overload 87

OCTOBER 2008 £3

DynamicAny
The second peek at DynamicAny, looking into its performance and size

Focus, Quality, Time-boxes and Ducks
More "On Management", focusing our teams on their lines of ducks

The Model Student
We find out whether chance can make fine things

Seeing Things Differently
Building 3D models from 2D slices in C++ using the Multiple Material Marching Cubes algorithm

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 87

October 2008

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 88 should be submitted to the editor by 1st November 2008 and for Overload 89 by 1st January 2009.

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

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.

The Invisible Hand

Large groups can behave as one, but how predicatable are they?

Welcome to Overload 87. If you were thinking the post had something missing, then you'd be right – there's no CVu this month. This is the first of the new monthly mailings, and we're starting with Overload. You'll get the next edition of CVu next month followed by Overload the month after, and so on. In this period of change it would be good to have a think about what we'd like from CVu and Overload, what's good, and what could be improved. I welcome opinions and ideas on this or any other issue.

The domino effect

I'm writing this at the end of September, after a few quite remarkable weeks in international finance and the markets, after a year of the so called Credit Crunch crisis. Perhaps by the time you read this the causes and consequences will be clearer, but I thought one story in particular was interesting because of the role of technology.

Our world now uses computers to do huge amounts of work. Examples include online newspapers plus their archives, indexing for searching, automatic collation of information sources, processing vast numbers of market trades, and even automating tasks using rule-based systems.

But recently all these examples came together and caused major headaches for United Airlines [BBC].

What appears to have happened started with a newspaper, the *Florida Sun-Sentinel*, which has an online version with an archive of all past stories. For some unknown reason, enough people clicked on a story about United Airlines to put a link to it on its 'Most Popular Stories' section. Unusually this story was an old one from around 2002, when many airlines were struggling with the aftermath of 9/11, and was about United filing for Chapter 11 protection (which roughly means bankruptcy). However, when you looked at the web page the only date visible was the current date: September 7th, 2008.

Google News' web crawler found the page, and reasoning that as it hadn't seen that link last time and the only date it could find was recent, decided that it was a new news story and duly indexed and published it. As airlines are having a rough time due to high oil prices recently, such a story was plausible and of interest, causing plenty of people to read it, which made it rise up the rankings, gaining even more attention. Then someone thought it important enough to put it on the Bloomberg newswire service that is used by the financial markets.

At which point traders and automated trading systems saw the bad news and sold United Airlines stock, causing the price to drop quickly. This triggered automatic stop-loss rules (that is, if the price falls below a pre-set limit, sell your shares to avoid losing even more), which sent the price even lower, triggering more automatic stop-loss rules and panicking traders, and so on in a vicious cycle. By the time the stock was suspended, it had lost around 75% of its value, around $1 billion, in just fifteen minutes! All because a few people had clicked on an old story...

Interestingly, pretty much everyone had acted rationally and cannot really be blamed (I would say the exceptions are the original website's developers for not clearly tagging stories with a date and time, and the journalist who posted the story to Bloomberg without checking it sufficiently.) And yet the consequences were anything but rational.

One big problem here was the way computers automated some simple rules which were fine in isolation, but when combined with many other similar rules led to the computer equivalent of a market panic. Because of course, markets aren't as perfectly efficient and rational as some simple theories make out, for two reasons – they involve people who can respond irrationally to rumour or panic or exuberance; and computers blindly carrying out rules, that while locally rational in normal market conditions, interact with the rest of the system to produce an overall irrational result in unusual situations.

This is just one example of how when things are highly interconnected, effects can ripple out and unexpected emergent behaviour can arise suddenly. (A version of the Law Of Unintended Consequences [Wikepedia])

These can be almost impossible to predict, and take some tricky mathematics to model, but computers are making it possible to process the vast amounts of data involved, or even simulate these huge networks of autonomous agents, and understand some of the resulting behaviours. This could be very useful for policy makers trying to predict the effects of new laws, taxes, incentives etc, which cause all sorts of unexpected results as people try to work around or game the new systems.

The book *Critical Mas*s [Ball] is a readable introduction to how some of these problems can be tackled, looking at the ways large groups act according to the statistical laws that were first used to model molecule velocities in a fluid. So while each agent acts independently and unpredictably according to their own desires and circumstances, the higher-level pattern that emerges is often highly ordered and predictable.

As well as the trading patterns of markets, other examples include the properties of matter arising from the interactions of atoms and molecules;

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.

the structure of galactic spiral arms (which are thought to be pressure waves and not static structures [SpiralArms]); traffic flow and how congestion occurs; how birds flock; epidemiology; the spread of email viruses along social networks; prediction of power grid usage; population changes and urban planning; and the evolution of populations and genomes.

This latter is an interesting one: the basics of evolution are so simple – all you need is reproduction (so successful populations grow), heritable variation (to produce a range of creatures), and inadequate resources (so that not every creature can reproduce) – and yet the results are enormously complex. Partly because the sheer enormity of the time involved allows changes to accumulate, but also because of the complex feedback of each creature itself being part of the environment that determines which genomes survive (a classic example is the arms race between preditor and prey). For a taster, Richard Harris looks at some of the maths behind how natural selection builds complexity.

The black art of estimating release dates

These sorts of ideas can also be applied to trying to work out when a product will be good enough to ship. A simple rule-of-thumb is to look at a graph of open bugs over time and estimate when it will go down to zero. Of course this is an overly simplistic model – depending on where you are in the development cycle, the number of bugs could be increasing!

So obviously there are different phases that have their own distinctiveness: such as development of features, stabilization, first mass test, release preparation, and post-release.

We can categorise bugs in many ways, such as the probability of being triggered, how severe are their effects, what effort is needed to fix them, and the risk of the fix causing further bugs (and what sort those are). Each development phase has a different mix of bugs. For example, early on creating new features will cause many new bugs to be created of many different types, from trivial spelling mistakes to serious design flaws; conversely, close to release most of the easy bugs will have been found and fixed, the design has settled down, and what remains will be the hard bugs whose fix will likely cause many more bugs.

Thinking about this reminded me of Ross Anderson's keynote at the conference a few years ago [Anderson] which analysed what would be the expected number of security flaws in open and closed source software. Doing a similar analysis of expected bug numbers based on the number of bugs already found, the number of testers, and the rate of fixes, could provide a much better set of rules that can be easily applied. For example, when in the pre-release phase, work out how much effort it took to get bug numbers halved from the peak, and the expected effort to get it to zero will be twice that. (This is just my guess based on the fact that you won't have found all the bugs yet, mostly the hardest bugs will be left, and fixes will cause further bugs).

Others have done just such an analysis: a quick search for terms such as Reliability Growth Models, or Software Release Estimation reveals plenty of examples and research into their effectiveness. And yet I've rarely seen or heard about people doing much more than simple extrapolation and guestimates. This is probably good enough for a rough guess for non-critical software, but as most projects are classed as 'failed' due to being later than predicted there's plenty of room for improvement.

As an example of how using even slightly more sophisticated models can help, I once worked on a team where things always took longer than the estimates. So I introduced a sightly different technique. Instead of asking how long something would take, I asked for estimates in three situations: a best case estimate if everything went really well, most likely (which is the estimate people usually give you), and a worst case if things go badly. These numbers tell you several things: if they vary wildly, it indicates the task is risky and probably not very well understood. But interestingly they are rarely symmetrical – e.g. a task that is most likely to take 10 days will have a best case estimate of 7 days, but a worst case of 20. By combining these three numbers in a simple weighted distribution, something like E=(best + 4 x likely + worst)/6, you get the Expected time, which was always a little bit longer than the 'likely' time, e.g. (7 + 4x10 + 20)/6 is 11.16. By using this number in the plans instead (and adding a half day a week for synchronising with the main code base and checking in), the estimates became remarkably accurate, and everyone became a lot happier.

References

[Anderson] Anderson, Ross 'Open and Closed Systems are Equivalent (that is, in an ideal world)' http://www.cl.cam.ac.uk/~rja14/Papers/toulousebook.pdf

[Ball] *Critical Mass* – Philip Ball. ISBN 0374281254

[BBC] http://news.bbc.co.uk/1/hi/business/7605885.stm

[SpiralArms] http://www.astronomynotes.com/ismnotes/s8.htm

[Wikipedia] http://en.wikipedia.org/wiki/Unintended_consequence

Seeing Things Differently

The Multiple Material Marching Cubes (M3C) algorithm builds 3D models from 2D slices. Stuart Golodetz introduces it and provides a C++ implementation.

Visualizing organs and other features of interest (such as the spine, or tumours, etc.) in 3D is an important process in the medical domain. It makes it much easier for doctors to picture the overall size of a tumour and its relative spatial location if they can see it in 3D, rather than having to try and mentally visualize what's going on after looking through a series of 2D slices. As part of my medical imaging work during my doctorate, I recently spent some time implementing the multiple material marching cubes (M3C) algorithm [Wu03], one of the methods currently used for this process, and (as is often the case when implementing algorithms from research papers, which can only ever give you an overview of the process) the experience turned out to be rather interesting. In my next couple of articles, I'd like to describe not just the algorithm itself (which is probably described more precisely in the original research paper), but how it can be implemented at a code level.

A 3-stage process

First of all, though, let's discuss the algorithm at a somewhat higher level. M3C takes a labelled volume (grid) of data as input, and produces a coloured triangle mesh as output (see Figure 1).

Mathematically-speaking, given a label set L = {i | i in [0, n)}, the input is a scalar field $f : R^3 |_{[0,X) \times [0,Y) \times [0,Z)} \to L$, such that f(x,y,z) indicates the label assigned to grid point (x,y,z). (For example, point (1,2,3) could be labelled with the number 5, which might correspond to the kidney, and indicate that point (1,2,3) should be considered as being inside the kidney.) Such a labelled volume can be constructed, with varying degrees of automation, from a sequence of medical images by a process known as segmentation (something I've referred to in previous articles [Golodetz08]).

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

The algorithm works in 3 stages: in the first stage, a basic mesh is generated from the volume. This is generally stair-stepped (meaning exactly what it sounds like) and contains far more triangles than it is possible to render at a reasonable frame-rate. The second and third stages thus smooth the mesh (to mitigate the stair-stepping) and decimate it (to drastically reduce the triangle count) respectively.

Stage 1: Basic mesh generation (overview)

The mesh generation process treats the volume as a 3D array of voxels (volume elements) or, to put it another way, as a large number of small cubes. Each label in the volume corresponds to one of the vertices at the points where adjacent cubes meet (as opposed to each voxel having a particular label), thus each cube has 8 labels associated with it, one for each of its vertices (see Figure 2). These labels may all be the same, or all different, or somewhere in between. (In principle, there are thus 8^8 different combinations which can arise, since there are 8 vertices, and 8 possible label choices per vertex. Note that it doesn't matter that there are

M3C converts a labelled volume (left) into a coloured triangle mesh (right). (Note that I've used symbolic rather than numeric labels here for clarity only; the actual labels are numeric. Note also that the labelled volume on the left here is not the actual labelled volume for the mesh on the right, which is huge!)

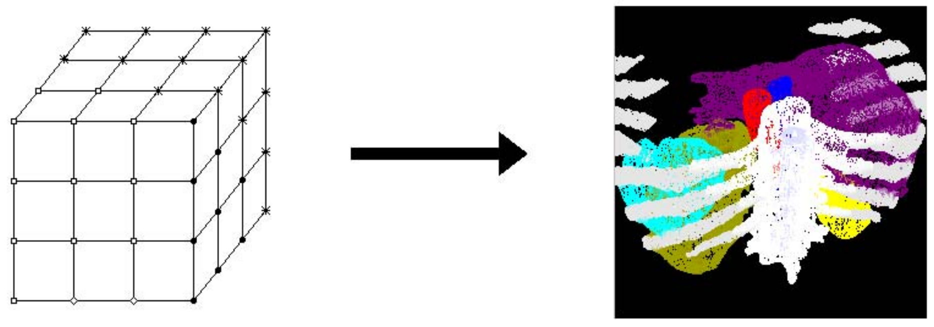


Figure 1

Each cube has 8 labels associated with it, one for each of its vertices.

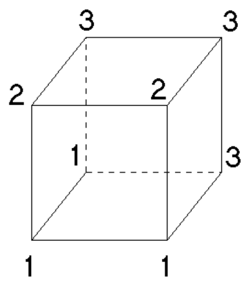
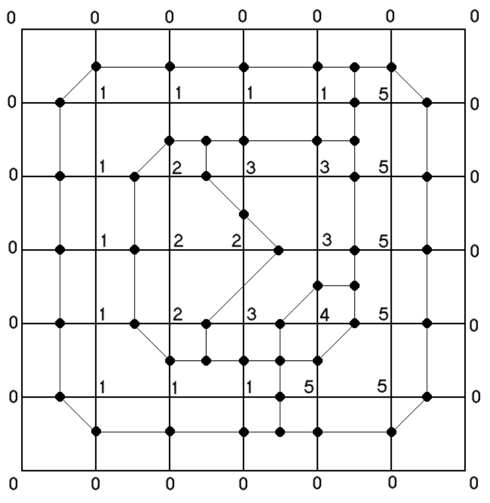


Figure 2



Meshing a 6x6 labelled area in 2D (left) using the patterns shown (right). The patterns show (top-to-bottom, left-to-right) how to handle squares with 2 labels split (1,3); 2 labels split (2,2) opposite; 2 labels split (2,2) diagonal; 3 labels opposite; 3 labels diagonal; and 4 labels.

