardization. Watch this space.

References

[Brown] http://www.griffinbrown.co.uk/blog/default.aspx#a3e2202cd-59a3-4356-8f30-b8eb79735e1a

[WG21] http://www.open-std.org/jtc1/sc22/wg21/

[N2588] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

[Concept] http://www.generic-programming.org/software/ConceptGCC/

Postscript

Just before publication I checked my job title: my business card says 'Senior Development Engineer'...

Performitis

Patterns try to solve problems. Klaus Marquardt looks at one from a medical perspective.

Patterns for software systems are a success story. They provide insights that have been gained over years and decades, and convey them in a conveniently accessible way. Programmers and software architects can use them as instant solutions.

A communication view of patterns

Patterns are not just solutions; they are a key mechanism in passing on the software discipline's heritage. Looking at how knowledge and experience is passed on elsewhere, we can see a spectrum: from highly formal languages in mathematics, through examples in arts and architecture, to some cultural and ethical insight hidden in stories and fairy tales. Patterns about software design and architecture are certainly more on the formal side of this spectrum.

Patterns embrace the value of examples to give credibility to their message. However, story telling is virtually irrelevant to patterns. Not surprisingly, it is not irrelevant to the software people themselves. When a group of software people gathers in a canteen or a bar, patterns play, if anything, a minor part in the conversation. Stories about projects are dominant, often told without references and from merely indirect experiences. Some resemble more the characteristics of urban legends rather than sound research. Still, we love to hear these stories and we learn from them.

So what is missing from patterns that can be found in our traditions and collective knowledge?

The project stories hardly mention fundamental technical problems – in contrast with patterns, which are usually about technical problems and their solutions. (While there are numerous exceptions to this, these happen to be less popular and well known.) This observation triggers two thoughts. First, if projects fail they appear to fail for non-technical reasons only. Second, a technical problem also has a project context. This context offers many non-technical reasons for the technical solution chosen by the developers.

Beyond design: diagnosis patterns

Team culture, organization structure, and software solutions mutually influence each other. Conway's Law [Conway68] is probably the best known coupling force, but there are other less well known relationships, and our stories and anecdotes are attempts to bring some light to them.

Again, a look at other disciplines brings further insights. Medicine is as old as mankind. Medical doctors have learned that beyond injuries and diseases, their patients' sociological situation influences their health. Depending on the individual, scientific western, traditional eastern, or alternative medicine may be most effective. Pragmatists have come to accept that whatever cures is right. And even more pragmatic, a patient need not be cured to feel much better and gain perceived (and possibly objective) health.

We could probably make use of the approaches medical doctors use to help their patients. They have found ways to identify many influences on a disease, and in turn know many different treatments that may address symptoms, or causes, or just help for whatever reason.

This resembles an important aspect of what senior developers, architects, and consultants do. Facing an unfamiliar situation or a project in trouble, which is the rule rather than the exception, their task is problem solving rather than developing a master plan. They observe on many different levels, and they start acting before everything is known.

The idea behind the pattern form of diagnoses and therapies is that these diagnoses collect knowledge from the non-technical regime and combine therapy options from different angles. This is not a contradiction to design and architecture patterns, but a complement acknowledging the actual needs in project dynamics. Finally, diagnoses make use of stories from real or fictitious projects to trigger memories and thoughts.

The doctor will see you now

As an example, the rest of this article describes a disease found in some software systems. PERFORMITIS arises from an overly narrow focus on performance during development. While it appears a technical issue at first, closer examination shows that its foundations are people and process issues. Accordingly, PERFORMITIS should be treated by team and process therapies as well as technical, and a combination of both is typically most successful.

Medical approach

Once we start with some medical terminology, a whole bunch of concepts, vocabulary, and approaches comes to us. The good news is that these concepts have been in use for ages and can safely be considered more mature than patterns. The bad news is that they sound only vaguely familiar. How does a doctor work?

To get to a diagnosis, doctors start with an examination. They watch for symptoms in the broadest sense: physical findings, movements, behaviour, clothing, smell, speech. Unlike many engineers, doctors often start some treatment even without a clear diagnosis. In emergency situations it is essential to keep a patient alive regardless of his injuries. And if you happen to have a fever you'd appreciate paracetamol and a broad-band antibiotic before someone has identified the exact germ.

In case of doubt, and if the consequences would be significant, a differential diagnosis might be required with a more detailed examination. Finally, a number of treatment options are considered and a treatment scheme initiated. As the disease and the cure proceeds, the treatment is adjusted appropriately. Often a therapy is continued while the doctors "proactively wait" and observe.

Klaus Marquardt is a technical manager and system architect with Dräger Medical in Lübeck, Germany. His experience includes life support systems and large international projects. Klaus is particularly interested in the relations between technology, organization, people and process. He has contributed sessions to many conferences including OOP, JAOO, ACCU, SPA and OOPSLA. Klaus can be contacted at marquardt@acm.org patter@kmarquardt.de

All decisions for tuning measures are made individually on a local scale

Diagnosis: PERFORMITIS

Every part of the system is directly influenced by local performance tuning measures. There is either no global performance strategy, or it ignores other qualities of the system such as testability and maintainability.

Tuned Trabant 601, GDR, 19kW [Trab03]

With PERFORMITIS developers use every opportunity to optimize their code, and are eager to discuss different performance measures with their peers. While each single measure may be perfectly justifiable, check whether you observe the majority of the strategies from the following list:

- Usage of C or C++ and its language specific low level features.
- Extensive and early usage of language specific constructs that favour local performance gains over other qualities like encapsulation, reusability or portability. Examples would be public attributes, or inlining compromising the design.
- Avoidance of indirections, at the expense of tighter coupling, limited extensibility and increased effect of individual changes. Examples for indirection range from virtual functions to entire encapsulation layers.
- Keeping control of implementation details by preferring hand written code over advanced language or library features. E.g., extensive usage of pointer arithmetic instead of advanced specific data types (C/C++).
- Avoidance of code libraries that are not controlled within the organization, due to the expectation that its performance will be less optimal than when implemented for the specific project.
- Responsibility of code modules is clustered according to execution threads rather than to consistent logical responsibilities.
- Wide-spread usage of environment specific features, like direct calls to OS, DB triggers and stored procedures, linker directives, or scheduling priorities.
- Application specific code contains knowledge of technical issues and their optimization. Examples would be knowledge of network package size and frequencies, or database structure and joins.

These techniques are applied by more than a small minority of the programmers involved, and all over the source code. Trying to improve qualities that have been neglected due to performance tuning would cause expensive changes to large portions of the code.

The most significant observation is that there is no overall evaluation or strategy in place that determines where and when which of the above measures is taken. All decisions for tuning measures are made individually on a local scale.

The project is staffed with experienced developers who are familiar with the domain and the task at hand. This is not the first project the key team members have worked on, and they have learned some important lessons in their previous projects. The key developers all agree that performance is the single property that is hardest to achieve, and that it could cause a project to fail even when it is presumably almost completed. They exhibit a strong desire to get the performance aspects right first, paying merely lip service to other quality aspects.

When new colleagues join the project, or some contractor gets insight into the system, discussions about the way to ensure performance and other intrinsic qualities will arise. These discussions tend to become emotional quickly because they are at the heart of the individual working style and value system – and they could potentially exhibit deficits in the high level systems design. PERFORMITIS is only resident in projects that are either closed against outside influences, or have developed mechanisms to terminate discussions that raise inconvenient questions. Look out for the closed project society (if you have the chance).

PERFORMITIS infected systems are unstable with respect to technology or requirement changes. Because the structure ignores logical separations, the changes have a large impact and are costly and timely. The effort related to such changes sometimes exceeds the effort required for initial feature development. Look out for fear of changes and explicitly scheduled, often postponed technology updates.

The negative effects of PERFORMITIS are also visible to upper management. For all but the most experienced teams, delivery dates can frequently be missed. Performance oriented development neglects qualities like testability, and creating tests is an expensive endeavour. The project typically provides little tests on code level and integrates late in the development cycle. The spread-out local tuning measures make it difficult to fix the problems that arise with testing and integration while preparing the delivery. It is ironic that the measures taken then may dramatically degrade performance. Look out for more than one seemingly surprising schedule slip immediately before releases, or for an extremely high effort in test and integration of limited functionality.

Not all of these symptoms are unique to PERFORMITIS, and some only become visible in late states of the disease. Early and sufficient signs are the combination of the spread of local optimizations, the lack of a global strategy, and the team attitude.

The pathogen is the limited experience of key developers. Humans tend to remember their failures better than their successes, and many developers have learned particular painful lessons. Mainly in distributed or embedded systems, projects can miserably fail due to performance problems – which none of the participants will ever forget. The limitation in their minds is

A number of different practices are used to increase local performance at the expense of other qualities

Symptoms checklist	
A number of different practices are used to increase local performance at the expense of other qualities	☑
These practices are not limited to a dedicated portion of the code, but spread all over the project	☑
There is no strategy for which local tuning practices are applied where	☑
Performance is considered by far the most significant system quality	☑
There is no objective measure of what performance is required	☑
The project closes itself against external influence	☑
Changes occupy large parts of the schedule and are frequently postponed	☑
Either a lot of effort is spent in tests and integration, or several milestone dates have not been met with very little warning time	☑

Diagnosis: PERFORMITIS[1], also known as PERFORMANCE BLOAT.

Figure 1

like a Pavlovian Reflex that happens especially to developers with extensive experience in a particular domain only. They will not realise that different systems need to balance different qualities.

Prescription

Now that we have a diagnosis, treatment can be applied. Next time we'll introduce some therapeutic patterns and possible application strategies. ■

Acknowledgements

The first and foremost thanks go to Jens Coldewey, the shepherd of the first version of this paper. The workshop participants at EuroPLoP 2003 gave valuable and encouraging feedback: Frank Buschmann, Arno Haase, Kevlin Henney, Wolfgang Herzner, Michael Kircher, Alan O'Callaghan, Kristian Sörensen, Markus Völter, Nicola Vota, and Tim Wellhausen.