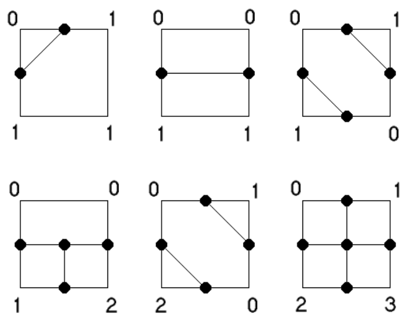


Figure 3

more than 8 labels in the entire volume when saying this, since there can only be at most 8 in the same cube at once.)

The mesh is generated on a cube-by-cube basis by adding faces to separate adjacent labels which do not match. This has to be done in a cunning way to ensure that the mesh pieces in adjacent cubes ‘match up’. To get a feel for how this process works, we’ll first examine the 2D analogue, namely multiple material marching squares (M3S). M3S turns out to be an important subroutine when we repeat the process in 3D, as we’ll see later.

Consider Figure 3a, which shows the result of ‘meshing’ a 6x6 labelled area in 2D. We treat each square individually, and match it against a set of patterns (see Figure 3b). In each case, the pattern dictates how to mesh the square by (potentially) adding vertices at the midpoints of the square’s edges and its centre, and adding mesh edges to join them together. When a vertex is added, it is assigned a set of labels based on the material types it separates. A node at the midpoint of one of the square's edges is assigned the labels of the two corner vertices that edge connects; the centre node (if any) is assigned all the labels present in the square. The mesh piece for each square will contain between 0 and 5 vertices (there are 4 midpoints, plus the square centre) and between 0 and 4 mesh edges. Note that adjacent squares share the midpoint vertices: they can be thought of as belonging to the square edge rather than the square itself.

The patterns themselves are mostly straightforward (note that those not explicitly drawn out are derived by symmetry), and adjacent patterns are guaranteed to match up by design. The only tricky point occurs in Case 3 (the top-right pattern), where there is an ambiguity over whether the material labelled 0 or that labelled 1 should remain contiguous. This was a known problem with the single-material marching cubes algorithm, which M3C extends. It is resolved here by partially-ordering the materials and keeping the lower label contiguous; thus 0 has priority over 1 here.

Having seen how M3S works, we are now ready to consider the equivalent in 3D. The idea here is similarly to generate mesh pieces for each cube so that adjacent mesh pieces join up to form a valid mesh. The way we ensure validity is to use M3S (the 2D analogue) to generate patterns on each cube face; we then connect the vertices generated on the cube faces with triangles within each cube. Since the cube faces (and thus the patterns on them) are shared between adjacent cubes, this guarantees that the mesh pieces will join up appropriately.

The trick now is in how to generate the triangles within each cube. Unfortunately, this turns out to be a somewhat intricate process (but at least it’s fun!). The first step is to count the number of centre nodes that have been created by M3S on the cube faces. There are three distinct cases with which to deal: 0 such nodes, 2, or more than 2 (Wu proved in his dissertation that there can’t be exactly one face-centred node). The goal in all cases is to try and determine a number of node loops (see Figure 4) that we can then triangulate, as we’ll see shortly.

In the case of 0 face-centred nodes, we don’t need to do anything special. The nodes and edges added by M3S already form closed loops, and their nodes share the same labels. The situation is more complicated when there are two or more face-centred nodes. In the former situation, we need to add an edge between the two face-centred nodes to form the node loops (see Figure 5). The loops are then distinguished by the fact that there are exactly two labels in common for all the nodes in a loop.

The situation with more than two face-centred nodes is even more complicated, and necessitates adding an extra node at the centre of the cube itself. This is assigned all the labels in the cube, and an edge is added between it and each face-centred node.

Having ensured that the node loops (or triangulatees, as I will now call them, since they are entities that will later need to be triangulated in one of two ways) we’re looking for are actually present, the next step is to find them. This process is not described in the original paper, but we can use an algorithm similar to depth-first search to do the job.

We find triangulatees one at a time, until there are no more left to find. The key to this is to find a start node for a loop with exactly two material IDs and a remaining edge. (Once no more such nodes exist, we’ve found

An example showing a cube with two node loops (based on Figure 6b in [Wu03]: see that paper for further examples).

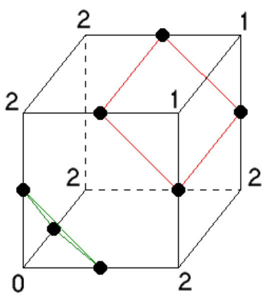
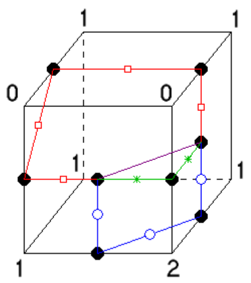
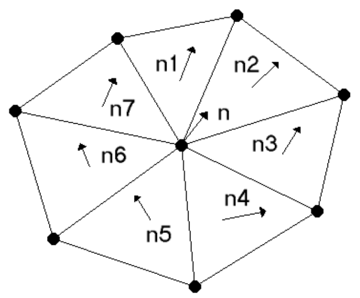


Figure 4



If there are exactly two face centres, we need to add an edge joining them to connect up the node loops. (The symbols – square, circle, star – indicate edges which belong to the same node loop. The edge we need to add – shown without a symbol – is shared between the three node loops.) This image is based on Figure 7a of [Wu03], but is slightly simplified.

Figure 5



We calculate the average plane of the node loop by imagining a fan-triangulation of it from an imaginary centre point and performing a weighted average of the normals of the fan triangles to deduce the average plane normal.

Figure 6

The Schroeder triangulation process as a tree. At each stage, we divide the node loop (shown here as a polygon) across one of its diagonals and recurse on both halves. Eventually, we end up with triangles at the leaf nodes (unless a suitable diagonal could not be found at any point, which only happens with more complicated node loops that do not crop up in M3C). Bear in mind that the node loops are generally non-planar (although this page is!).

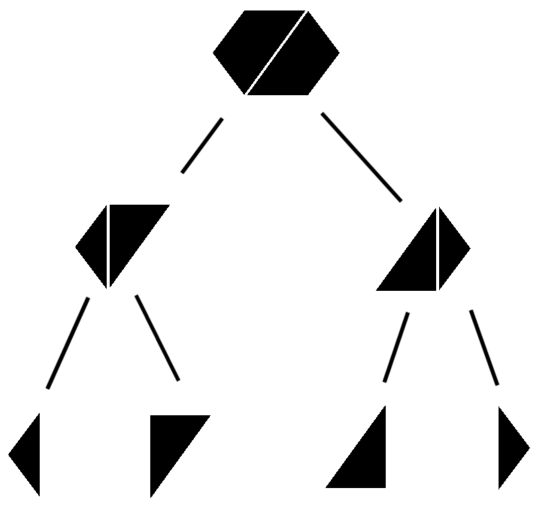


Figure 7

all the triangulatees for the cube and can move on to actually triangulating them.) Assuming such a node can be found, we follow the trail laid by the labels until we get back to the start node again (note that this can require backtracking). At each step, we follow an unused edge to an adjacent node with at least the two material IDs of the start node. If no such adjacent node exists, we back up and try another route. If one of the nodes is the cube centre node, we make a note (for a reason we'll see in a minute) and carry on. Once we get back to the start node again, we've found one of the triangulatees. This is guaranteed to happen eventually because our earlier work ensured that there is a loop back to each valid start node.

Having found a node loop, we next iterate through its edges, removing from future consideration in other node loops any edge that has at least one endpoint with only two material IDs. Finally, we store the node loop for later triangulation and carry on searching for other loops. The way we will ultimately triangulate the loop depends on whether it passes through the cube centre node, which is why we made a note of that above. Loops which pass through the centre node can be triangulated using a simple fan triangulation scheme. Other loops must instead be triangulated by a divide-and-conquer method which I will refer to as Schroeder triangulation because it appears in [Schroeder92].

Eventually, then, we'll have found all the triangulatees for a particular cube and be ready to triangulate them, using whichever method is appropriate. I'll assume we all agree that fan triangulation is a relatively trivial process (if you don't agree, a quick Google search for 'triangle fan' should point you in the right direction), so I won't labour the point, but Schroeder triangulation is a different story. The difficulty it aims to solve is that each of the remaining node loops we need to triangulate is not guaranteed to lie in a plane: this makes it impossible to use the usual 2D triangulation algorithms. Instead, we solve the problem by divide-and-conquer as follows, relying on the fact that the node loops we're trying to triangulate can be projected onto a plane without self-intersecting.

First, we calculate the 'average' plane for the node loop. To do this, we average the positions of all the nodes in the loop and treat that as a point on the plane. We then imagine the loop to be fan-triangulated from that point (see Figure 6). The average plane normal is calculated as the normalized weighted average of the triangle normals (where the weights are the areas of the triangles in question):

$$\vec{n} = \frac{\sum_i area_i \vec{n}_i}{\sum_i area_i}$$

The plane can then be easily constructed by deducing the plane distance value to be **`d = n.c`**, where **`c`** is the imaginary average point at the centre of the node loop.

Having constructed the average plane, we now try and find a diagonal of the node loop (an imaginary edge joining two non-adjacent points on the loop) that divides the loop into two in such a way that the projections of the two halves onto the average plane are on opposite sides of the projection of the diagonal. We don't actually do this by projecting the points as it's unnecessary: instead, we construct a dividing plane which is perpendicular to the average plane and passes through the diagonal we've selected. We then classify all the unused points in the node loop against this plane and check that the two halves lie on opposite sides.

Assuming that we can find at least one such diagonal (if we can't, this particular triangulation process fails, but in practice that only happens with more complicated loops than you see in M3C), we now have a choice to make among the potential candidates. Any suitable criterion can in principle be used for this, but the usual one tries to keep the aspect ratio of the generated triangles as good as possible. The metric used for this is the minimum distance of a point from the diagonal, divided by the length of the diagonal.

Having chosen a diagonal, we divide the loop in two and recurse on both halves until we reach loops with only three nodes. At that point, the recursion terminates, and we return the triangle formed by those three points (see Figure 7).

At this point, we've finished our description of the basic mesh generation process. Let's now look at the implementation in a bit more detail.

Stage 1: Basic mesh generation (implementation)

For implementation purposes, the mesh generation process can essentially be divided into two pieces. The first task is to generate the mesh vertices on the cube faces, since they're shared between adjacent cubes. Each node stores its position and labels, and the indices of any adjacent nodes (mesh edges are thus stored implicitly during the process). We store the nodes themselves in a node map, which allows us to look them up either by global index or by their location in the volume, and the indices of which nodes lie on which face in a cube face table. The designs of these two data structures are key to the whole algorithm: see Listing 1 for their interfaces. (For what it's worth, my code implements the **`NodeMap`** by using a **`std::vector`** to store the actual nodes, and a `std::map` to look them up by volume location. The **`CubeFaceTable`** is implemented as a straightforward **`std::map`** internally. The abstraction proved useful, though, because it allowed me to try out several different implementations and pick the best one.)

With these data structures in place, we can now think about actually generating the nodes on the various cube faces. The way I've written it, there are two main parts to this: (a) we need to write a routine to generate the edges on an arbitrary cube face with some given labels, and (b) we need

```
template <typename Label>
class NodeMap
{
private:
  typedef Node<Label> NodeL;
public:
  enum NodeDesignator
  {
    NODE_001, // node at the midpoint of the +z
              // edge emerging from a point
    NODE_010, // node at the midpoint of the +y
              // edge emerging from a point
    NODE_011, // node at the centre of the +y+z
              // emerging from a point
    NODE_100, // node at the midpoint of the +x
              // edge emerging from a point
    NODE_101, // node at the centre of the +x+z
              // face emerging from a point
    NODE_110, // node at the centre of the +x+y
              // face emerging from a point
    NODE_111, // node at the centre of the +x+y+z
              // cube emerging from a point
  };
  NodeMap();
  NodeL& operator()(int n);
  const NodeL& operator()(int n) const;
  int find_index(int x, int y, int z,
                 NodeDesignator n);
  //...
};
class CubeFaceTable
{
public:
  enum FaceDesignator
  {
    FACE_XY,
    FACE_XZ,
    FACE_YZ
  };
  CubeFace& operator()(int x, int y, int z,
                       FaceDesignator f);
  bool has_face(int x, int y, int z,
                       FaceDesignator f) const;
};
```

Listing 1

```
template <typename Label, typename PriorityPred>
std::list<Edge> edges_on_face(Label topleft,
   Label topright, Label bottomleft,
   Label bottomright);
```

Listing 2

to map the nodes connected by these edges to their global equivalents (which we create/look up in the node map) and store the edges implicitly in the global nodes. The interface for (a) is as shown in Listing 2 (I'm happy to provide source code by email if anyone's interested – it's a bit lengthy so I won't show it here). The code for (b) is shown in Listing 3, which shows filling in the global node map and cube face table via mapping the generated local edges and points onto their global counterparts.

Having generated the global nodes we need and ensured that it's possible to look up which nodes are in a given cube, things are now looking good. The second part of the mesh generation process is now to generate triangles for each cube. As explained before, to do this we need to first determine the number of face centres needed for the cube. This is actually quite simple, because we stored the local node map for each cube face in the face table. We now simply need to check whether the MIDDLE node for each face was used or not. Given the number of face centres, we then add edges or extra nodes as necessary (as described in the previous section). Adding

```
// The local node map initially contains UNUSED
// for each node, indicating that the relevant
// node wasn't needed.
std::vector<int> localNodeMap(
   POTENTIAL_NODE_COUNT, UNUSED);

// Run through all the edges and mark the endpoints
// with USED in the node map: this indicates that
// we need to lookup the global nodes for them.
for(std::list<Edge>::const_iterator
   it=edges.begin(), iend=edges.end(); it!=iend;
   ++it)
  localNodeMap[it->u] = localNodeMap[it->v] =
USED;

// Build the mapping from local node indices to
// global coordinates.
TripleI locs[POTENTIAL_NODE_COUNT];
NodeMapL::NodeDesignator
   nodeDesignators[POTENTIAL_NODE_COUNT];
switch(faceDesignator) {
  case CubeFaceTable::FACE_XY:
    locs[TOP] = TripleI(x,y+1,z);
    nodeDesignators[TOP] = NodeMapL::NODE_100;
    //...
    break;
  //...
}

// Lookup the global node indices.
for(int i=0; i<POTENTIAL_NODE_COUNT; ++i)
  if(localNodeMap[i] == USED)
    localNodeMap[i]
       = m_nodeMap->find_index(locs[i],
       nodeDesignators[i]);

// Fill in the labels for each node.
if(localNodeMap[TOP] != UNUSED) {
  NodeL& n = (*m_nodeMap)(localNodeMap[TOP]);
  n.labels.insert(topleft);
  n.labels.insert(topright);
}
if(localNodeMap[MIDDLE] != UNUSED) {
  NodeL& n = (*m_nodeMap)(localNodeMap[MIDDLE]);
  n.labels.insert(topleft);
  n.labels.insert(topright);
  n.labels.insert(bottomleft);
  n.labels.insert(bottomright);
}
//...

// Run through the edges and replace the local node
// indices with their global equivalents.
// Update the adjacent node entries in the global
// nodes at the same time.
for(std::list<Edge>::iterator it=edges.begin(),
   iend=edges.end(); it!=iend; ++it) {
  it->u = localNodeMap[it->u];
  it->v = localNodeMap[it->v];
  NodeL& uNode = (*m_nodeMap)(it->u);
  NodeL& vNode = (*m_nodeMap)(it->v);
  uNode.adjacentNodes.insert(it->v);
  vNode.adjacentNodes.insert(it->u);
}

// Fill in the cube face in the global face table.
(*m_faceTable)(x, y, z,
    faceDesignator) = CubeFace(localNodeMap);
```

Listing 3

```
int cubeCentreIndex =
   m_nodeMap->find_index(TripleI(x,y,z),
   NodeMapL::NODE_111);
NodeL& c = (*m_nodeMap)(cubeCentreIndex);
```

Listing 4

edges just involves modifying the adjacent node sets of global nodes; adding a new node just involves finding its index in the node map and using that to retrieve it (as per Listing 4).

Having set things up, we then iteratively find all the triangulatees as described in the previous section. The first step is to create a local node map (see Listing 5: Creating a local node map which only references nodes in the current cube), since nodes in the global map refer to adjacent nodes which are not in the current cube.

Having done that, we then find triangulatees using the depth first search-style routine in Listing 6.

The only remaining step is to triangulate the results. I won't show the code for this, because it's rather lengthy and not especially difficult to construct given the description above, but if you'd like to see the code then feel free to drop me an email.

Figure 6 shows an example image.

Summary

In this article, we have seen the first stage of the mutliple material marching cubes algorithm for generating meshes from labelled volumes. As can be seen in the example image (which shows a portion of the right kidney and aorta), the output is a bit crude at this stage (and note that we're only able to render a smaller mesh), but the results are still quite good. In the next article, I'll explain the remainder of the algorithm, namely how to smooth and decimate the results to make the rendering of larger, nicer-looking meshes possible. ■

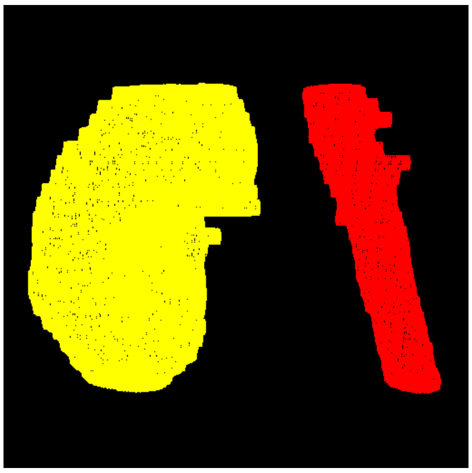


Figure 6

```
std::map<int,NodeL> localNodeMap;
for(std::set<int>::const_iterator
   it=nodeSet.begin(), iend=nodeSet.end();
it!=iend; ++it) {
  std::map<int,NodeL>::iterator loc =
     localNodeMap.insert(std::make_pair(*it,
     (*m_nodeMap)(*it))).first;
  NodeL& n = loc->second;
  std::set<int> relevantNodes;
  std::set_intersection(n.adjacentNodes.begin(),
     n.adjacentNodes.end(), nodeSet.begin(),
     nodeSet.end(), std::inserter(relevantNodes,
     relevantNodes.begin()));
  n.adjacentNodes = relevantNodes;
}
```

Listing 5

References

[Golodetz08] Golodetz, SM, 'Watersheds and Waterfalls' (Parts 1 and 2), *Overload 83/84*, February/April 2008.

[Schroeder92] Schroeder, WJ, et al., *Decimation of Triangle Meshes*, 1992.

[Wu03] Wu, Z, and Sullivan Jr., JM, 'Multiple material marching cubes algorithm', *International Journal for Numerical Methods in Engineering*, 2003.

```
Triangulatee_Ptr
find_triangulatee(std::map<int,NodeL>&
localNodeMap, int cubeCentreIndex)
{
  // Step 1: Find a start node with exactly two
  // material IDs and a remaining edge. If no such
  // node exists, we've found all the loops.
  int startIndex = -1;
  for(std::map<int,NodeL>::const_iterator
     it=localNodeMap.begin(),
     iend=localNodeMap.end(); it!=iend; ++it)
  {
    const NodeL& n = it->second;
    if(n.labels.size() == 2 &&
       !n.adjacentNodes.empty())
    {
      startIndex = it->first;
      break;
    }
  }
  if(startIndex == -1) return Triangulatee_Ptr();
  // Step 2: Follow the trail laid by the material
  // IDs - at each step, follow an unused edge to
  // an adjacent node with at least the two
  // material IDs of the start node. If no such
  // adjacent node exists, back up and try another
  // route. If one of the nodes is the cube centre
  // node, make a note and carry on - we'll need
  // to triangulate this loop using a fan approach.
  // Terminate when we reach the start node again.
  // The way M3C works guarantees that there is a
  // loop back to each valid start node, so
  // termination is guaranteed.
  std::map<Edge,bool> used;
  for(std::map<int,NodeL>::const_iterator
     it=localNodeMap.begin(),
     iend=localNodeMap.end(); it!=iend; ++it)
  {
    const int u = it->first;
    const NodeL& n = it->second;
    for(std::set<int>::const_iterator
       jt=n.adjacentNodes.begin(),
       jend=n.adjacentNodes.end(); jt!=jend; ++jt)
    {
      const int v = *jt;
      used.insert(std::make_pair(Edge(u,v),
         false));
    }
  }
  NodeL& startNode = localNodeMap[startIndex];
  std::vector<Label> labels(
     startNode.labels.begin(),
     startNode.labels.end());
```

Listing 6

```
std::list<int> nodeLoop;
int curIndex = startIndex;
bool fanTriangulation = false;
do
{
  nodeLoop.push_back(curIndex);
  if(curIndex == cubeCentreIndex)
     fanTriangulation = true;
  NodeL& curNode = localNodeMap[curIndex];
  int adjIndex = -1;
  for(std::set<int>::const_iterator
     it=curNode.adjacentNodes.begin(),
     iend=curNode.adjacentNodes.end();
     it!=iend; ++it)
  {
    NodeL& adjNode = localNodeMap[*it];
    // If the edge has not yet been used, and the
    // adjacent node has at least the two labels
    // of the start node, traverse the edge.
    if(!used[Edge(curIndex, *it)] &&
       adjNode.labels.find(labels[0]) !=
       adjNode.labels.end() &&
       adjNode.labels.find(labels[1]) !=
       adjNode.labels.end())
    {
      adjIndex = *it;
      break;
    }
  }

  if(adjIndex != -1)
  {
    used[Edge(curIndex, adjIndex)] = true;
    curIndex = adjIndex;
  }
  else
  {
    // If we couldn't find an adjacent node
    // with the right material IDs, backtrack
    // and try another route. Note that there's
    // no danger of setting the current index
    // back to the start index here: the
    // first step will always be a valid one.
    nodeLoop.pop_back();

    if(!nodeLoop.empty())
    {
      curIndex = nodeLoop.back();
      nodeLoop.pop_back();
    }
    else
    {
      throw Exception("Something went wrong:
         couldn't find an adjacent node with the
         right material IDs.");
    }
  }
}

while(curIndex != startIndex);
// Step 3: Remove edges from further
// consideration in future loops if at least
// one of their endpoints has only two
// material IDs.
std::vector<int> nodeLoopArray(nodeLoop.begin(),
   nodeLoop.end());
int nodeCount =
   static_cast<int>(nodeLoopArray.size());
```

Listing 6 (cont'd)

Cititec

www.cititec.com

Greenfield Software Development Careers in Financial Markets

Cititec are specialist and boutique global suppliers to a large number of Investment banks as well as many smaller financial institutions.

Stefano Cicu specialises in placing C++, C# and Java developers across a range of business sectors and he is always interested in speaking to fellow ACCU members and introducing them to compelling opportunitites.

We have a variety of vacancies ranging from Tactical / RAD Development through to huge global strategic projects using emerging technologies.

Stefano can be contacted to discuss opportunities or your recruitment needs if you are hiring at **Stefano.cicu@cititec.com** or you can call him directly on **0207 608 5879**

atsco INVESTOR IN PEOPLE Recruitment & Employment Confederation

```
  for(int i=0; i<nodeCount; ++i)
  {
    int j = (i+1)%nodeCount;
    int curIndex = nodeLoopArray[i];
    int adjIndex = nodeLoopArray[j];
    NodeL& curNode = localNodeMap[curIndex];
    NodeL& adjNode = localNodeMap[adjIndex];
    // Remove the edge iff one of its endpoints
    // has only two material IDs.
    if(curNode.labels.size() == 2 ||
       adjNode.labels.size() == 2)
    {
      curNode.adjacentNodes.erase(adjIndex);
      adjNode.adjacentNodes.erase(curIndex);
    }
  }

  // Step 4: Construct the triangulatee according
  // to whether or not the 'fan' flag was set in
  // Step 2.
  if(fanTriangulation)
  {
    return Triangulatee_Ptr(new FanTriangulateeL(
       nodeLoop, cubeCentreIndex));
  }
  else
  {
    return Triangulatee_Ptr(
       new SchroederTriangulateeL(nodeLoop,
       m_nodeMap->retrieve_nodes()));
  }
}
```

Listing 6 (cont'd)

DynamicAny (Part 2)

Alex Fabijanic uncovers the internals of DynamicAny with some performance and size tests.

In the first installment of this article, **`Poco::DynamicAny`** class was presented, along with rationale for it as well as practical usage examples. The convenient advantages, such as direct dynamically typed mapping from an external data source to the program storage were described. Like **`boost::any`**, **`DynamicAny`** readily provides storage and value extraction of an arbitrary user defined data type. The most common data types conversions are supported out-of-the-box through specializations provided by POCO framework, while the ones for user defined types can be added through value holder template specialization. In this installment, we delve into the internals of the class and run some tests to compare **`DynamicAny`** to similar C++ data type conversion facilities.

Performance

It is well known that there is no such thing as a free lunch. **`DynamicAny`** shall clearly pay a hefty price in CPU cycle currency for its flexibility and safety. But how well does **`DynamicAny`** perform compared to bare-bone static C++ casts and other dynamic typing solutions?