Further thanks to many unnamed colleagues who sometimes unintentional contributed to this diagnosis, its therapies and the examples.

References

[Conway68] Conway, M.E. 'How do Committees Invent', *Datamation*, 14 (5): 28-31.

[Trab03] Picture available at http://www.tuning-scene-droyssig.de/601maik1.jpg

1. A term closer to the medical nomenclature would be 'performance related softwareosis'. Any chronic disease is called an ~osis. Furthermore, it is not the performance that is infected but the system itself. However, 'performitis' is a more popular name.

The Model Student: A Knotty Problem, Part 2

Tying yourself in knots is easy. Richard Harris cuts through the complexity.

Last time we took a look at the number of self crossings of a random walk with the intention of shedding some light on why headphone cables and the like behave so malevolently. We developed some code to exhaustively search through all possible walks of a given length, but as is often the case this gets computationally burdensome for large problems. What we really need is to find a formula for the expected number of self crossings which we will hopefully be able to exploit to deduce the likelihood that a walk will contain any given number of crossings, or to use the technical term, the distribution of crossings.

To that end, the first thing we should observe is that it is relatively easy to calculate the probability that a walk of n steps will end at its starting point. It is simply the probability that the number of steps to the left is equal to the number of steps to the right and the number of steps forward is equal to the number of steps back [Feller, 68].

To construct the formula we need to derive an expression for the probability that a walk of n steps will have exactly i steps to the left and right, or equivalently forward and backward.

If we didn't allow a step to leave the walk in the same place, the number of ways we could have i steps to the left, and hence n-i steps to the right, would simply be the number of ways we could choose i from n things, given by the combination formula:

$$^{n}\mathrm{C}_{i} = \frac{n!}{i!(n-i)!}$$

The probability that these numbers of left and right steps occur is determined by the binomial distribution. To calculate it we need only multiply the number of ways it can occur by the probability of each of those ways.

$$P(left-i) = {}^{n}C_{i} p_{left}^{\ i} p_{right}^{\ n-i}$$

Since we have an equal probability for each kind of step, this simplifies to

$$P(left=i) = {}^{n}C_{i}\frac{1}{2^{n}} = \frac{n!}{i!(n-i)!}\times\frac{1}{2^{n}}$$

The probability that we will take i steps to the left, j steps to the right and hence n-(i+j) non moving steps is given by the related trinomial distribution.

$$P(left=i,right=j) = \frac{n!}{i!j!(n-i-j)!}\times\frac{1}{3^{n}}$$

So the probability that we return to the start with exactly i steps to the left and right is given by

$$p_{n,i} = \frac{n!}{i!i!(n-2i)!}\times\frac{1}{3^{n}}$$

Therefore the probability that we return to the starting point after n steps, at least in one dimension, is simply the sum of the probabilities that with return with every possible number of steps to the left and right. These range from zero up to ½n.

$$p_{n} = \sum_{i=0}^{\lfloor\frac{n}{2}\rfloor} p_{n,i}$$

The weird brackets at the top of the summation mean the largest integer less than or equal to ½n, by the way.

To transform this into the probability that we return to the starting point in two dimensions we need only observe that the probabilities in each dimension are independent; the likelihood of our being in the same place horizontally has no bearing on the likelihood of our being in the same place vertically. We can therefore simply multiply the two, identical as it happens, probabilities.

$$p_{n}^{2}\mathrm{v} = \left(\sum_{i=0}^{\lfloor\frac{n}{2}\rfloor} p_{n,i}\right)^{2}$$

The next step in our derivation is to deduce the expected number of times a walk will return to its starting point. This is simply one times the probability that it returns after one step plus one times the probability that it returns after two steps and so on. In other words we simply sum the probabilities from the first to the last step.

$$E_{0,n} = \sum_{i=1}^{n} p_{n}^{2}$$

The final step is to note that at each point in the walk, we can consider ourselves to be at the starting point of a shorter walk. The total expected number of knots should therefore be the sum of the expected number of returns for every walk from one to n-1 steps.

$$E_{n} = \sum_{i=1}^{n} E_{0,j}$$

Unfortunately we're going to run into a few problems calculating this number. The factorial at the heart of the formula is an O(n) operation and thus the entire calculation is O(n^4); a rather daunting task. Furthermore, as was pointed out in the analysis of the travelling salesman problem, the factorial grows very quickly as its argument increases. Very, very quickly; 13! is too large to fit into a 32 bit unsigned integer and at 171! we'll exceed the maximum value of an IEEE 64 bit floating point number.

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.

The former problem can be mitigated with judicious use of caching

We can address the latter problem using logarithms. As you are no doubt aware, the logarithm of a product is equal to the sum of the logarithms of its terms.

$$\ln n! = \sum_{i=1}^{n} \ln i$$

Since the trinomial distribution divides an $n!$ term by three factorial terms and a further $3n$ term, we can be confident that it will not overflow as quickly as the terms $n!$ will. We will be therefore better off working with log factorials and inverting the logarithm at the end. The inverse of the logarithm is the exponential function; we can recover the final value by taking the exponential of its logarithm. Exploiting the remaining properties of logarithms, we have

$$p_{n,i} = e^{\ln n! - 2\ln i! - \ln(n-2i)! - n\ln 3}$$

The former problem can be mitigated with judicious use of caching. Both the log factorial and the probability of return, p_n^2, are calculated multiple times for the same arguments. If we cache the results of p_n^2 we can reduce the complexity of the calculation by two orders of magnitude to $O(n^2)$; much less daunting. Caching the results of the log factorial similarly reduces the cost of calculating p_n^2. This only results in a further constant factor improvement in efficiency, but will probably be useful in future, so we might as well implement it.

When we need a log factorial not already in the cache, we can exploit the fact that we have already done some of the work. Rather than iterate all the way from one to n, we need only work our way up from the greatest value we already have, populating the intervening entries in the cache on the way (Listing 1).

As it stands the cache could grow uncontrollably, severely reducing the efficiency of the function. If we were planning to use this as a general purpose algorithm we should probably address this problem. One approach would be to set a fixed upper limit, after which we return infinity (Listing 2).

The problem with this is that it puts an artificial limit on the maximum factorial we can calculate. Fortunately, there is an approximation we can use instead; a (relatively) modern improvement [Robbins, 55] on a long standing approximation known as Stirling's formula.

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

For values of n greater than a few thousand or so, the relative error in the logarithm of this approximation is of the order of 10^{-12}, which is probably sufficient for our needs (Listing 3).

An exact value can be derived for all n, up to numerical overflow at least, from the Gamma function. This is a fundamental mathematical function that crops up in a variety of situations and is defined by

```
namespace knots
{
  double log_factorial(size_t n);
}

double
knots::log_factorial(size_t n)
{
  static std::vector<double> results(1, 0.0);
  if(n>=results.size())
  {
    size_t i = results.size();
    results.reserve(n+1);
    while(i!=n+1)
    {
      results.push_back(
         results.back() + log(double(i++)));
    }
  }
  return results[n];
}
```

Listing 1

```
double
knots::log_factorial(size_t n)
{
  static const size_t max_n = 2047;
  if(n>max_n)
    return std::numeric_limits<double>::infinity();
    ...
}
```

Listing 2

```
double
knots::log_factorial(size_t n)
{
  static const size_t max_n = 2047;
  static const double pi = 2.0*acos(0.0);
  if(n>max_n)
  {
    return 0.5*log(2.0*pi*double(n)) +
           double(n)*(log(double(n))-1.0) +
           1.0 / (12.0 * double(n));
  }
  ...
}
```

Listing 3

We finally have the scaffolding in place to calculate the expected number of knots

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

Unfortunately it has no closed form solution. It can be approximated numerically, Press et al [Press, 92] provide an implementation, but it's not particularly straightforward so I'm going to stick with Stirling's formula.

The next step is to implement a function to calculate the probability of return, p_n^2 (Listing 4).

Once again, we simply resize the cache if we get a request for a value we have not yet calculated and populate it for all values up to the one we need. Unfortunately, this time there's no approximation we can rely on to restrict the growth of the cache. Since this function isn't really useful outside of this analysis we don't have to take possible future applications into account, so it doesn't present quite so much of a problem as it did for the log factorial calculation.

Most of the work is done in the inner loop; it is simply the calculation of p_n. As discussed we use log factorials to calculate the log of each term and take the exponential at the end in order to avoid overflow. Finally, we square the probability to transform it to two dimensions before storing it in the cache.

We finally have the scaffolding in place to calculate the expected number of knots. The last two functions we need to implement are *E0, n* and *En* and, fortunately, both are relatively simple. They are illustrated in Listing 5 (which is calculating the expected proportional number of knots) and, in keeping with what they represent, we refer to $E_{0,n}$ as **expected_returns_to_start** and E_n as **expected_knots**.

So how does this function compare to the observations we've already made? Not very well as it happens (Figure 1):

```
namespace knots
{
  double p_returns_after(size_t n);
}

double
knots::p_returns_after(size_t n)
{
  static std::vector<double> results(1, 0.0);
  if(n>=results.size())
  {
    size_t i = results.size();
    results.reserve(n+1);
    while(i!=n+1)
    {
      double next = 0.0;
      for(size_t j=0;2*j<=i;++j)
      {
        double term = log_factorial(i)
                    - 2*log_factorial(j)
                    - log_factorial(i-2*j)
                    - double(i)*log(3.0);
        next += exp(term);
      }
      next *= next;
      results.push_back(next);
      ++i;
    }
  }
  return results[n];
}
```

Listing 4

n	calculated E(knots)/n	observed E(knots)/n
6	0.2969	0.2492
7	0.3200	0.2623
8	0.3408	0.2737
9	0.3598	0.2837

Figure 1

```
namespace knots
{
  double expected_returns_to_start(size_t n);
  double expected_knots(size_t n);
}

double
knots::expected_returns_to_start(size_t n)
{
  double result = 0.0;
  for(size_t j=1;j!=n+1;++j)
     result += p_returns_after(j);
  return result;
}

double
knots::expected_knots(size_t n)
{
  double result = 0.0;
  for(size_t i=1;i!=n+1;++i)
  {
    result += expected_returns_to_start(i);
  }
  return result / double(n);
}
```

Listing 5

Congratulations to those of you who've already spotted the error in my reasoning

Congratulations to those of you who've already spotted the error in my reasoning; it took me a while to figure it out. The problem is that when we add up the expected number of returns, we're not taking into account the probability that we've already visited the current position.

To illustrate this point imagine that a random walk has returned to the starting point on the second step. We should not therefore include the expected number of returns to the second point in the calculation; since it is the starting point they have already been accounted for. Unfortunately this over-count occurs whenever we return to a point we have already visited during a given walk. Quite frankly, it's surprising that the approximation isn't much, much worse.

A further irritation is that I can't quite see how to figure it into the calculation. We can make some progress by observing that a random walk looks the same both forwards and backwards. The probability that the *n*th step is into a previously unvisited position is therefore equal to the probability that we never return to the starting point in an *n* step random walk. This can be expressed with a conditional probability; it is the probability that the *n*th step doesn't return to the start given that none of the previous steps have. This naturally recursive property could be used to construct the desired result. Well, it could if I could work out what the value of the conditional probability actually is.

Formally, the conditional probability of an event A occurring given that an event B already has is given by

$$p(A|B) = \frac{p(A \wedge B)}{p(B)}$$

I should point out that the caret like symbol is one of the mathematical representations of 'and'.

If the events A and B are independent then the probability of them both occurring is simply the product of the probabilities of each of them occurring. The conditional probability then simplifies to:

$$p(A|B) = \frac{p(A)p(B)}{p(B)} = p(A)$$

The trouble is, our steps are not independent, and their mutual dependence is too complicated for me to express the conditional probability mathematically.

We can, however, try to approximate the expected number of returns by assuming that the steps are independent, allowing us to simply multiply the probabilities that each remaining step does not return to the current position.

We already have an expression for the probability that a step *does* return. The probability that it doesn't is simply this value subtracted from one. The probability that the *n*th position is unique is therefore

$$p_n^u = \prod_{i=0}^{n-1} (1 - p_{n-1}^2)$$

I'm using the u superscript to distinguish this probability of uniqueness from that of a walk returning to its starting point.

The giant Π is the mathematical notation for the product of a series of terms and is analogous to the use of Σ to represent sums. In fact they were both chosen because they are the Greek versions of the first letters of product and sum, respectively.

The implementation is pretty straightforward as Listing 6 illustrates, which calculates the probability that a step enters an unvisited position.

To avoid double counting the knots from positions we have already visited we need only to multiply the expected number of returns to a given step by the probability that it's unique.

$$E_n = \sum_{i=1}^{n} E_{0,i} \times p_{n-i}^u$$

The change to the implementation is equally straightforward (See Listing 7, which calculates the expected proportional number of knots.)

```
namespace knots
{
  double p_is_unique(size_t n)
}

double
knots::p_is_unique(size_t n)
{
  double result = 1.0;
  for(size_t i=0;i!=n;++i)
     result *= 1.0-p_returns_after(n-i);
  return result;
}
```

Listing 6

```
double
knots::expected_knots(size_t n)
{
  double result = 0.0;
  for(size_t i=1;i!=n+1;++i)
  {
    result += expected_returns_to_start(i)
       *p_is_unique(n-i);
  }
  return result / double(n);
}
```

Listing 7

it's a bit tricky to come up with a satisfying method to randomly generate one of a fixed number of arbitrary states

n	calculated E(knots)/n	observed E(knots)/n
6	0.2476	0.2492
7	0.2598	0.2623
8	0.2701	0.2737
9	0.2790	0.2837

Figure 2

n	calculated E(knots)/n	observed E(knots)/n
10	0.2867	0.2928
20	0.3315	0.3499
50	0.3704	0.4196
100	0.3843	0.4655

Figure 3

Figure 2 documents the impact of this change on the calculated value of the expected number of knots.

As you can see, it's a much better approximation. The question remains as to how well it fits for lengthy walks. Exhaustively enumerating long walks is out of the question, so we're going to have to take random samples.

We'll need an equivalent to **`std::random_shuffle`** with which we can generate them. Ideally, it should be as general as our **`next_state`** function. The trouble is that it's a bit tricky to come up with a satisfying method to randomly generate one of a fixed number of arbitrary states. The best I can think of is to use a sequential container of states and randomly generate indices into it. Admittedly, it's not very natural for integer types, but at least we can use it with any type we want. (Listing 8: Generating random states.)

Listing 9 illustrates the code to take a random sample of a walk of length *n*.

So how does the new formula rate for long walks? Figure 3 shows the results for one million samples from walks of various lengths.

```
double
rnd(double x)
{
  return x * double(rand())/
    (double(RAND_MAX)+1.0);
}


template<class BidIt, class States>
void
random_state(BidIt first, BidIt last,
   const States &states)
{
  while(first!=last)
  {
    *first++ = states[size_t(
       rnd(double(states.size())))];
  }
}
```

Listing 8

Clearly, the assumption of independence is inappropriate; the calculated number of knots diverges from the observed number pretty quickly. Unfortunately, this does not bode well for the distribution I had suspected.

That distribution is the Poisson distribution. This is the distribution of the expected number of occurrences of events with exponentially distributed waiting times. Now this may not sound as general as the normal distribution since that's the limit distribution of sums of independent random numbers drawn from almost any given distribution. It would be a mistake to dismiss it too quickly though.

If the average time we need to wait before an event occurs is independent of the amount of time we have already been waiting, then the waiting times must be exponentially distributed. And these kinds of events show up a

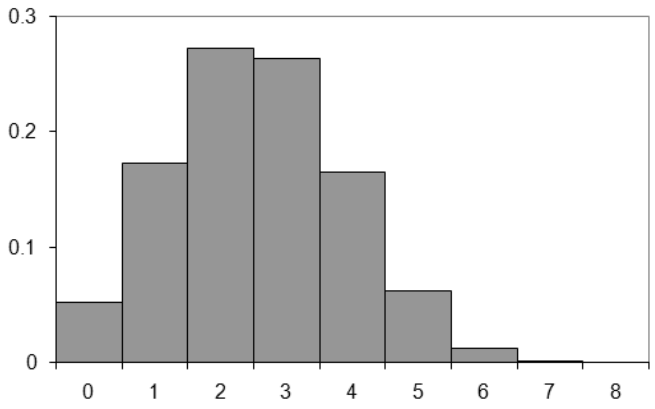
```
namespace knots
{
  void sample_crossings(
   knot_histogram &h, size_t samples);
}

void
knots::sample_crossings(knot_histogram &h,
                        size_t samples)
{
  if(h.walk_length())
  {
    std::vector<size_t> states(9);
    for(size_t i=0;i!=states.size();++i)
       states[i] = i;

    walk w(h.walk_length());

    while(samples--)
    {
      random_state(w.begin(), w.end(), states);
      h.add(crossings(w));
    }
  }
}
```

Listing 9



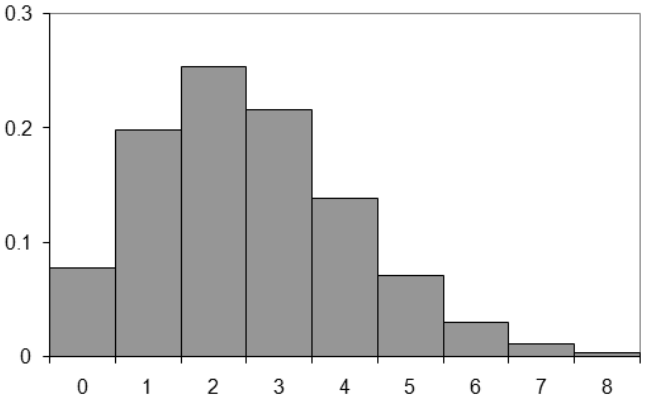
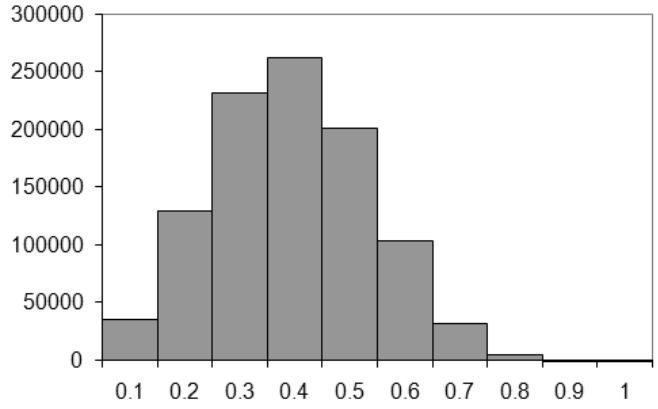
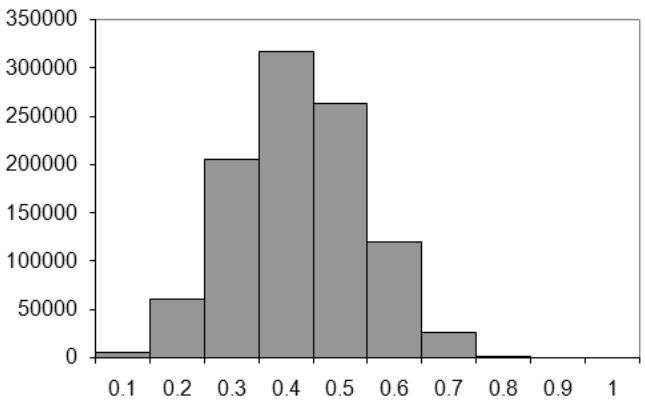
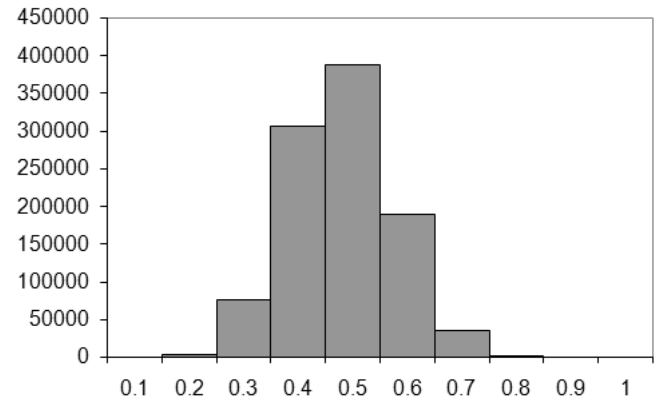