Two types of tests were performed:

- conversion (**`Int32`** in, **`double`** out; **`Int32`** in, **`Uint16`** out; **`std::string`** in, **`double`** out)
- extraction (**`double`** in, **`double`** out; **`std::string`** in, **`std::string`** out)

Various **`Any`** extractions are compared with the **`DynamicAny`** extraction. As described in the first part of the article, **`Any`** is not capable of doing conversions. For conversions, we are comparing **`DynamicAny`** with **`boost::lexical_cast`**. The test code is shown in Listing 1.

Tests have been executed on different platforms, with results shown in Table 1 as well as Figure 1a and 1b (Windows) and 2a and 2b (Linux)[1]. It is worth mentioning that the relative performance comparison between different implementations of comparable functionalities is what we were after here, not the absolute values comparison between the two sets of test results (i.e. the two platforms). To get measurable results, the tested code must be called multiple times. Since compiler optimizations undoubtedly count in the real world, it was desirable to obtain the results reflecting the optimization benefits. At the same time, loop optimization could cause results to be misleading. In order to reconcile the opposing forces, executables were compiled at a reasonable level of optimization[2], with actual conversion code placed in functions (see Listing 2.) located in a

1 Black bars represent **`DynamicAny`**, grey bars represent **`Any`**/**`lexical_cast`** results; shorter bar means better performance.
2 /02 for MSVC++, -02 for G++

Aleksandar Fabijanic Alex is a C++ and POCO enthusiast. He is using POCO at work for industrial automation and process control software development. Alex spends a lot of his free time contributing, supporting and managing the project. Contact him at alex@appinf.us

```
// Static cast Int32 to double
Int32 i = 0;
double d;
Stopwatch sw; sw.start();
do { staticCastInt32ToDouble(d, i); }
while (++i < count); sw.stop();
print("static_cast<double>(Int32)",
sw.elapsed());

Any a = 1.0; i = 0; sw.start();
do { unsafeAnyCastAnyToDouble(d, a); }
while (++i < count); sw.stop();
print("UnsafeAnyCast<double>(Int32)",
sw.elapsed());

// Conversion Int32 to double
i = 0; sw.start();
do { lexicalCastInt32ToDouble(d, i); }
while (++i < count); sw.stop();
print("boost::lexical_cast<double>(Int32)",
sw.elapsed());

DynamicAny da = 1;
i = 0; sw.restart();
do { convertInt32ToDouble(d, da); }
while (++i < count); sw.stop();
print("DynamicAny<Int32>::convert<double>()",
sw.elapsed());
i = 0; sw.restart();
do { assignInt32ToDouble(d, da); }
while (++i < count); sw.stop();
print("operator=(double, DynamicAny<Int32>)",
sw.elapsed());
// Conversion signed Int32 to UInt16
// …
// Conversion string to double
// …
// Extraction double
a = 1.0; i = 0; sw.restart();
do { anyCastRefDouble(d, a); }
while (++i < count); sw.stop();

i = 0; sw.restart();
do { anyCastPtrDouble(d, a); }
while (++i < count); sw.stop();

da = 1.0; i = 0; sw.restart();
do { extractDouble(d, da); }
while (++i < count); sw.stop();
// Extraction string
//
```

Listing 1

It is well known that there is **no such thing as a free lunch**

```
void staticCastInt32ToDouble(double& d, int i)
{ d = static_cast<double>(i); }

void unsafeAnyCastAnyToDouble(double& d, Any& a)
{ d = *UnsafeAnyCast<double>(&a); }

void lexicalCastInt32ToDouble(double& d, int i)
{ d = boost::lexical_cast<double>(i); }

void convertInt32ToDouble(double& d,
   DynamicAny& da)
{ d = da.convert<double>(); }

void assignInt32ToDouble(double& d,
   DynamicAny& da)
{ d = da; }

void lexicalCastInt32toUInt16(UInt16& us,
   Int32 j)
{ us = boost::lexical_cast<UInt16>(j); }

void convertInt32toUInt16(UInt16& us,
   DynamicAny& da)
{ us = da.convert<UInt16>(); }

void assignInt32toUInt16(UInt16& us,
   DynamicAny& da)
{ us = da; }

void lexicalCastStringToDouble(double& d,
   std::string& s)
{ d = boost::lexical_cast<double>(s); }

void convertStringToDouble(double& d,
   DynamicAny& ds)
{ d = ds.convert<double>(); }

void assignStringToDouble(double& d,
   DynamicAny& ds)
{ d = ds; }

void anyCastRefDouble(double& d, Any& a)
{ d = RefAnyCast<double>(a); }

void anyCastPtrDouble(double& d, Any& a)
{ d = *AnyCast<double>(&a); }

void extractDouble(double& d, DynamicAny& da)
{ d = da.extract<double>(); }
```

Listing 2

Operation	Windows	Linux
Static cast Int32 => double		
static_cast<double>(Int32)	31.25	29.11
UnsafeAnyCast<double>(Int32)	78.13	60.10
Conversion Int32 => double		
boost::lexical_cast<double>(Int32)	41187.50	14469.90
DynamicAny<Int32>::convert<double>()	78.13	76.85
operator=(double, DynamicAny<Int32>)	78.13	76.79
Conversion Int32 => unsigned short		
boost::lexical_cast<UInt16>(Int32)	33546.90	8631.60
DynamicAny<Int32>::convert<UInt16>()	218.75	84.85
operator=(UInt16, DynamicAny<Int32>)	218.75	84.85
Conversion std::string => double		
boost::lexical_cast<double>(string)	37312.50	13999.70
DynamicAny<string>::convert<double>()	6046.88	3858.34
operator=(double, DynamicAny<string>)	6031.25	3858.89
Extraction double		
RefAnyCast<double>(Any&)	171.88	131.26
AnyCast<double>(Any*)	140.63	102.21
DynamicAny::extract<double>()	171.83	98.71
Extraction string		
RefAnyCast<string>(Any&)	890.63	189.33
AnyCast<string>(Any*)	906.25	160.26
DynamicAny::extract<string>()	906.25	152.99
Loop count: 5,000,000		
Results in milliseconds		

Table 1

separate compilation unit. This arrangement ensured that all tests incur the same function call penalty, thus preserving the performance ratios, while benefiting from the tested functionality optimization improvements. As the test results demonstrate, **`DynamicAny`** performs significantly better than competition in conversion and approximately the same in extraction

```
void anyCastPtrString(std::string& s, Any& as)
{ s = *AnyCast<std::string>(&as); }

void extractString(std::string& s,
   DynamicAny& ds)
{ s = ds.extract<std::string>(); }
```

Listing 2 (cont'd)