Figure 4







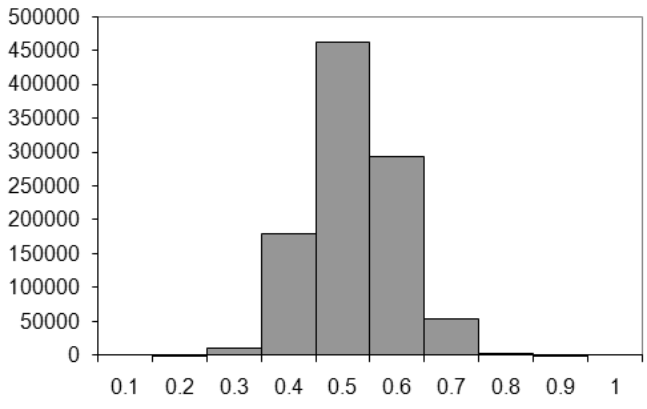


Figure 5

this universe favours a high proportion of self crossings and hence, presumably, knots

lot; consider how long we should wait before rolling a six on a fair die. Of course this is a discrete, rather than continuous, time example but should give you an idea of how common these types of events really are.

The Poisson distribution gives the probability that r events will occur given that the expected number occurrences is λ and is defined by the formula

$$P(X = r) = e^{-\lambda}\frac{\lambda^r}{r!}$$

So how does it compare to the observed results? Figure 4 compares the observed and approximate histograms for a walk of nine steps.

Well, they're sort of similar I suppose. Not very much so, though. It seems that I was right to be concerned about the inaccuracy of the independence assumption.

Does the situation improve for these longer walks? I suspect not given that the approximation of the expected number of knots seems to get less accurate for longer walks. Still, it's a simple matter to check, so let's take a look at some longer walks.

The sample distributions from which we derived the expected number of knots above are shown in Figure 5. Top left shows walks of 10 steps; top right of 20 steps; bottom left of 50 steps and bottom right of 100 steps.

Figure 6 compares the difference between the observed and Poisson approximation for the one hundred step walk.

As expected, the approximation is not really any better for lengthy walks.

So, we shall have to rely upon the sample data. They certainly seem to indicate that this universe favours a high proportion of self crossings and hence, presumably, knots. The question that remains is whether or not the universe is out to get us. Would we be liberated from the foul machinations of stringy things if we lived somewhere else?

```
namespace knots
{
  class position
  {
  public:
    position();
    position(long x, long y, long z);
    position move(long dx,long dy,long dz) const;
    bool   operator<(const position &rhs) const;
    bool   operator==(const position &rhs) const;

  private:
    long x_;
    long y_;
    long z_;
  };
}
```

Listing 10

Well, to answer that let's take a look at what life might be like if we lived in a universe with four spatial dimensions. Firstly, we'll need to change the `position` class to represent a point in three dimensional space (Listing 10).

The changes to the member functions are fairly obvious, as Listing 11 illustrates.

Next we need to update the calculation of the number of crossings in a given walk. Again, it's reasonably straightforward (Listing 12).

The main point here is that we now have twenty seven valid states rather than nine. The extraction of the steps in each direction follows the approach used for the two dimensional case, with a little additional complexity to cope with the third dimension.

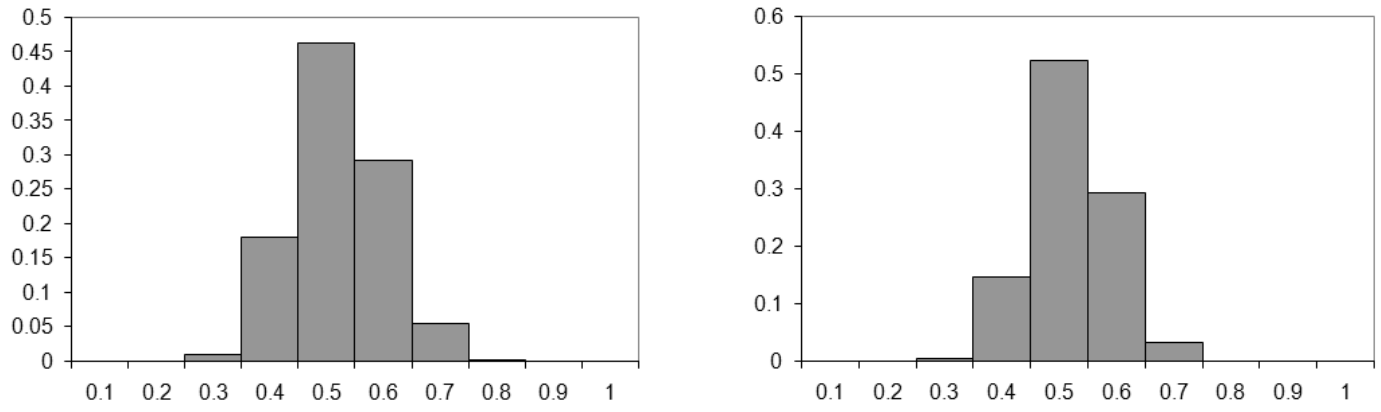


Figure 6

the universe really is out to get us, since it seems to have left our four dimensional neighbours relatively unmolested

```
knots::position
knots::position::move(long dx,
                      long dy, long dz) const
{
  return position(x_+dx, y_+dy, z_+dz);
}

bool
knots::operator<(const position &rhs) const
{
  return x_<rhs.x_ ||
         (x_==rhs.x_ && y_<rhs.y_) ||
         (x_==rhs.x_ && y_==rhs.y_ && z_<rhs.z_);
}
```

Listing 11

The one million sample histograms for walks of ten, twenty, fifty and one hundred steps are illustrated in Figure 7 (knot histograms for three dimensional walks of length 10, 20, 50 and 100).

The difference between the two and three dimensional walks is fairly dramatic, I'm sure you'll agree. In the two dimensional case the most probable number of knots seems to increase as a proportion of the number of steps as the walk lengthens. In the three dimensional case it seems to tend to 0.1 to 0.2 times the length of the walk.

Does this trend continue? Let's compare the results for a really long walk, say one thousand steps.

I think the evidence speaks for itself; the universe really is out to get us, since it seems to have left our four dimensional neighbours relatively unmolested.

Spared from the burden of ever having to untangle the power cables for their diabolical death rays, any invading horde from the fourth dimension is well set to spring a surprise attack upon us at a moment's notice. We can only be thankful that they haven't yet done so.

I honestly believe that we are not adequately prepared for such a contingency.

Given the enormous difficulty they'll surely have tying their shoelaces, I suspect that a great many of their dread number will have to attack barefoot. I therefore suggest that we should petition the government to arm every household with an ample supply of brass tacks. If we are sufficiently alert, we should be able to stop them dead in their tracks.

In the meantime, we should continue our research. I am out of ideas, but if you, dear reader, can discover anything further please write in. The future of humanity may depend upon it. ■

```
size_t
knots::crossings(const walk &w)
{
  size_t knots = 0;
  position p(0, 0, 0);
  std::set<position> visited;

  walk::const_iterator first = w.begin();
  walk::const_iterator last  = w.end();

  visited.insert(p);

  while(first!=last)
  {
    if(*first>26) throw std::invalid_argument("");

    long dx = long(*first)%3     - 1;
    long dy = (long(*first)/3)%3 - 1;
    long dz = long(*first)/9     - 1;

    p = p.move(dx, dy, dz);
    if(!visited.insert(p).second) ++knots;
    ++first;
  }

  return knots;
}
```

Listing 12

Acknowledgements

With thanks to Andrey Biryuk and Chris Morris for proof reading this article.

References & Further Reading

Bachelier, L., *Théorie de la Spéculation*, PhD thesis, 1900.

Brown, R., 'A brief account of microscopical observations made in the months of June, July and August, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies.' *Edinburgh New Philosophical Journal*, July-September, pp. 358-371, 1828.

Feller, W., *An Introduction to Probability Theory and its Applications*, vol. 1, 3rd ed., Wiley, 1968.

Hayes, B., 'Square Knots', *American Scientist*, vol. 85, num. 6, pp. 506-510, 1997

Hull, J., *Options, Futures and Other Derivatives*, 6th ed., Prentice Hall, 2005.

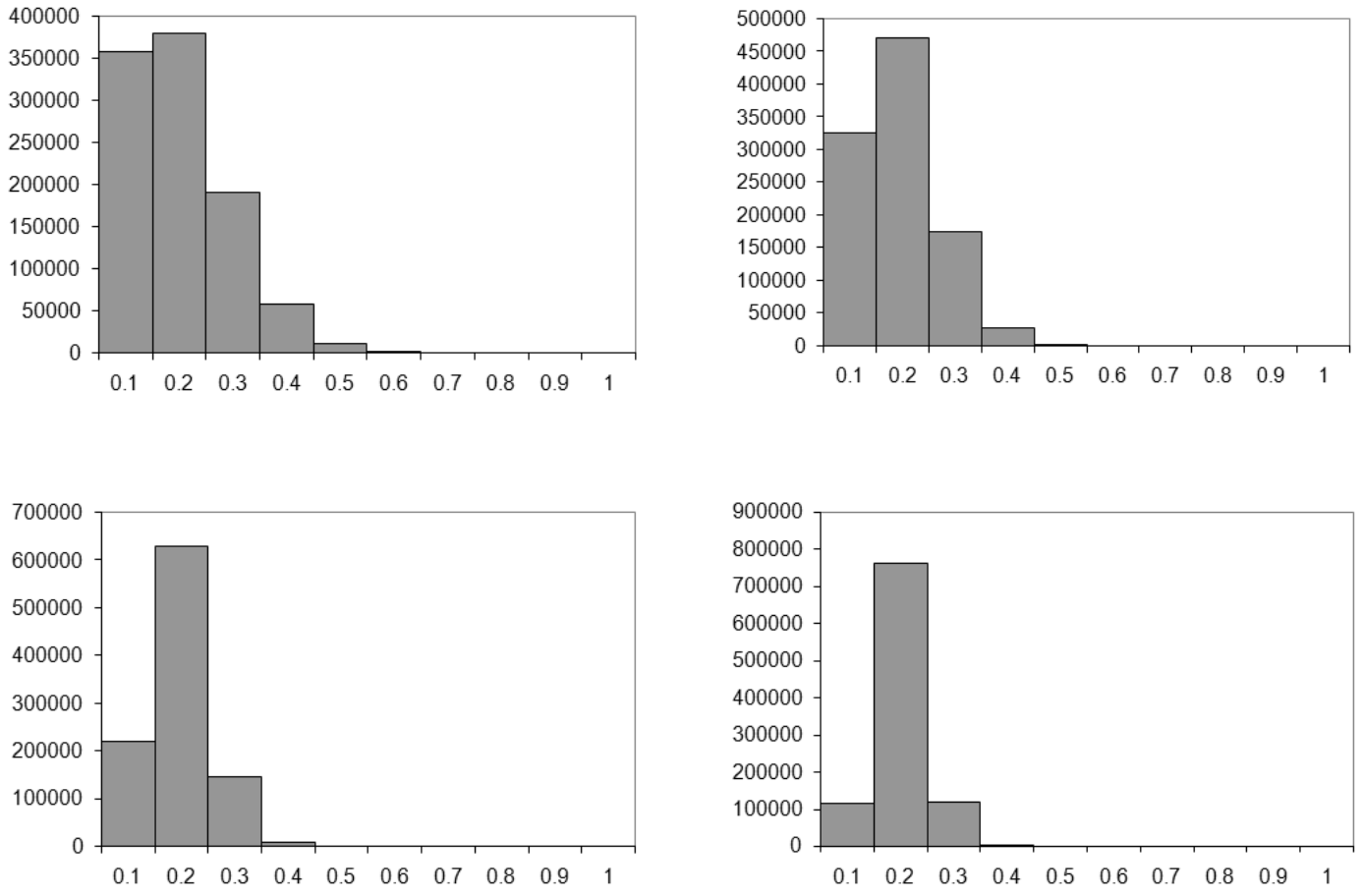


Figure 7

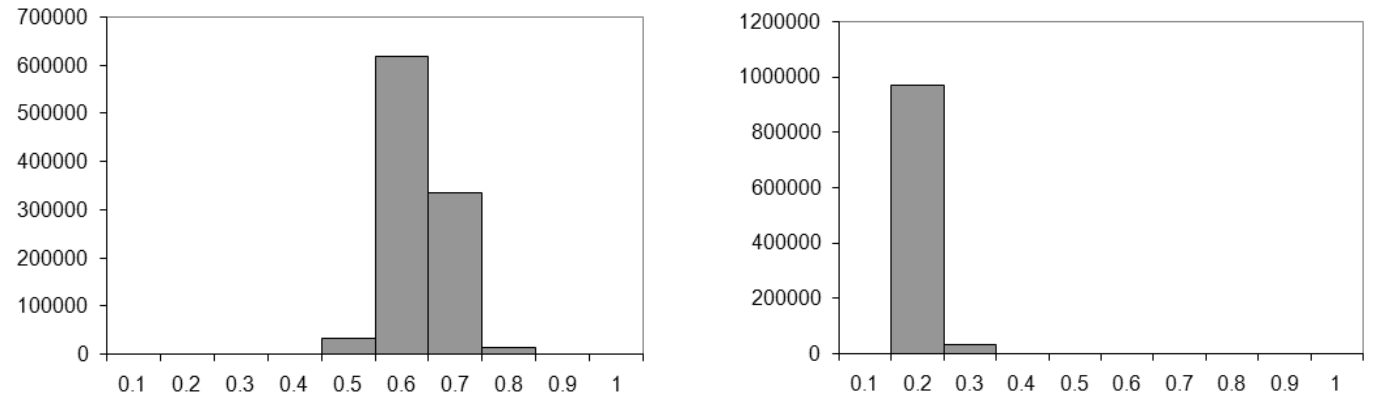


Figure 8

Jerome, J. K., *Three Men in a Boat*, J. W. Arrowsmith, 1889.

Nechaev, S., *Statistics of Knots and Entangled Random Walks*, arXiv:cond-mat/9812205 v1, www.arxiv.org, 1998.

Press et al., *Numerical Recipes in C*, 2nd ed., Cambridge, 1992.

Robbins, H., 'A Remark of Stirling's Formula', *American Mathematical Monthly*, vol. 62, pp. 26-29, 1955.

Theile, T. N., Om Anvendelse af mindste Kvadraters Methode i nogle Tilfælde, hvor en Komplikation af visse Slags uensartede tilfældige Fejlkilder giver Fejlene en 'systematisk' Karakter, Vidensk. Selsk. Skr. 5. Rk., naturvid. og mat. Afd., vol. 12, pp. 381-408, 1880.

RSA Made Simple

RSA is a common public key cryptography algorithm. Stuart Golodetz explains the mathematics behind it and shows us how to use it in Java.

As we all know, when designing a protocol for electronic communication we must continually bear in mind the security goals we need to satisfy. For example, we may wish to prevent our messages from being read by eavesdroppers, or to digitally sign our messages to guarantee that no-one else could have sent them. Cryptography, the science of encrypting and decrypting messages (almost invariably using ciphers), is the method of choice for these tasks.

An alternative method of trying to avoid our messages being read by a third party is steganography. This involves hiding the message being sent, e.g. in the pixels of an otherwise innocuous-looking image. A famous example is the message sent by Histiaeus of Miletus on the head of one of his slaves: he shaved the man's head, wrote the message and then waited for his hair to regrow before sending him to deliver the message. It goes without saying that this rather ingenious trick is only suitable for non-urgent messages.

One particularly interesting type of cryptography is asymmetric, or public key, cryptography. The idea works as follows. Everyone has two keys, a public key and a private key, neither of which can be derived from the other. They publish the public key and keep the private key hidden somewhere safe. The keys are such that:

- Messages encrypted with one of the keys can only be decrypted using the other.
- Each key can only decrypt messages encrypted with its partner.

Thus, if Alice wants to send a message to Bob that only the latter can read, she encrypts the message with Bob's public key, and (provided the encryption scheme is secure) can be sure that Bob is the only person who can decrypt it again, as only he has knowledge of his own private key.

To digitally sign a message to Bob to guarantee that it came from her, on the other hand, she would encrypt the message with her own private key. The encrypted message can be read by anyone, since Alice's public key is public knowledge, but only Alice can have sent the message, as no-one else had her private key and thus no-one else could have encrypted a message which could only be read using her public key. The two security goals we mentioned above can thus be amply satisfied by public key cryptography.

As a theory, this is all well and good, but it remains to be shown how to implement such a scheme. This is where RSA enters the picture.

The RSA algorithm

The RSA algorithm was first described in a 1977 paper by Ron Rivest, Adi Shamir and Leonard Adleman (hence the name), three researchers working at MIT, although an equivalent algorithm had actually been invented (but not published, for security reasons) by a GCHQ mathematician called Clifford Cocks four years earlier. Its foundation is the belief that factoring the product of two humongously large primes is computationally infeasible, i.e. it can't be done on a computer in a sensible amount of time. In theory, it should be possible (by choosing suitably large primes) to make it take an expected time longer than the expected lifetime of the universe, even if we used all the computers on Earth, but in practice this is rarely done. All that's really necessary is to make sure it can't be decrypted before the encrypted information's 'use by' date, i.e. the time at which the information contained within the message ceases to be of any use to an attacker. (It's worth noting that not all systems in current use achieve even this modest goal.)

Starting from the assumption of computational infeasibility, it is possible to generate suitable public and private keys which are mutual inverses but cannot be derived from each other. Let's take a closer look at how this works.

A little aside on Alice and Bob

Alice and Bob are traditional names in cryptography, presumably because A and B (which they clearly represent) felt a bit too impersonal. A third character, Eve, can be added when talking about eavesdropping.

Key pair generation

We start by randomly guessing two large (distinct) primes, p and q. This sounds difficult, but in practice it has been shown that the probability of a given number n being prime is roughly $1/\ln n$. (As an example, for 100-digit numbers this means that roughly 1 in every 230 numbers is prime.) Furthermore, efficient primality-testing algorithms like Miller-Rabin can be used to tell us whether a given number is prime or not. To pick two large primes, therefore, we simply guess lots of large random numbers and keep the first two primes we come across.

From these primes, we compute the product $n = p \times q$ and define $\varphi(n) = (p-1)(q-1)$. (Note that under the assumption that n cannot be factored, $\varphi(n)$ cannot be derived from n without knowledge of p or q.) We pick a small number e coprime to $\varphi(n)$ (65537 is a common choice), i.e. $\gcd(e, \varphi(n)) = 1$ and compute $d = e^{-1} \bmod \varphi(n)$ using an algorithm called the Extended Euclidean Algorithm. Finally, we publish (e,n) as our public key and keep (d,n) as our private key. All of this relies on a lot of mathematics (which is unfortunately beyond the scope of a short article like this one), but the basic steps are relatively easy to follow. The key point is that given either of the keys we've just generated and no additional information, it is computationally infeasible to calculate the other without being able to factor n, since we have no way of calculating $\varphi(n)$ from n.