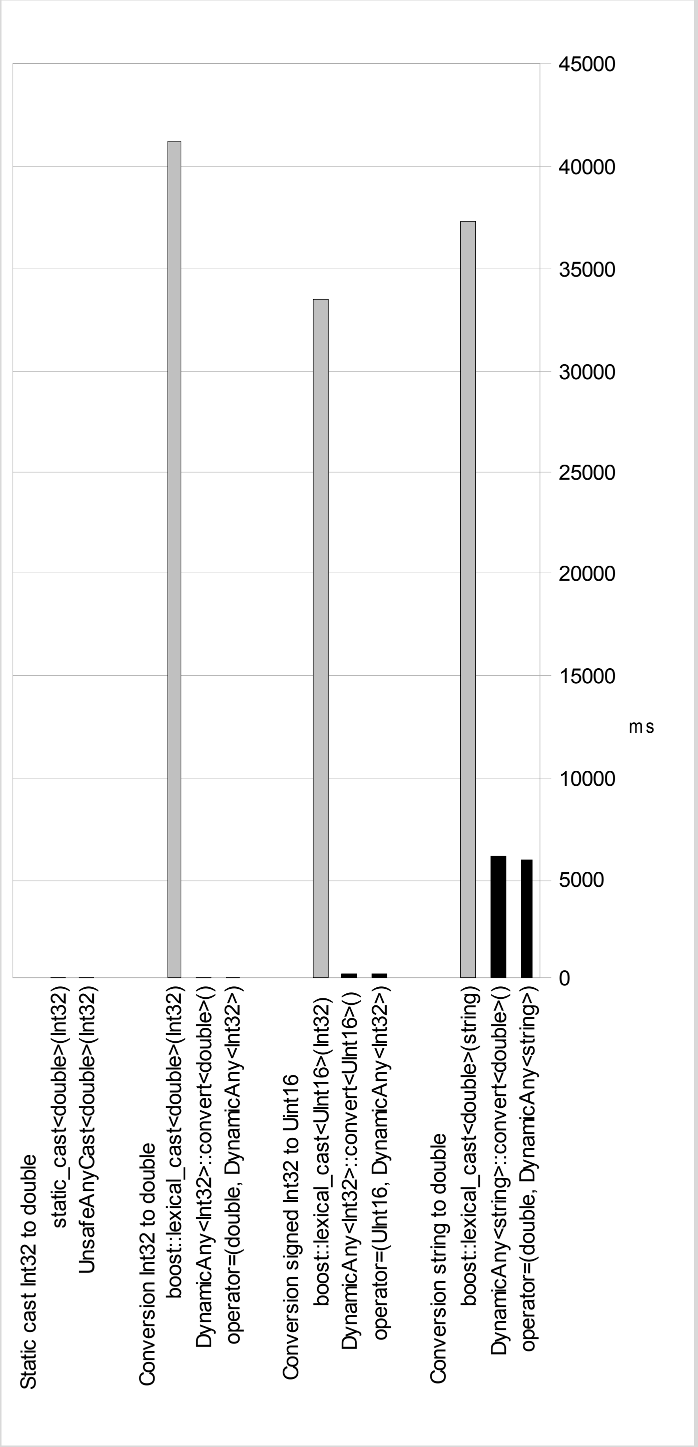


Figure 1a

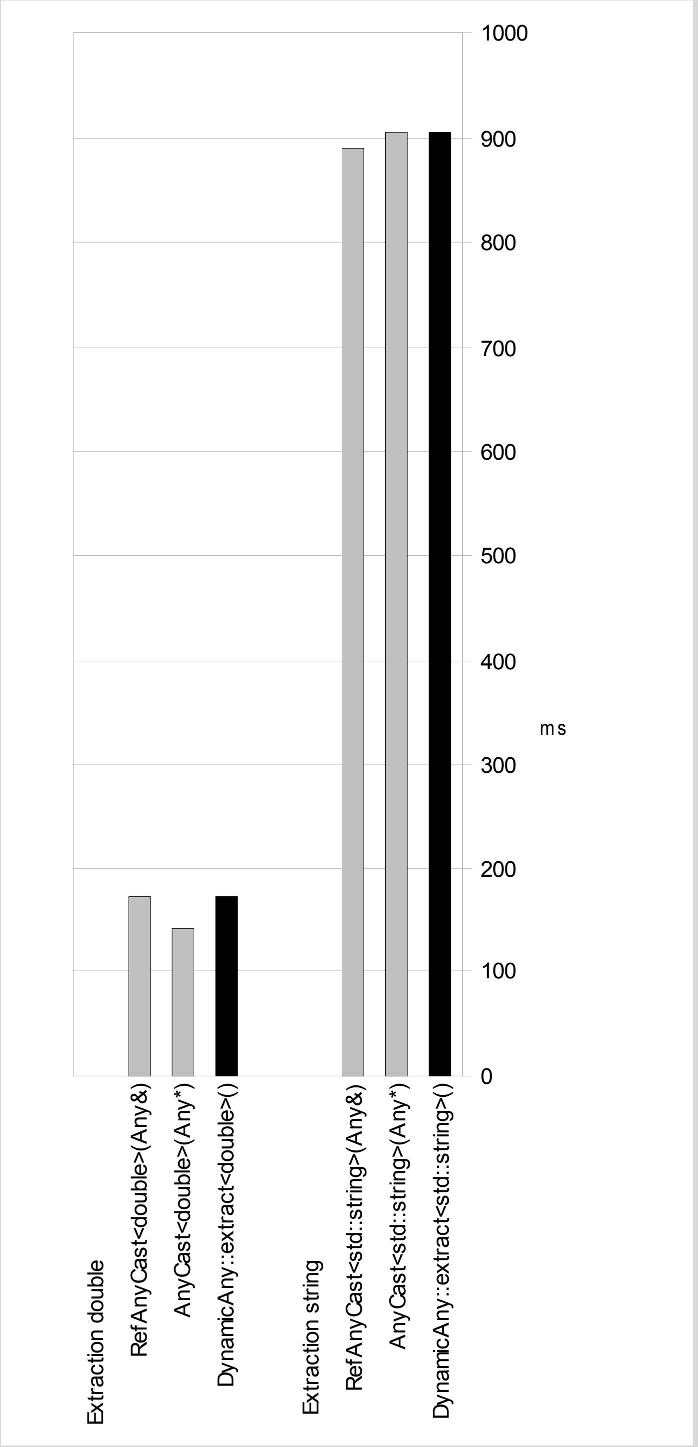


Figure 1b

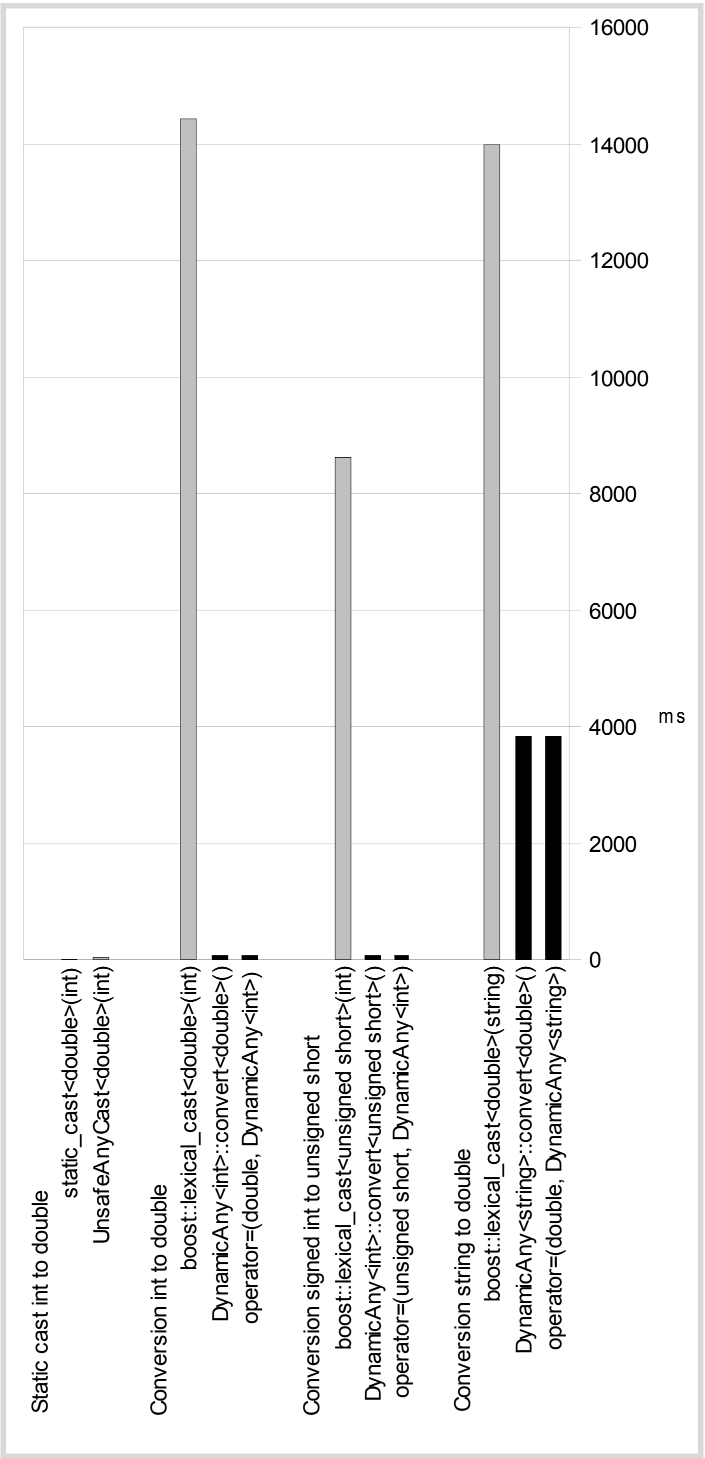


Figure 2a

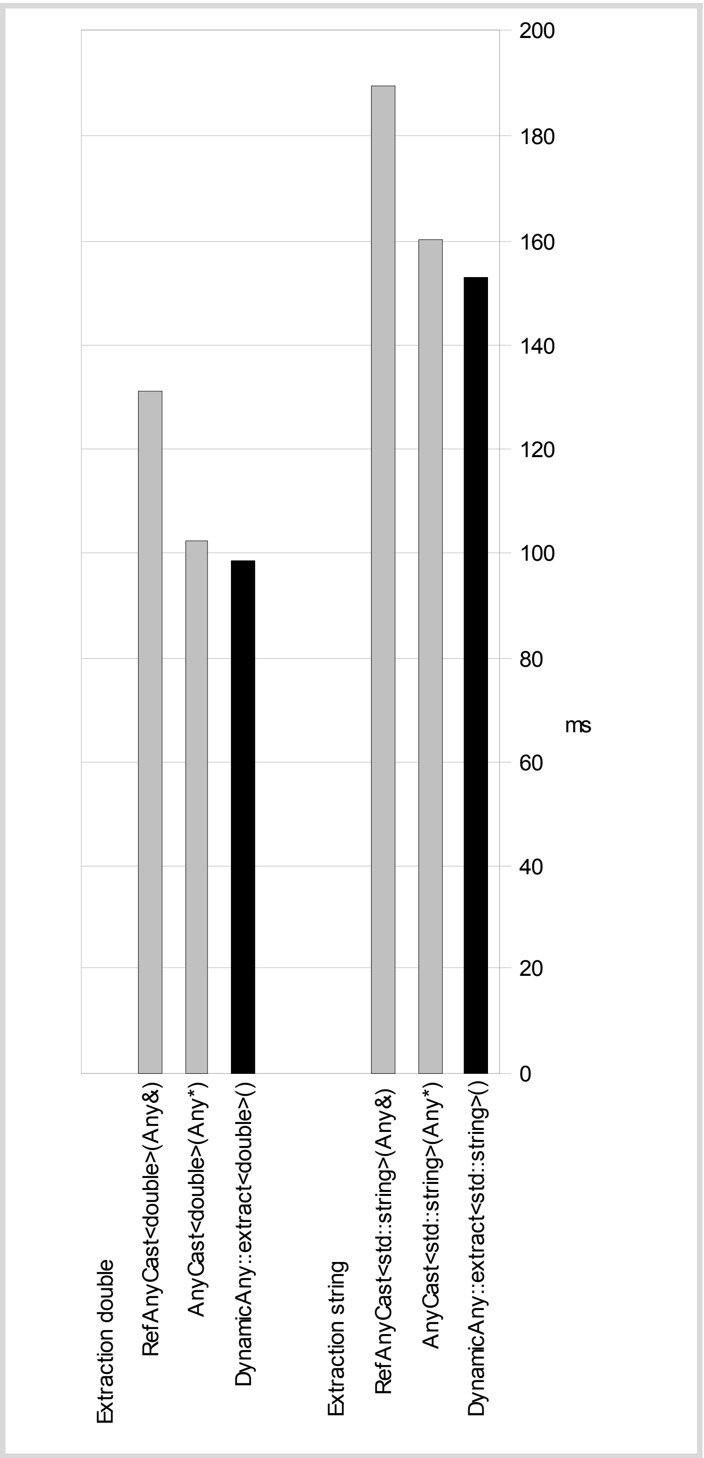


Figure 2b

Dynamic typing in C++ is a niche functionality with limited application domain

scenarios. Tests have additionally been compiled and executed on Solaris with Sun CC compiler yielding similar results. For conciseness purposes, those results are not included here but can be obtained from [DynamicAny].

It is important to mention that the performance results were obtained using the most recent development snapshot from the POCO source code repository [POCO]. The `DynamicAny::extract<>()` code used for performance testing purposes performs approximately two times faster than the code from the last release (1.3.2)[3]. The improvement was achieved by substituting the `dynamic_cast` with `typeid()` check in conjunction with `static_cast`. Additionally, the performance of `AnyCast<>()` and `RefAnyCast<>()` has been improved in the development code base through inlining.

Dynamic typing in C++ is a niche functionality with limited application domain and certainly not aimed for use in code on the high-end of the performance requirements spectrum. Nevertheless, we felt that performance is a relevant concern for `DynamicAny`. Given the database querying scenario mentioned in the first installment of the article, it is easy to envision circumstances where code performs acceptably with small data sets but performance significantly deteriorates as data sets grow. As demonstrated in performance tests, in such circumstances milliseconds rapidly accumulate into seconds or even minutes. The attention given to performance definitely yields a nice return on investment in such scenarios and provides a wider scale range available for comfortable type-agnostic coding.

Size

So far, it all went nice and well for `DynamicAny`. It provides more features than `boost::any` with no performance penalty for the overlapping ones. The conversions are significantly faster than `boost::lexical_cast`. However, when it comes to software, size definitely matters and the moment of truth is inevitably coming. What exactly is the memory overhead of this luxury and how much memory does this class hierarchy consume? Holding only a pointer, the size of `DynamicAny` on a 32-bit architecture is exactly the same as the size of `boost::any` (or `integer`, for that matter) – four bytes. Where `DynamicAny` pays its price is the code size. There is a significant amount of code doing the heavy lifting behind the scenes and it clearly shows in the size. See Listing 3a for size test source code, Listing 3b for binary sizes and Listing 3c for source code line counts. The results were obtained by compiling non-debug, statically-linked code and running SLOCCount tool [Wheeler] on relevant source code files.

Implementation internals

Based on the information provided in the first installment and the tests results, `DynamicAny` clearly champions convenience and performance in an optimal way. How was this winning combination achieved? Let's peek into `DynamicAny`'s internal implementation. At the heart of `DynamicAny` is the value holder class with virtual conversion function for each supported type. Conversions between numeric types are performed by implementation specializations in following manner:

- implicitly between 'sibling' types for widening conversions
- through `static_cast` for narrowing and signedness conversions (after series of thorough signedness and numeric limits checks)

```
// AnySize.cpp
static Poco::Any a = 1;
static int ai = *Poco::AnyCast<int>(&a);

// DynamicAnySizeExtract.cpp
static Poco::DynamicAny da = 1;
int dai = da.extract<int>();

// DynamicAnySizeConvert.cpp
static Poco::DynamicAny dac = 1;
static std::string das = dac;

// lexical_cast_size.cpp
static int lci = 1;
static std::string lcis =
boost::lexical_cast<std::string>(lci);
```

Listing 3a

```
Binary sizes:

Linux
-----
 5160 AnySize.o
23668 DynamicAnySizeExtract.o

25152 DynamicAnySizeConvert.o
 9600 lexical_cast_size.o

Windows
-------
 26,924 AnySize.obj
 96,028 DynamicAnySizeExtract.obj

103,943 DynamicAnySizeConvert.obj
 84,217 lexical_cast_size.obj
```

Listing 3b

```
Lines of code:

Any            145
DynamicAny*  3,588
lexical_cast   971
```

Listing 3c

3 Release information accurate at the time of writing the article.

The objectives were to achieve optimal **dynamic coding convenience** within limits of standard C++

```
template <>
class DynamicAnyHolderImpl<Int16>: public
DynamicAnyHolder
{
public:

// ...

  // implicit widening conversion
  void convert(Int32& val) const {
    val = _val;
  }

  // safe narrowing conversion
  void convert(Int8& val) const {
    convertToSmaller(_val, val);
  }

  // safe signed/unsigned conversion
  void convert(UInt64& val) const {
    convertSignedToUnsigned(_val, val);
  }

  // static_cast based conversion
  void convert(float& val) const {
    val = static_cast<float>(_val);
  }

  // conversion to std::string
  void convert(std::string& val) const {
    val = NumberFormatter::format(_val);
  }
// ...
};
```

Listing 4

Conversions between numeric and string values are performed by means of **`Poco::NumberFormatter`** and **`Poco::NumberParser`** classes. These classes perform the conversion by means of the **`sprintf()`** and **`sscanf()`** standard C library functions. The pair of otherwise much dreaded functions is used in a controlled and safe way with target buffers properly sized, so the security problems usually associated with those functions are not a concern [Seacord06]. The performance benefits are obvious from the comparison of the test results with **`boost::lexical_cast`** equivalent functionality.

A portion of conversion code for signed 16-bit integer specialization is shown in Listing 4.

Conclusion

During the development of POCO Data library, we wanted to provide a convenient **`RecordSet`** class capable of internally storing results and providing values without having programmer worrying about the exact data types returned and the column order thereof. The objectives were to achieve optimal dynamic coding convenience within limits of standard C++ while retaining as much efficiency as possible for a dynamic typing scenario. Additionally, conversion safety and data loss prevention were addressed as well. Through **`DynamicAny`** class hierarchy we were able to achieve the objectives. Of course, this work was possible thanks to a solid foundation being laid down by our predecessors. DynamicAny is built on **`boost::any`** foundation as well as crucially important C++ features such as C language compatibility, operator overloading and free-standing functions as interface extensions.

Readers curious about implementation and usage details are invited to download POCO from the links supplied at the end of this article. POCO is distributed under Boost license. The community features weblog, forum, mailing list and a friendly attitude toward newcomers. General interest inquiries, bug reports, patches, feature requests as well as code contributions are encouraged and very much appreciated. ■

Acknowledgements

Kevlin Henney is the originator of the idea and author of **`boost::any`** class. Kevlin has provided valuable comments on the article.

Peter Schojer has ported **`boost::any`** to POCO, implemented major portions of **`DynamicAny`** and provided valuable comments on the article.

Günter Obiltschnig has written majority of the POCO framework. Günter has provided valuable comments on the article as well as advice for test code.

Laszlo Keresztfalvi has provided valuable development and testing feedback, sample usage code as well as valuable comments on the article.