Encryption and decryption

Encryption and decryption are both done the same way in RSA. Both use something called modular exponentiation. This is very like raising

Stuart Golodetz has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

Instead of encrypting our message with the public key and decrypting it with the private key, we swap them round

something to a power, but in modular arithmetic rather than the normal way of doing things. As an example, $2^3 \equiv 3 \bmod 5$, since 8 leaves remainder 3 when divided by 5. For RSA, then, we take a numeric encoding, m, of our message, and encrypt it using

$$c = m^e \bmod n$$

To decrypt it, we do

$$m' = c^d \bmod n$$

and we will find that $m' = m$. The reason this is true is down to a theorem produced by Euler, which tells us that for $n > 1$ and $a \in Z_n^* = \{b \mid 0 < b < n \text{ and } \gcd(b, n) = 1\}$,

$$a^{\varphi(n)} \equiv 1 (\bmod n).$$

Assuming that $m \in Z_n^*$, therefore, our proof that $m' = m$ goes through as follows:

$$\begin{aligned} m' &= c^d \bmod n \\ &= (m^e \bmod n)^d \bmod n \\ &= m^{ed} \bmod n \\ &= m^{1 + k\varphi(n)} \bmod n \\ &= (m \times (m^{k\varphi(n)} \bmod n)) \bmod n \\ &= (m \times ((m^{\varphi(n)} \bmod n)^k \bmod n)) \bmod n \\ &= (m \times (1^k \bmod n)) \bmod n \\ &= (m \times 1) \bmod n \\ &= m \end{aligned}$$

The proof relies on various properties of modular arithmetic, such as that $ab \bmod n = (a \bmod n \times b \bmod n) \bmod n$, etc. The key point to note is that $ed = 1 + k\varphi(n)$ for some k, since $ed \equiv 1 + \bmod \varphi(n)$. Having finished this proof, we now know that encrypting and decrypting the message will get us back to where we started from.

As mentioned before, it is also possible to use RSA for digital signing. This turns out to be very simple. Instead of encrypting our message with the public key and decrypting it with the private key, we swap them round. So a signed version of m can be produced by doing

$$s = m^d \bmod n$$

and verified using

$$m' = s^e \bmod n.$$

Once again, $m' = m$, using an entirely analogous proof.

RSA in Java

The theory behind RSA is interesting (if tricky in places), and I seriously recommend that you look into it in more detail. For the rest of this article, however, I want to approach things from a more practical standpoint and discuss how to use RSA in your Java code. This turns out to be rather simple, as an implementation already exists in the Java standard libraries. It works internally by treating ASCII text as a number in base 256 and then applying the RSA algorithm using the **`modPow`** method of **`java.math.BigInteger`**. (For more details, you might want to examine the files `RSACipher.java` and `RSACore.java` in the OpenJDK implementation at http://openjdk.java.net.)

Let's start by looking at how to generate keys (see Listing 1). The process is relatively straightforward: we start by getting a **`KeyPairGenerator`** instance which will generate RSA key pairs, initialise it using a key-size (1024 bits) and a 'cryptographically-secure' pseudo-random number generator (i.e. a random number generator which satisfies a number of desirable cryptographic properties), and use it to generate the keys. Finally, we extract the required private and public keys from the returned **`KeyPair`**.

Having generated the keys, we can use them to encrypt and decrypt messages. A simple example of this is shown in Listing 2. We get an instance of **`Cipher`** to do the actual encryption, initialise it with the required mode and RSA key, and set it to work on our message with a call to **`doFinal`**. Decryption works the same way. Note that the **`doFinal`** method works on byte arrays rather than strings (for various reasons, one of which is to allow it to encrypt more arbitrary data), so we have to convert between the two, but other than that there's very little effort we have to put in to get it to work.

Listing 2 shows a simple example using javax.crypto.Cipher.

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

...

KeyPairGenerator keyGen =
   KeyPairGenerator.getInstance("RSA");
SecureRandom random =
   SecureRandom.getInstance("SHA1PRNG", "SUN");

keyGen.initialize(1024, random);

KeyPair pair = keyGen.generateKeyPair();
RSAPrivateKey priv =
   (RSAPrivateKey)pair.getPrivate();
RSAPublicKey pub =
   (RSAPublicKey)pair.getPublic();

...
```

Listing 1

The best cryptography in the world won't help you if someone has access to your plaintext

Conclusions

RSA can be a complicated algorithm to fully understand, but using it in Java is relatively simple. If you take a look at the internal implementation of the algorithm (the source for the Java libraries is freely-downloadable), you'll see that the actual code which does the encryption and decryption amounts to a single line, namely a call to **`BigInteger.modPow`**. The only real work being done apart from that is converting between the textual and numeric representations of a message, and even that is relatively straightforward. This makes it easy to (for example) port the RSA implementation to other languages. For one of my current projects, I've ported it to PHP (another language for which a **`BigInteger`** class is conveniently available) with much less than an hour's work. Writing an implementation I trusted from scratch could easily have taken several days.

Ultimately, I hope that the message you'll take away from this article is that cryptography is interesting and worth exploring further. Algorithms like RSA may seem very mathematical and slightly scary, but with the ever-increasing emphasis placed on the security of our communications, it is vital that we make the effort to understand the tools we need to ensure it. Existing implementations of algorithms allow us to see how the mathematical theories translate into practical code, something that we as programmers can more readily understand, and are thus a valuable resource when trying to get to grips with a seemingly complicated algorithm like RSA.

Addendum

During the review process for this article, a number of interesting points were raised by the review team (many thanks). In particular, Roger Orr highlighted one very good reason for the Java implementers to choose byte arrays over strings for their implementation. Since the latter are immutable and Java itself is a garbage-collected language, the data contained in strings hangs around in memory until the garbage collector reclaims it. This can present a security hole, since anyone who can access your process's memory can continue to access (for example) the plaintext of your message, and you have no way to prevent this. Byte arrays, in contrast, don't suffer from the same issue, as the data in them can be manually overwritten with random rubbish.

The motto of this tale is that there is more to security than just cryptography. The best cryptography in the world won't help you if someone has access to your plaintext. This article doesn't pretend to be about security in general (it was intended more as a cryptographic taster), but it does go to show that security in a wider context can be seriously hard. ■

```
import javax.crypto.Cipher;

...

// Create the cipher
Cipher rsaCipher = Cipher.getInstance("RSA");

// Initialize the cipher for encryption
rsaCipher.init(Cipher.ENCRYPT_MODE, pub);

// Cleartext
byte[] cleartext = "This is just an example".getBytes();
System.out.println("Original cleartext: " + new String(cleartext));

// Encrypt the cleartext
byte[] ciphertext = rsaCipher.doFinal(cleartext);
System.out.println("Ciphertext: " + new String(ciphertext));

// Initialize the same cipher for decryption
rsaCipher.init(Cipher.DECRYPT_MODE, priv);

// Decrypt the ciphertext
byte[] cleartext1 = rsaCipher.doFinal(ciphertext);
System.out.println("Final cleartext: " + new String(cleartext1));

...
```

Listing 2

Quality Manifesto

Software quality is a systems engineering job.Tom Gilb explains the importance of knowing where we're going.

The main idea with this paper is to wake up software engineers, and maybe some systems engineers, about quality. The software engineers (sorry, 'softcrafters') seem to think there is only one type of quality (lack of bugs), and only one place where bugs are found (in programs). My main point here is that the quality question is much broader in scope. The only way to get total necessary quality in software is to treat the problem like a mature systems engineer. That means to recognize all critically interesting types of quality for your system. It means to take an architecture and engineering approach to delivering necessary quality. It means to stop being so computer program-centric, and to realize that even in the software world, there a lot more design domains than programs. And the software world is intimately entwined with the people and hardware world, and cannot simply try to solve their quality problems in splendid isolation. I offer some principles to bring out these points.

A quality manifesto

A group of my friends spent the Summer of 2007 emailing discussions about a Software Quality Manifesto. I was so unhappy with the result that I decided to write my own. At least I was unhampered by the committee.

Headline: **'Software Quality' is a Systems Engineering Job**.

Slogan:

Proposition:

'**Excellent system qualities** are a continuous management and engineering challenge, with no perfect solutions'.

Corollary:

'when **management and engineering fail** to execute their quality responsibilities professionally, the quality levels are accidental; and probably unsatisfactory to most stakeholders.'

Quality manifesto/declaration

- **System Quality** can be viewed as a set of quantifiable performance attributes, that describe how well a system performs for stakeholders, under defined conditions, and at a given time.
- **System Stakeholders** judge past, present, and future quality levels; in relationship to own their perceived needs/values.
- **System Engineers** can analyze necessary, and desirable, quality levels; and plan, and manage to deliver, a set of those quality levels, within given constraints, and available resources.
- **Quality Management** is responsible for prioritizing the use of resources, to give a satisfactory fit, for the prioritized levels of quality: and for trying to manage the delivery of a set of qualities – that maximize value for cost – to defined stakeholders.

Quality principles: an overview

Heuristics for action:

1. **Quality Design**: Ambitious Quality Levels are designed in, not tested in. This applies to work processes and work products.
2. **Software Environment**: 'Software' Quality is *totally dependent on its resident system quality*, and *does not exist alone*; 'software qualities' are dependent on a defined system's qualities – including stakeholder perceptions and values.
3. **Quality Entropy**: Existing or planned quality levels will deteriorate in time, under the pressure of other prioritized requirements, and through lack of persistent attention.
4. **Quality Management**: Quality levels can be systematically managed to support a given quality policy. Example : 'Value for money first', or 'Most competitive World Class Quality Levels'.
5. **Quality Engineering**: A set of quality levels can be technically engineered, to meet stakeholder ambitions, within defined constraints, and priorities.
6. **Quality Perception**: Quality is in the eye of the beholder: objective system quality levels may be simultaneously valued as great for some stakeholders, and terrible for others.
7. **Design Impact on Quality**: any system design component, whatever its *intent*, will likely have *unpredictable* main effects, and side effects, on *many* other quality levels, many constraints, and many resources.
8. **Real Design Impacts**: you cannot be *sure* of the totality of effects, of a design for quality, on a system, except by measuring them in *practice*; and even then, you cannot be sure the measure is *general*, or will *persist*.
9. **Design Independence**: Quality levels can be measured, and specified, independently of the means (or designs) needed to achieve them.
10. **Complex Qualities**: many qualities are best defined as a subjective, but useful, set of elementary quality dimensions; this depends on the degree of control you want over the separate quality dimensions.[1]

Tom Gilb is an international consultant, teacher and author.

His 9th book is *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage* (August 2005 Publication, Elsevier) which is a definition of the planning language 'Planguage'.

He works with major multinationals such as Credit Suisse, Schlumberger, Bosch, Qualcomm, HP, IBM, Nokia, Ericsson, Motorola, US DOD, UK MOD, Symbian, Philips, Intel, Citigroup, United Health, Boeing, Microsoft, and many smaller and lesser known others. See www.Gilb.com for contact details.

1 CE Chapter 5, download, http://www.gilb.com/community/tiki-download_file.php?fileId=26 will give rich illustration to this point. See for example Maintainability, Adaptability and Usability.

It is a well-known paradigm that you 'do not test quality into a system, you design it in'

Quality principles: detailed remarks

Heuristics for action:

1. **Quality Design**: *Ambitious* Quality Levels are **designed in**, not tested in. *This applies to work processes and work products*.

   There is far too much emphasis on testing and reviews, as a means of dealing with defects and bugs. It is a well-known paradigm that you 'do not test quality into a system, you design it in'. We can look at this problem from both an economic and an effectiveness point of view.

   From an economic point of view, it pays off, by one or two orders of magnitude, to solve problems early. 44–64% of all coding defects are the results of defects in specifications (requirements, design) given to programmers [GilbIFM], as reference for this and other facts about test and reviews]. The cost of removal of defects at late stages explodes by 10x to 100x and more. A stitch in time saves nine.

   From an effectiveness point of view, both tests and reviews are ineffective. The range of effectiveness is roughly 25% to 75% (probability of actually detecting defects that are present. [GilbIFM, CapersJones96]. Jones reckons that if we had an effective series of about 11 reviews and tests, we could only remove a maximum of 95% of the injected defects. My conclusion is that 'cleaning up injected defects' is a hopeless cause. There are better options.

   The interesting option is that 'an ounce of prevention is worth a pound of cure'. We have to learn to avoid the infection of defects in the first place. It is clear that we can reduce the injection rates by at least 100 to 1. Most requirements documents today (my personal client measurements) contain about 100 major defects per page (300 words). The standard that advanced developers (IBM [Humphrey89], NASA) have long since established is a tolerance (process exit level) of less than 1.0 majors/page (IBM : 0.25, NASA : 0.10). This is the primary focus of CMMI Level 5 (Defect Prevention Process [IBM90]). It takes my clients about 6 months to reduce injection by factor of 10, and another 2–3 years by another factor of 10. This is obviously more cost-effective than waiting until we can test for defects, or until customers complain.

2. **Software Environment**: 'Software' Quality is *totally dependent on its resident system* quality, and *does not exist alone*; 'software qualities' are dependent on a defined system's qualities – including stakeholder perceptions and values.

   We tend to treat software quality as something inherently resident in the software itself. But all qualities (example Security, Usability, Maintainability, Reliability) are highly dependent on people, their qualifications, and they way the use systems. The consequence is that we must plan, specify and design with a stronger eye to identifying and controlling the factors that actually decide the system quality. We have to engineer the system as a whole, not just the 'code'). We must be systems engineers, not program engineers.

   This has large implications for how we train people, how we organize our work, and how we motivate people. We will also have to shift emphasis from the technology itself (the means) to the results we actually need (the ends, quality requirement levels).

3. **Quality Entropy**: Existing or planned quality levels will deteriorate in time, under the pressure of other prioritized requirements, and through lack of persistent attention.

   Even the concept of numeric quality levels, for most qualities – example usability, security, adaptability – is alien to most software engineers, and to far too many systems engineers. But the basic concept of quantified quality levels is old and well established in engineering.

   In spite of this poor starting environment, of too many people satisfied with using words ('easy to use') instead of numbers ('30 minutes to learn task X by Employee type Y'), we need to do more than merely achieve planned quality levels upon initial delivery and acceptance of systems. We need to imbed in the systems the measurement of these qualities, and the warning systems needed to tell us they are deteriorating or have drastically fallen. We need to expect to take action to improve the quality levels back to planned levels, and perhaps improve them even more in the future.

4. **Quality Management**: Quality levels can be systematically managed to support a given quality policy. Example: 'Value for money first', or 'Most competitive World Class Quality Levels'.

   It is useful management if there is a policy about the levels of quality we aspire to, both at a corporate level, and a project level. We cannot really allow isolated individuals to make their dream levels of quality be taken as requirements, without due balance towards the priorities of the other competing levels. And we need to keep our eyes on available resources and technological limits and opportunities.

   We need to decide if we are there to 'be the state of the art' (as Rockwell explained to me once) or 'get the most value for money', as others need to worry about that more.

   A policy like this might be generally useful: 'Quality levels will be engineered to a level that gives us arguably high return on the investment needed to get them there, and so that the levels do not steal resources for other parallel investment opportunities in quality, or elsewhere.'

5. **Quality Engineering**: A set of quality levels can be technically engineered, to meet stakeholder ambitions, within defined constraints, and priorities.

   It is a tricky business to decide which numeric quality levels are appropriate. Initially we cannot decide the right levels in isolation. We need to know about the larger environment, both the environment for the single quality attribute, and for the set of attributes – for their environment.

Elementary scalar requirement template <with hints>
Tag: <Tag name of the elementary scalar requirement>.
Type:
<{Performance Requirement: {Quality Requirement,
Resource Saving Requirement,
Workload Capacity Requirement},
Resource Requirement: {Financial Requirement,
Time Requirement,
Headcount Requirement,
others}}>.
========== Basic Information==========
Version: <Date or other version number>.
Status: <{Draft, SQC Exited, Approved, Rejected}>.
Quality Level: <Maximum remaining major defects/page, sample size, date>.
Owner: <Role/e-mail/name of the person responsible for this specification>.
Stakeholders: <Name any stakeholders with an interest in this specification>.
Gist: <Brief description, capturing the essential meaning of the requirement>.
Description: <Optional, full description of the requirement>.
Ambition: <Summarize the ambition level of only the targets below. Give the overall real ambition level in 5–20 words>.
========== Scale of Measure ==========
Scale: <Scale of measure for the requirement (States the units of measure for all the targets, constraints and benchmarks) and the scale qualifiers>.
========== Measurement ==========
Meter: <The method to be used to obtain measurements on the defined Scale>.
========== Benchmarks ========== ''Past Numeric Values'' ==========
Past [<when, where, if>]: <Past or current level. State if it is an estimate> <- <Source>.
Record [<when, where, if>]: <State-of-the-art level> <- <Source>.
Trend [<when, where, if>]: <Prediction of rate of change or future state-of-the-art level> <-<Source>.
========== Targets ========== ''Future Numeric Values'' ==========
Goal/Budget [<when, where, if>]: <Planned target level> <- <Source>.
Stretch [<when, where, if>]: <Motivating ambition level> <- <Source>.
Wish [<when, where, if>]: <Dream level (unbudgeted)> <- <Source>.
========== Constraints ========== ''Specific Restrictions'' ==========
Fail [<when, where, if>]: <Failure level> <- <Source>.
Survival [<when, where, if>]: <Survival level> <- <Source>.
========== Relationships ==========