References

[DynamicAny] Article code and test results archive – http://appinf.us/poco/download/DynamicAny/DynamicAnyArticle.zip

[POCO] C++ Portable Components development repository – http://poco.svn.sourceforge.net/viewvc/poco/

[Seacord] Robert C. Seacord. *Secure Coding in C and C++*, Addison-Wesley, 2006

[Wheeler] David A. Wheeler. 'SLOCCount' – http://www.dwheeler.com/sloccount/

Further reading

Bjarne Stroustrup. *The C++ Programming Language*, Addison-Wesley, 1997

Herb Sutter. 'Modern C++ Libraries', *Proceedings, SD West 2007*

Kevlin Henney. 'Valued Conversions', *C++ Report*, July-August 2000

Boost libraries – http://www.boost.org

C++ Portable Components – http://poco.sourceforge.net

Article code repository – http://poco.svn.sourceforge.net/viewvc/poco/poco/articles/DynamicAny/

On Management: Focus, Quality, Time-boxes and Ducks

Project success depends on organisation. Allan Kelly looks at how to keep things in order.

Software development is easy. I was 12 when I started programming and I picked it up no problem. Businesses are full of people who program without even knowing it: Excel is programming by another name. To paraphrase someone at the ACCU conference a few year or two ago 'My organization bases its software on SOA – Spreadsheet Oriented Architecture.'

Yes, programming is really easy – just read *C for Dummies*. People can, and do, create small world-changing programs without ever really being taught how to program. But... software development is also one of the hardest things human kind attempts. In fact, developing good software is so complex it might be the most complex thing man has ever attempted.

Writing a small piece of software can be (but is not always) very easy. Problems set in when you want to grow the software, want more people to use it, when you want to sell it, or package it, to re-use the code or ideas, take bugs out of the system, add features, make it run 24x7x365.

It becomes hard because: more people need more co-ordination, more people need to understand what is needed, expectations rise, objectives get confused, costs are higher, money has to come from somewhere and who ever provides it expects to get a return on their investment.

Programming might be easy but managing the effort is difficult.

Ducks in a row

It is very easy to write simple software and have it do something useful. It is an order of magnitude harder to get it to commercial quality and keep doing it. And, to make it harder, there is no single accepted method for doing this and getting it right.

It sometimes amazes me that anyone ever gets this right. More amazing is that software which is fundamentally flawed works, or at least seems to work. Stuff that by any 'engineering' criteria is broken is used by companies every day and businesses depend on it.

Developing good software, delivering on time, producing with sufficient quality, etc. etc. – all the things you expect from 'professional systems' is a matter of getting your Ducks in a Row. When everything goes right the effort can hit the target – illustrated in Figure 1, a successful project.

Creating good software, meeting expectations, delivering on time and the rest is not just a matter of getting one thing right. It is a matter of getting many things 'right' – or at least workable. You can get some of these wrong and still deliver something – maybe a little late, or lacking features but you will do something.

Allan Kelly After years at the code-face Allan realised that most of the problems faced by software developers are not in the code but in the management of projects and products. He now works as a consultant and trainer to address these problems by helping teams adopt Agile methods and improve development practices and processes. He can be contacted at allan@allankelly.net and maintains a blog at http://allankelly.blogspot.net.

Ducks in a Row?

'Getting ducks in a row' is an expression (possibly American) used to describe situation were many elements or details need to be put in their correct position to ensure success. Think of the way a mother duck leads her ducklings, each duck follows the mother in a straight line, one behind the other. The ducks know whom to follow and should one deviate it is immediately clear.

In early versions of 10-pin bowling the pins were called ducks and needed to be positioned manually. Thus, setting up a game was a case of getting the ducks in a row.

Others have suggested the expression comes from the fairground airgun game of shooting metallic ducks to win prizes. Since the ducks are moving on a conveyer belt at the start of the game they need to be lined up before the game begins.

Yet another explanation of the phrase comes from shipbuilding where devices known as ducks are used to position the keel. Before building can begin all the ducks need to be in a row.

Lesson 1: On a software project there are very few things that can happen which will kill the project immediately. Failure comes through many small diversions, mistakes, errors and poor decisions. Delivering a software project is about making thousands of small decisions right rather than a few big ones.

Each time you do something badly, each time one of your ducks is out of position, you won't break the whole thing. Instead each duck out of position reduces your productivity slightly, creates a little distance from the target, or reduce quality slightly. With each duck that moves out of position it gets worse but it still works, somehow – illustrated in Figure 2.

No one duck causes the project to fail but each duck out of position increases the distance from the target. If creating 'the best software ever' means scoring 100:

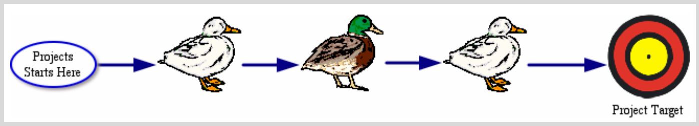


Figure 1

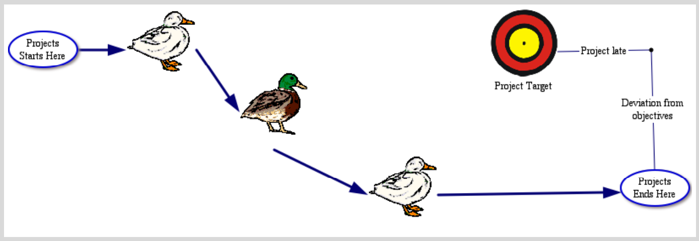


Figure 2

One of the reoccurring problems in the software industry is the willingness to pursue short-term objectives

- Lose 10 points for not co-locating the development teams
- Deduct 10 points for starting work without talking about design
- Lose 10 points for spending the first three months talking about design and drawing UML
- Work to a fixed specification or work to no specification, lose 10 points.
- ‘We don’t need no management, developers know best’ – deduct 10 points
- Fire developers half way through the project, lose 10 for lost productivity, and lose 10 for morale
- Project’s looking late, add more developers, lose 10 for invoking Brooks’ Law.
- And lose 50 if you believe ‘Developers are the problem, our consultants say we fire them and snap together Lego bricks from their toy box’

Getting the ducks in a row is the role of management. In this series of articles I hope to show how to get the ducks in a row. However every case is different so it’s hard to describe specifics without knowing the details of a project and the problems the team faces. All I can do is provide guidance and theories.

Quality

There are two aspects of quality: internal and external. External quality is that which other people see and use, it measures the degree to which the product is ‘fit for purpose’. Internal quality concerns the things most users never see – in other words the software code.

External quality is obviously important, internal quality is also important but not so obviously. Neither quality has to be so perfect that it is flawless, it only needs to be good enough that nobody has cause to question it and ask for rework.

It is acceptable for an online purchase system to lose the odd order, say one in 1000, if the owners are willing to accept that. Similarly, it is acceptable for software code to use procedural programming where object-oriented might be better, provided the code works and is maintainable.

Quality doesn’t mean gold-plating systems or over-engineering systems. To do so adds to costs without adding significantly to benefits. Provided quality is high enough not to cause unacceptable future problems then it is good enough.

When quality falls below this level problems emerge. In the case of software this means bugs and poor usability. When this happens there are two effects.

The immediate or direct effect is the problem itself. A customer finds a bug, the customer is not happy, the bug has to be reported, fixed, tested and the software reshipped. All this takes time and money.

The second, indirect effect is perhaps more damaging still: the workflow is disrupted. Instead of doing new work, managers, developers and testers have to devote time to rework. Either the usual routine is disrupted or extra resources must be kept ready to fix problems.

Lesson 2: Rework costs far more than most people realise. Many of the costs are hidden.

One of the reoccurring problems in the software industry is the willingness to pursue short-term objectives – such as meeting a deadline – rather than invest in quality. The competitive environment start-up companies find themselves in sometimes allows no other choice. Accepting lower quality today stores problems for tomorrow.

Lesson 3: Accepting lower quality today stores up problems for later.

The software industry is not the first to face this problem. Nearly 30 years ago Philip Crosby identified and described these problems in other industries. This led to his seminal book *Quality is Free* [Crosby80]. This lesson has been described again and again in industries such as car manufacturing [Womack91] and semi-conductor manufacturing [Reid85].

Those managing software development projects need to put a greater emphasis on quality - both internal and external – than is often the case today. The days when making a deadline meant trading quality for time are over.

Lesson 4: The software industry needs to collectively raise the standard of produce and reduce tolerance for rework.

Raising the game on quality starts with attitude: deciding on higher quality is essential. Then it moves to approach: allowing time for quality, rewarding quality work and prioritising quality over new features. Approach gives way to practices: requirement inspection, code reviews, test driven development and other techniques.

Critically, quality cannot be tested into a product. Improving quality is not about employing more Software Testers and fixing more bugs. It is about stopping bugs from happening in the first place. Again there isn’t anything new here, W. Edward Deming preached this mantra from the 1940s onwards.

When faults do occur they should be seen as the result of the production system not the individuals performing the work. Systems allow, even encourage, individuals to make mistakes. If a developer creates a bug it is because the system they are working within allowed the bug to be created.

To make the opposite assumption - the put the fault at the feet of the developers – is not a very useful approach. First it creates a blame culture which breeds fear. Neither fear nor blame is useful in the workplace. Secondly this view limits the scope to improve things, asking a developer to ‘stop writing bugs’ is unlikely to improve things.

allow the discussion to progress by focusing on **what** the customer requires and **when**

Rework is lost work

The cost of work to fix faults – and bugs – is graphically demonstrated by the contrast between General Motors (GM) and Toyota told in *The Machine that Changed the World* (Womack et al. 1991).

The researchers visited a GM car factoring in 1986 where they found the production line running at full capacity constantly. Anything that might stop the production line was to be avoided because that would mean lost production. When the car reached the end of the line it was inspected for faults that may have occurred during production.

Cars that failed inspection were put in a queue for repair. The direct costs of repairs included: the parts to replace, workers time to do the fix, storage space while the car was waiting for repair and then re-inspection (testing). There were also indirect costs because the company could not predict when a particular car would be ready to ship. The repair process did not disrupt assembly directly because it was so routine but it did require management time to administer the process and extra (specialist) workers.

Next the researches visited a Toyota factory. Here all workers looked for defects as the car was being assembled. If one was detected the whole production line was stopped, the fault was fixed and an investigation launched into how the fault had occurred. When the reason was found the cause would be fixed and the production line restarted.

At first sight Toyota might not have the throughput as GM but the savings from the repair work more than made up for this. Over time Toyota debugged their production system – rather than faulty cars – their production system actually got faster and faster. Productivity was higher than GM.

Under the systems view, as advocated by W. Edwards Deming, the next step would be to ask: What caused the developer to make the mistake? What can be done to prevent that happening again?

Lesson 5: Systems, not individuals, are responsible for bugs. Improve the software production system to improve quality.

The reward for improved quality is not just happier customers; it is less disruption in the workplace, improved product flow, happier workers and more predictable results. Or, as Philip Crosby would say: Quality is Free.

Time boxing

It is traditional in software projects to ask individuals how long they thing each task will take, aggregate the estimates – perhaps using a Gannt or PERT chart – and produce an estimated completion date. Almost anyone who has ever worked on a software project with formal project management will recognise the technique. But, estimates are just that: estimates. In the world of software engineering estimates are notoriously poor indicators of how long a piece of work will take,

There are exceptions: some individuals and organizations monitor their estimates, apply adjustment factors and as a result have reliable estimates. However these organizations are the exceptions, most struggle with estimates. For such organizations estimates are more trouble than benefit.

An alternative approach is to turn the question around. Fix the length of time available and ask What can be achieved in this time? Reduce each work item into small pieces which can deliver benefits and can fit within the time allowed.

This approach is called time boxing. People find it difficult to accurately estimate how long something will take so instead they are asked what they can fit into a given time. Then they are asked to commit doing the work in the time given.

The iterations and sprints used in Agile methods are examples of time boxing. Time boxes can be applied outside of Agile methods too. A team might decide to do a monthly release and stick to the date no matter what.

Importantly time boxes are the same length and are relatively short – one or two weeks is typical. Because work is divided into a series of time boxes of the same length people get fixed reference points to measure their progress by.

Lesson 6: People are poor at estimating how long a piece of work will take. So turn the problem around. Set the time allowed and ask: What can be achieved in this time?

One time box is followed by another. When a piece of work is large it is broken down into several smaller pieces and allocated over several time boxes. If there is any question of the work not being completed within the time box then it is broken down into several smaller pieces. Each piece of work should represent some measurable improvement in the system.

Using time boxes creates a rhythm to work. People are better able to understand what can be achieved in any set period when they always work to the same schedules. Like the rowers in a boat race, there is a predictability that allows co-ordination.

However time boxing does not answer that age old question: When will it be ready? If experience teaches anything in software development it is that this question cannot be answered with predictability.

Lesson 7: *When will it be ready* is not a useful question, and the answer is almost certainly wrong. Such questions and answers lay the foundations for disappointments.