Is Part Of: <Refer to the tags of any supra-requirements (complex requirements) that this requirement is part of. A hierarchy of tags (For example, A.B.C) is preferable>.
Is Impacted By: <Refer to the tags of any design ideas that impact this requirement> <-<Source>.
Impacts: <Name any requirements or designs or plans that are impacted significantly by this>.
========== Priority and Risk Management ==========
Rationale: <Justify why this requirement exists>.
Value: <Name [stakeholder, time, place, event]: Quantify, or express in words, the value claimed as a result of delivering the requirement>.
Assumptions:<State any assumptionsmade in connection with this requirement> <-<Source>.
Dependencies: <State anything that achieving the planned requirement level is dependent on> <- <Source>.
Risks: <List or refer to tags of anything that could cause delay or negative impact> <- <Source>.
Priority: <List the tags of any system elements that must be implemented before or after this requirement>.
Issues: <State any known issues>.

Figure 1

it is easier to be confident if no particular numeric impact is ever asserted

We need to learn to specify this environment together with the requirement ideas themselves. It will be easier to make decisions about the relative levels of quality and their priority if we have a decisive set of facts about each attribute. For example, it is useful to know things like the:

- Value for a level
- The stakeholders for a quality and for various levels
- The timing needs of levels of quality
- The planned strategies and their expected costs for reaching given levels.

And quite a few other things – that will help us reason about the right levels of quality.

Figure 1 is an example of a template that tries to collect some of the information that I think we ought to know about in order to decide how to prioritize particular levels of quality for a single attribute. [Gilb05]

6. **Quality Perception**: Quality is in the eye of the beholder: objective system quality levels may be simultaneously valued as great for some stakeholders, and terrible for others.

   The point is that any really complex, large system will have many different stakeholders. Even one stakeholder category (Novice User, Call Center Manager) can have many individuals, with highly individual needs and priorities. The result will inevitably be a compromise. But we can make that compromise as intelligent as possible. We do not have to design systems with only one level for all stakeholders. We can consciously decide to have different quality levels of the same quality, for different stakeholders, at different times and situations.

   An example is shown in Figure 2.

7. **Design Impact on Quality**: any system design component, whatever its *intent*, will likely have *unpredictable* main effects, and side effects, on *many* other quality levels, many constraints, and many resources.

   I see far too much narrow reasoning, of the type: 'we are going to achieve great quality X using technology X, Y and Z'. This reasoning is not with numbers, but only nice words. Yet I have seen in it $100 million projects, often!

Learnability:

Scale: the time needed for a defined [Stakeholder] to Master a defined [Process].

Goal [Stakeholder = Top Manager, Process = Get Report] 5 minutes.

Goal [Stakeholder = Offshore Clerk, Process = Create New Account] 1 hour.

Figure 2

	On-line Support	On-line Help	Picture Handbook	On-line Help + Access Index
Learning Past: 60 min <<-> 10 min				
Scale Impact	5 min	10 min	30 min	8 min
Scale Uncertainty	±3 min	±5 min	±10 min	±5 min
Percentage Impact	110%	100%	67% (2/3)	104%
Percentage Uncertainty	±6% (3 of 50 minutes)	±10%	±20% ?	±10%
Evidence	Project Ajax, 1996, 7 min	Other Systems	Guess	Other Systems + Guess
Source	Ajax report, p.6	World Report, p.17	John B.	World Report + John B.
Credibility	0.7	0.8	0.2	0.6
Development Cost	120K	25K	10K	26K
Benefit-to-Cost Ratio	110/120 = 0.92	100/25 = 4.0	67/10 = 6.7	104/26 = 4.0
Credibility adjusted to B/C Ratio (to 1 decimal place)	0.92*0.7 = 0.6	4.0 * 0.8 = 3.2	6.7 * 0.2 = 1.3	4.0 * 0.6 = 2.4
Notes: Time Period is two years.	Longer timescale to develop			

Figure 3

Learnability:

Scale: minutes to learn a Task by a User.

Meter [Weekly Development, 2 Users, 10 Normal tasks]

Meter [Acceptance Test, Duration 60 day, 200 Users, 10 normal tasks, 20 extreme tasks]

Meter [Normal Operation, Sampling Frequency 2%, Tasks = All Defined]

Figure 4

We can never take critical qualities for granted, or act as if they are stable

We have to learn to specify, analyze and think in terms of 'multiple numeric impacts of many designs, on our many critical quality and cost requirements'. Quality Function Deployment (QFD) takes this position, but I am not happy with the way in which numbers are used in QFD) – too subjective, too undefined [GilbQFD].

We need to systematically, as best we can, estimate all the multiple effects or each significant design.

Figure 3 shows a systematic analysis of four designs on one quality level (10 minutes). This is an impact estimation table. [Gilb05].

8. **Real Design Impacts**: you cannot be *sure* of the totality of effects, of a design for quality, on a system, except by measuring them in *practice*; and even then, you cannot be sure the measure is *general*, or will *persist*.

   I have seen books, papers, and project specifications for software that confidently predict a good result (not usually quantified) from a particular design, solution, architecture or strategy. Maybe it is easier to be confident if no particular numeric impact is ever asserted.

   In normal engineering, no matter what the engineering handbook says, no matter what we would like to believe; the prudent engineer takes the trouble to measure the *real* effects.

   We need to carefully do early measurements, then repeat measurements when scaling up, at acceptance times, and later in long-term operation. We can never take critical qualities for granted, or act as if they are stable.

   We can plan this in advance to a reasonable degree (see Figure 4).

   Each 'Meter' specification defines or sketches a different intended test to measure the quality level.

9. **Design Independence**: Quality levels can be measured, and specified, independently of the means (or designs) needed to achieve them.

   There is far too much immediately coupling of named design ideas, with named quality types. 'We will improve product agility using structured tools' – type of specification.

Maintainability:
Type: Complex Quality Requirement.
Includes: {Problem Recognition, Administrative Delay, Tool Collection, Problem Analysis, Change Specification, Quality Control, Modification Implementation, Modification Testing {Unit Testing, Integration Testing, Beta Testing, System Testing}, Recovery}.

Problem Recognition:
Scale: Clock hours from defined [Fault Occurrence: Default: Bug occurs in any use or test of system] until fault officially recognized by defined [Recognition Act: Default: Fault is logged electronically].

Administrative Delay:
Scale: Clock hours from defined [Recognition Act] until defined [Correction Action] initiated and assigned to a defined [Maintenance Instance].

Tool Collection:
Scale: Clock hours for defined [Maintenance Instance: Default: Whoever is assigned] to acquire all defined [Tools: Default: all systems and information necessary to analyze, correct and quality control the correction].

Problem Analysis:
Scale: Clock time for the assigned defined [Maintenance Instance] to analyze the fault symptoms and be able to begin to formulate a correction hypothesis.

Change Specification:
Scale: Clock hours needed by defined [Maintenance Instance] to fully and correctly describe the necessary correction actions, according to current applicable standards for this. Note: This includes any additional time for corrections after quality control and tests.

Quality Control:
Scale: Clock hours for quality control of the correction hypothesis (against relevant standards).

Modification Implementation:
Scale: Clock hours to carry out the correction activity as planned. "Includes any necessary corrections as a result of quality control or testing."

Modification Testing:

Unit Testing:
Scale: Clock hours to carry out defined [Unit Test] for the fault correction.

Integration Testing:
Scale: Clock hours to carry out defined [Integration Test] for the fault correction.

Beta Testing:
Scale: Clock hours to carry out defined [Beta Test] for the fault correction before official release of the correction is permitted.

System Testing:
Scale: Clock hours to carry out defined [System Test] for the fault correction.

Recovery:
Scale: Clock hours for defined [User Type] to return system to the state it was in prior to the fault and, to a state ready to continue with work.

Source: The above is an extension of some basic ideas from Ireson, Editor, Reliability Handbook, McGraw Hill, 1966 (Ireson 1966).

Figure 5

2 CE Chapter 10, download, http://www.gilb.com/community/tiki-download_file.php?fileId=77 will illustrate this point.

Specifying a 'design', when you need to focus on the quality level, should be considered a major defect

We need to focus our specifications on the quality levels we require, and studiously avoid mentioning our favoured design idea in the same sentence.

Specifying a 'design', when you need to focus on the quality level, should be considered a major defect in the specification. Dozens or more such 'false requirements' per page of 'requirements' are not uncommon in our 'software' culture.

10. **Complex Qualities**: many qualities are best defined as a subjective, but useful, set of elementary quality dimensions; this depends on the degree of control you want over the separate quality dimensions.[2]

    I think there is too little awareness of the fact that quality words often are the name of a set of qualities. The only way to define such complex qualities is to list all the components of the set. Only in this way will we understand what the real requirements are.

    We need to learn the general patterns of the most common qualities, as in the example below.

    We need to avoid oversimplification of qualities when the detailed set of sub-attributes will give us a fair chance at getting control over the critical qualities we want to manage.

Figure 5 shows an example of Maintainability as a set of other measures of quality. [Gilb05].

Summary

Purpose [of Quality Manifesto]:

To promote a healthy view of software quality.

Gap Analysis:

To help people get to where they really need to be in order to meet their stakeholders expectations as well as resources permit.

Justifications [for positions taken here]

1. We must take a **systems-centric**, not a programming-centric view of quality.

   Because: Software only has quality attributes in relation to people, hardware, data, networks, values. It cannot be isolated from the related world that decides

   - which quality dimensions are of interest (critical)
   - which quality levels are of value to a given set of stakeholders.

2. We must take a **'stakeholder' view** – not customer or user or any much-too-limited limited set of stakeholders.

   Because: the qualities that must be engineered and finally present in a software system depend on the entire set of critical stakeholders, not a on a limited few.

3. We must make a clear **distinction** between **various 'defect' types**, as good IEEE engineering standards already do.

   Because: we cannot afford to confuse specification defects, with their potential product faults, and product faults with potential product malfunctions. See these definitions. ■

References and further reading

[CapersJones96] Capers Jones, *Applied Software Measurement*, McGraw Hill, 2nd edition 1996; ISBN 0070328269.

[Gilb05] Gilb, Tom, *Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, ISBN 0750665076, 2005, Publisher: Elsevier Butterworth-Heinemann. Sample chapters will be found at Gilb.com.

- Chapter 5: Scales of Measure: http://www.gilb.com/community/tiki-download_file.php?fileId=26
- Chapter 10: Evolutionary Project Management: http://www.gilb.com/community/tiki-download_file.php?fileId=77

Gilb.com: www.gilb.com. our website has a large number of free supporting papers , slides, book manuscripts, case studies and other artifacts which would help the reader go into more depth

For example:

- [GilbIFM] Gilb, 'Inspection for Managers', a set of slides with facts and cases. http://www.gilb.com/community/tiki-download_file.php?fileId=88
- [GilbQFD] Gilb: 'What's Wrong with QFD?' http://www.gilb.com/community/tiki-download_file.php?fileId=119

[Humphreys89] Humphreys, W. S., *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989

[IBM90] Mays, R. G.; Jones, C. L.; Holloway, G. J.; Studinski, D. P., 'Experiences with Defect Prevention' in *IBM Systems Journal* Vol 29, No 1, 1990, available from: http://www.research.ibm.com/journal/sj/291/ibmsj2901C.pdf

NCOSE Systems Engineering Handbook v. 3 INCOSE-TP-2003-002-03, June 2006 , www.INCOSE.org

Software World Conference Website: Bethesda Md., USA, September 15-18th 2008 http://www.asq509.org/ht/display/EventDetails/i/1837 0

Source: of Quality Opinions: http://www.qualitydigest.com/html/qualitydef.html[2001]