Again the question needs to be turned on its head. This time the one question is replaced with two: When do you need it by? and What is it?

When we do this we change the nature of the conversation. The conversation changes from a confrontation between two or more parties about when some item may be available into a discussion about prioritisation and possibilities. Confrontation is replaced with negotiation.

These two questions allow the discussion to progress by focusing on what the customer requires and when. When will it be ready? is not a useful

In order to **hit a target** the team needs to **aim at the target**

question for a software developer, it tells them nothing of priorities and needs. Knowing exactly what is needed and by when is much more useful.

If the what cannot be delivered by the when, then conversation can proceed to break down the what into smaller pieces and possible compromises or interim solutions. In contrast, if the answer to When will it be ready? is not acceptable to the questioner the conversation tends to become confrontational.

Lesson 8: Schedule discussions need to be rich conversations. Understanding what the customers require and when they require it allows alternatives to be discussed and options examined.

Shared focus

Leave to one side the distractions of the work environment, office politics, the office move, the state of the kitchen. Leave aside personal distractions, painting the house, cleaning the car, your new girlfriend. Leave aside the distractions of everyday human life, this morning's late train, the football match and the new sandwich shop. Software projects need clear focus.

Software development is no longer an individual sport; large programs and systems are created and operated by teams. All members of the team need to be focused on the same thing and pulling in the same direction.

Unfortunately software projects are full of distractions. Ambiguous requirements may give different team members different understandings of what is required. Time schedules that lack credibility mean nobody knows when a product should complete let alone when it will. Debates over technical merits of technology X over technology Y mean neither is used.

Successful teams need focus. Focus removes distractions and channels all efforts in one direction. Individuals need to share objectives and work together.

In order to hit a target the team needs to aim at the target. So the team needs to know as much about the target as possible: what is it, when is it, why is it and where is the team now. Sports teams always know the score, software teams need to know where they are relative to the target.

Lesson 9: The first responsibility of any leader is to create a shared focus. Squeezing out ambiguity, wherever it is found, allows work to proceed more productively.

The larger the work effort and the longer the time scales the more difficult it is for a team to hold shared focus. In order to create focus software development teams need to shorten their time horizons and reduce the number of things they are attempting to accomplish.

Software is very abstract stuff so it can be difficult to create focus. When tasks, deadlines and constraints are made physical it becomes easier to focus on them. So a team might display the deadline on a large banner across the work area. Tasks might be written on cards or flips charts and ticked off as they are done.

Time boxing can help here too. Work is reduced into small pieces which can be focused on for a short while. Raising the quality bar is another, when everyone knows quality will not be compromised then the discussion goes away.

In developing software there are many, many, options and consequently many decisions to be made. After all, software is soft, everything is malleable. The development process is one of turning ideas into physical code, from abstraction to execution.

The temptation is to keep all options open and allow infinite flexibility in the development process. However this only increases the decisions to be made and provides more opportunities to make inconsistent decisions – to move a duck out of position.

Closing options, denying ourselves some of the tools helps reduce the decision space and the unpredictability encountered. Setting out a framework in which to operate reduces uncertainty and increases focus.

Summary

If teams are to be focused they need to know what they are trying to achieve. If work is broken down into pieces somebody needs to know which pieces are meaningful and which are waste. When work is worth doing it is worth doing well, with high quality. Doing unnecessary work poorly is not a compromise but a waste.

Project management is about delivering something, in some time frame. Project Managers are trained to report on progress, to manage a delivery and to adjust the delivery schedule to meet a time frame.

Project management is the dominant paradigm in the software development world. As such it tends to squeeze out the other aspects of management but it is not the only aspect of management in software development. Getting the ducks in a row involves other aspects of management.

Project success however should not be defined by delivering something but on delivering meaningful value to customers. Work packages should not be sliced and diced to fit a schedule, they should be sliced and diced to deliver value. The what is delivered is key to determining the when.

Future pieces in this series will look at other management aspects of software development and specifically at managing the what will be delivered. ■

References

[Crosby80] Crosby, P. B. 1980. *Quality is free: the art of making quality certain*: New American Library.

[Reid85] Reid, T.R. 1985. *Microchip Glasgow*: William Collins & Sons.

[Womack91] Womack, J.P., D.T. Jones and D. Roos. 1991. *The machine that changed the world*. New York: HarperCollins.

The Model Student: Can Chance Make Fine Things? (Part 1)

Evolution involves a random process. Richard Harris compares the mathematics to that of blind chance.

On the 28th May 2007, the Creation Museum [Creation] opened its doors to the public for the first time. Located in Kentucky USA, its purpose is to present scientific evidence in support of the thesis of Young Earth Creationism. This holds that Genesis is a literal account of the creation of the universe, that the Earth was created in six days between six and ten thousand years ago, that fossil beds were laid down during the flood and that human evolution is unsubstantiated religious doctrine. They propose that an international conspiracy of politically motivated secular humanists have duped the people of the world into accepting their faith as scientific truth [MatriscianaOakland91].

I must say that I find the claims of an international conspiracy somewhat surprising; I've certainly never been invited to contribute. That said, being a secular humanist unmotivated to the verge of apathy, I'd probably be the last person they'd want to enlist.

I can understand the doubt many creationists have that evolution can account for the rich array of complex life we find surrounding us. After all, it's fundamentally a random process and life exhibits a remarkable degree of organisation.

William Paley [Paley02] famously expressed this doubt by pointing out that if we were to come across a watch lying upon the ground, we should inevitably deduce the existence of a watchmaker. Richard Dawkins rejoinders this in his book *The Blind Watchmaker* [Dawkins86] with the familiar argument that it is the accumulation of small beneficial random changes rather than single large random changes that account for the emergence of order.

Unsurprisingly, my first instinct is to construct a simple model to investigate this argument.

I propose that we model iterated improvement by measuring how long it takes to randomly count from zero to a given threshold, *t*, in the range [0, n). I use the term randomly counting to describe subtracting one, doing nothing or adding one with equal probability and choosing the better of the original and the changed value 90% of the time. In other words we have six possible outcomes, as illustrated in Figure 1.

For comparison we will randomly select numbers from the range and measure how long it takes before we have one greater than or equal to the threshold.

There will be no need to run any simulations since the model is so simple that a comprehensive probabilistic analysis is relatively straightforward.

Let's start with the second of these; the number of times we need to randomly select numbers before we reach the threshold.

There are n-t numbers greater than or equal to t in the range. The probability of picking one of these is therefore

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.

$$\frac{n-t}{n}$$

The expected number of steps before we reach the threshold is simply the reciprocal of this probability.

$$E_r = \frac{n}{n-1}$$

To calculate the number of steps that iterated improvement requires, we should note that the expected increase from each step can be found by multiplying the value of each step by the probability that we take it and summing them. For simplicity, we'll allow the value to step below zero, so that every step is the same.

The expected number of steps is then the threshold divided by the expected step increase.

$$\begin{aligned} E_i &= \frac{1}{1 \times \frac{1}{3} \times \frac{9}{10} - 1 \times \frac{1}{3} \times \frac{1}{10}} \\ &= \frac{30t}{9-1} \\ &= \frac{15}{4}t \end{aligned}$$

This doesn't look good for iterated improvement. It will only take fewer steps than the totally random scheme if

$$\frac{15}{4}t < \frac{n}{n-t}$$

Move	Select	Probability
+1	accept	$\frac{1}{3} \times \frac{9}{10}$
+1	reject	$\frac{1}{3} \times \frac{1}{10}$

Move	Select	Probability
-1	accept	$\frac{1}{3} \times \frac{1}{10}$
-1	reject	$\frac{1}{3} \times \frac{9}{10}$

Move	Select	Probability
0	accept	$\frac{1}{3} \times \frac{9}{10}$
0	reject	$\frac{1}{3} \times \frac{1}{10}$

Figure 1

iterated improvement is not enough to explain why evolution should be any more effective than guesswork

Rearranging, we find that we require

$$\frac{14}{5}t(n-t) < n$$

$$\frac{14}{5}t - \frac{14}{5}t\left(\frac{1}{n}\right) < 1$$

$$\frac{14}{5}t < \frac{14}{5}t\left(\frac{1}{n}\right) + 1$$

Now, since *t* is strictly less than *n*, the right hand is smaller than the left for most choices of *n* and *t*, and hence iterated improvement is generally going to be even worse than random sampling. To make this absolutely clear let's pick some specific values, say one hundred for *n* and fifty for *t*.

$$\frac{15}{4} \times 50 < \frac{15}{4} \times 50\left(\frac{50}{100}\right) + 1$$

$$\frac{750}{4} < \frac{750}{8} + 1$$

$$1500 < 758$$

I think that this demonstrates the point; iterated improvement is not enough to explain why evolution should be any more effective than guesswork.

So does this weaken the argument for evolution?

Well, no. It does highlight a significant weakness in my model however.

A better model would reflect the fact that evolution operates upon many different attributes simultaneously. We can extend our model to account for this by using a *d* dimensional vector of integers instead of a single integer.

This time each step will randomly change the value of every element and we will consider the result to be a retrograde step if any of the elements are smaller than they were.

Given that the each element is changed independently of the others, the expected number of random guesses we need to take before every element is greater than the threshold is simply the product of the expected number of guesses needed for each of them.

$$E_r = \left(\frac{n}{n-t}\right)^d$$

With iterated improvement the calculation is slightly more complicated. We can still exploit the independence of the elements since every element will have the same expected increase at each step.

If the given element takes a positive step, it will be considered a success only if all of the remaining elements do not take a negative step. Once again we can exploit the independence of the elements by simply multiplying the probabilities that each of them is not negative.

$$p(success) = \left(\frac{2}{3}\right)^{d-1}$$

The probability of any of the remaining elements taking a retrograde step is simply one minus this.

$$p(failure) = 1 - \left(\frac{2}{3}\right)^{d-1}$$

The probability that an element takes a positive step is therefore

$$\begin{aligned} p(+1) &= \frac{1}{3}\left(p(success) \times \frac{9}{10} + p(failure) \times \frac{1}{10}\right) \\ &= \frac{1}{3}\left(\left(\frac{2}{3}\right)^{d-1} \times \frac{9}{10} + \left(1 - \left(\frac{2}{3}\right)^{d-1}\right) \times \frac{1}{10}\right) \\ &= \frac{1}{3}\left(\left(\frac{2}{3}\right)^{d-1} \times \frac{8}{10} + \frac{1}{10}\right) \\ &= \frac{1}{30}\left(8\left(\frac{2}{3}\right)^{d-1} + 1\right) \end{aligned}$$

If an element takes a negative step, the whole step is necessarily considered retrograde. The probability of taking it is therefore

$$p(-1) = \frac{1}{3} \times \frac{1}{10} = \frac{1}{30}$$

The expected increase of an element from each step is therefore

$$\begin{aligned} \delta &= 1 \times p(+1) - 1 \times p(-1) \\ &= \frac{1}{30}\left(8\left(\frac{2}{3}\right)^{d-1} + 1\right) - \frac{1}{30} \\ &= \frac{4}{15}\left(\frac{2}{3}\right)^{d-1} \end{aligned}$$

Since all of the elements change at the same rate, the expected number of steps needed for every element to reach the threshold is simply the threshold divided by δ.

$$E_i = \frac{15}{4}\left(\frac{3}{2}\right)^{d-1} t$$

So the advantage of iterated improvement is that, for a sufficiently large threshold, it scales far better than random guessing does as the number of dimensions increases. Note that sufficiently large means greater than just one third of the upper bound. Using the same values for *t* and *n* as before, Figure 2 illustrates the expected number of steps each scheme requires various numbers of dimensions.

At seventeen dimensions, iterated improvement becomes more efficient than simple random sampling. This may seem like a lot, but the threshold we chose wasn't particularly high. Figure 3 illustrates the dimensions at which iterated improvement is better than sampling for various thresholds for our *n* of one hundred.

One criticism of this as a model for evolution is that the rate of change is rather rapid and that the better candidate is selected far more often than would be expected in reality. However, if we consider each step to be the cumulative result of many generations, these choices start to look more reasonable.

evolution within a species, called microevolution, and the evolution from one species to another, called macroevolution

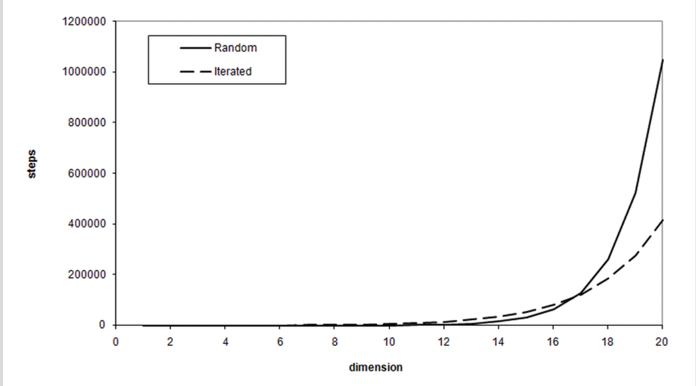


Figure 2

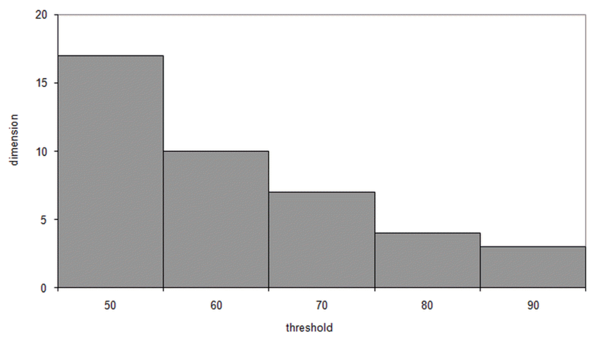


Figure 3

We can illustrate this by performing the same analysis using different probabilities of change and selection, say 1% chance of improvement, 1% chance of degradation and 51% chance of selecting the better of the two vectors. This yields an expected increase per step of

$$\delta' = \frac{2}{10,000}\left(\frac{99}{100}\right)^{d-1}$$

The proportional change in the number of steps we need to take before we have the same expected increase as before is simply the original expected increase divided by this new one

$$\begin{aligned} p &= \frac{\delta}{\delta'} \\ &= \frac{\frac{4}{15}\left(\frac{2}{3}\right)^{d-1}}{\frac{2}{10,000}\left(\frac{99}{100}\right)^{d-1}} \\ &= \frac{40,000}{30}\left(\frac{200}{297}\right)^{d-1} \\ &= 1,667\left(\frac{2}{3}\right)^{d-1} \end{aligned}$$

Whilst this may seem like a large number, it reduces rather quickly as the number of dimensions increases. Furthermore, even the worst factor of approximately 1,667 would result in one aggregate step in as little as 40 weeks for bacteria with a reproduction rate of one generation per hour.

The stated position of Answers in Genesis [Genesis], the owners of the Creation Museum, is that evolution is true, but in a limited form. They make the distinction between evolution within a species, called microevolution, and the evolution from one species to another, called macroevolution. They accept that the former is true, it can be observed in the adaptation of bacteria to antibiotics for example, but reject the latter claiming that nobody has ever seen one species evolve into another.

One of the arguments they use to refute the suggestion that given sufficient time a large number of small changes will accumulate into a large change is the assertion that evolution can only remove information, not create it.

In this light, it could be argued that my model presupposes the goal, that the answer I seek is encoded into the very vector that is subjected to iterated improvement; counting is, after all, what integers are *for*.

To investigate this, we'll need some code with which to run experiments.

To begin with we'll need a source of random numbers. The one we used in our last project will be fine and is reproduced in Listing 1.

Next, we'll need a class to represent one of our hypothetical creatures. In a fit of originality, I've called it **`individual`** and its definition is given in Listing 2.

The thus far undefined **`biochem`** class is an interface class that we will use to abstract the details of the model, allowing us to fiddle with them without rewriting all of our code. Before discussing the individual class further it's probably worth taking a look at the definition of the **`biochem`** class, given in Listing 3.

We represent the genetic makeup of our model individuals with the **`genome_type`** defined here and their features with the **`phenome_type`**. These terms, like many I shall use in this project, originate from the field of biology. The important thing to note about them is that the genetic

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

Listing 1

```
namespace evolve
{
  class individual
  {
  public:
    typedef biochem::genome_type  genome_type;
    typedef biochem::phenome_type phenome_type;
    individual();
    explicit individual(const biochem &b);
    const individual & operator=(
       const individual &other);
    void                reset();
    const genome_type &  genome() const;
    const phenome_type & phenome() const;
    void swap(individual &other);
    void reproduce(individual &offspring) const;
  private:
    void develop();
    void mutate();
    const biochem * biochem_;
    genome_type     genome_;
      phenome_type    phenome_;
  };
}
```

Listing 2

makeup is discrete in nature (i.e. integer valued) whereas the features are continuous (i.e. floating point valued).

The **`genome_size`** and **`phenome_size`** member functions determine the size of the two vectors. The **`genome_base`** member function determines the range of integers that constitute the elements of the genome. The **`develop`** member function maps a particular genetic makeup to a set of features. Finally, the **`p_select`** member function indicates the probability that we select the better of two individuals; remember that this was 0.9 in our original analysis.

In addition to managing the representation of the genome and phenome, the **`individual`** class is responsible for the reproductive process. As in our earlier analysis this involves a random change to a single element of the genetic makeup.

Before we can do this, however, we must initialise the **`individual`** so let's begin by taking a look at the constructors (Listing 4).

An interesting point to note is that whilst we pass the **`biochem`** by reference we store it by pointer. This is because we will want to store **`individual`**s in standard containers and they require contained classes to be both default initialisable and assignable. These are tricky properties to support for classes with member references since, unlike pointers, they cannot be re-bound.

```
namespace evolve
{
  class biochem
  {
  public:
    typedef
       std::vector<unsigned long> genome_type;
    typedef
       std::vector<double>        phenome_type;
    virtual size_t genome_base() const = 0;
    virtual size_t genome_size() const = 0;
    virtual size_t phenome_size() const = 0;
    virtual double p_select() const = 0;
    virtual void develop(
       const genome_type &genome,
       phenome_type &phenome) const = 0;
  };
}
```

Listing 3

```
evolve::individual::individual() : biochem_(0)
{
}
evolve::individual::individual(
   const biochem &biochem) : biochem_(&biochem),
   genome_(biochem.genome_size(), 0)
{
  develop();
}
```

Listing 4

```
void
evolve::individual::reset()
{
  if(biochem_->genome_base()<2)
     throw std::logic_error("");
  genome_type::iterator first = genome_.begin();
  genome_type::iterator last  = genome_.end();
  while(first!=last)
  {
    *first++ = (unsigned long)rnd(double(
       biochem_->genome_base()));
  }
  develop();
}
```

Listing 5

```
void
evolve::individual::develop()
{
  if(biochem_==0) throw std::logic_error("");
  biochem_->develop(genome_, phenome_);
}
```

Listing 6

Note that by default, we initialise the genome with **`0`**. If we want to randomly initialise the **`individual`**, we need to call the **`reset`** member function, illustrated in Listing 5.

The **`reset`** function is pretty straightforward; it simply sets each element of the genome to a random number picked from zero up to, but not including, **`genome_base`**.

Like the constructor, it ends with a call to the **`develop`** member function. This is responsible for mapping the genome to the phenome and is shown in Listing 6.

This simply forwards to the **`biochemistry`** object which will initialise the phenome according to whatever plan we wish.

Now we're ready to take a look at the **`reproduce`** member function. As Listing 7 illustrates, this simply copies the **`individual`** to an offspring and subsequently **`mutate`**s and **`develop`**s it.

```
void
evolve::individual::reproduce(
   individual &offspring) const
{
  if(biochem_==0) throw std::logic_error("");
  offspring.biochem_ = biochem_;
  offspring.genome_.resize(genome_.size());
  std::copy(genome_.begin(), genome_.end(),
            offspring.genome_.begin());
  offspring.mutate();
  offspring.develop();
}
```

Listing 7

```
void
evolve::individual::mutate()
{
  if(biochem_==0) throw std::logic_error("");
  if(biochem_->genome_base()<2)
     throw std::logic_error("");
  size_t element =
     size_t(rnd(double(genome_.size())));
  size_t change  =
     biochem_->genome_base()+size_t(rnd(3.0))-1;
  genome_[element] += change;
  genome_[element] %= biochem_->genome_base();
}
```

Listing 8

Note that we use **resize** and **copy** rather than assignment since for most of our simulation the **offspring** will already be the correct size, so we may as well reuse its **genome_**.

So the final member function of interest is therefore **mutate**. This simply picks a random element of the genome and, as in our earlier analysis, with equal probability increments, decrements or leaves it as it is (Listing 8).

We decide on the change we'll make to the selected element by generating a random unsigned integer less than three and subtracting one from it, leaving us with either minus one, zero or plus one. To ensure that we don't underflow the element if we try to decrement below zero, we add **genome_base** to the change, **d**. Since the change is ultimately made modulo **genome_base** to ensure that the changed element remains in the valid range this is cancelled out at the end of the calculation.

So now that we have a class to represent individual creatures we need a class to maintain a collection of them which, in another display of stunning originality, I will call **population** (Listing 9).

Whilst the basic purpose of the **population** class is to maintain a set of **individual**s, it is also responsible for managing the steps of the simulation. It does this with the **generation** member function which controls the reproduction and selection of **individual**s.

We initialise the **population** class with the constructor (Listing 10).

Note that, unlike for **individual**, we store the **biochem** by reference. This is because we will not need to store **population** objects in standard containers.

The constructor initialises the **population_** member with **n biochem_** initialised **individual**s and the **offspring_** member with **2*n** uninitialised **individual**s. The former represents the population upon which we will perform our simulation, whilst the latter provides a buffer for the reproduction and selection process.

Finally, the constructor calls the reset member function to randomly initialise the **population** (Listing 11).

The **reset** member function simply iterates over **population_** calling **reset** to randomly initialise each member.

As mentioned above the final member function, **generation**, is responsible for running each step of the simulation and it is illustrated in Listing 12.

This simply forwards to the **reproduce** and **select** member functions. The former is responsible for reproducing each **individual** in the **population_** member into two **individual**s in the **offspring_** buffer. The latter is responsible for competitively selecting members of the **offspring_** buffer back into the **population_** (Listing 13)

```
namespace evolve
{
  class population
  {
  public:
    typedef std::vector<individual>
       population_type;
    typedef population_type::const_iterator
       const_iterator;
    population(size_t n, const biochem &b);
    size_t              size() const;
    const individual & at(size_t i) const;
    const_iterator      begin() const;
    const_iterator      end() const;
    void reset();
    void generation();
  private:
    void                reproduce();
    void                select();
    const individual & select(const individual &x,
       const individual &y) const;
    const biochem & biochem_;
    population_type population_;
    population_type offspring_;
  };
}
```

Listing 9

Again, this is a relative simple function since the reproduction of each **individual** is delegated to the **individual** itself.

The **select** member function is a little more complex (Listing 14).

The **select** member function ensures that every offspring competes for inclusion in the new population. It does this by randomly reordering them, using **std::random_shuffle**, and then having each adjacent pair compete for selection (Listing 15).

This competitive selection more or less follows the scheme we used in our initial analysis. The main difference is rather than having an offspring compete with its parent, we now have offspring competing against each other.

To this end, we simply select the better of the two **individual**s, as determined by the **pareto_compare** function, with the probability given by the **biochem_**. The **pareto_compare** function returns **true** if each element in the first iterator range is strictly less than the equivalent element in the second range.

The terminology originates in mathematics. If we take a set of points in more than one dimension, those that are not less than any of the others in the set on at least one axis are known as the Pareto optimal front and represent the set of valid trade offs between the values on each axis. For example, if we had two values representing the efficiency and power of

```
evolve::population::population(size_t n,
   const biochem &biochem) :biochem_(biochem),
   population_(n, individual(biochem_)),
   offspring_(2*n)
{
  reset();
}
```

Listing 10

```
void
evolve::population::reset()
{
  if(biochem_.genome_base()<2)
     throw std::logic_error("");
  population_type::iterator first =
     population_.begin();
  population_type::iterator last  =
     population_.end();
  while(first!=last) first++->reset();
}
```

Listing 11

```
void
evolve::population::generation()
{
  reproduce();
  select();
}
```

Listing 12

```
void
evolve::population::reproduce()
{

assert(2*population_.size()==offspring_.size());
  population_type::const_iterator first =
     population_.begin();
  population_type::const_iterator last  =
     population_.end();
  population_type::iterator out =
     offspring_.begin();
  while(first!=last)
  {
    first->reproduce(*out++);
    first++->reproduce(*out++);
  }
}
```

Listing 13

```
void
evolve::population::select()
{

assert(offspring_.size()==2*population_.size());
  population_type::iterator first =
     offspring_.begin();
  population_type::iterator last  =
     offspring_.end();
  population_type::iterator out   =
     population_.begin();
  std::random_shuffle(first, last);
  while(first!=last)
  {
    const individual &x = *first++;
    const individual &y = *first++;
    if(x.genome().size()!=y.genome().size())
    {
      throw std::logic_error("");
    }
    if(x.phenome().size()!=y.phenome().size())
    {
      throw std::logic_error("");
    }
    *out++ = select(x, y);
  }
}
```

Listing 14

n engine, we may be willing to sacrifice power for the sake of efficiency, or efficiency for the sake of power. All other things being equal, we should not compromise on both. The Pareto optimal front therefore gives us the set of acceptable compromises.

Figure 4 illustrates the Pareto optimal front for a set of points randomly distributed inside a quarter circle.

The signature of the **pareto_compare** function is inspired by **std::lexicographical_compare** and since it must therefore deal with ranges of different lengths is a little more general than we require. We deal with ranges of different lengths by hypothetically extending the shorter range with values strictly less than those in the longer range. In practice this means if we reach the end of the first range without encountering an element not less than one from the second range, the first range is considered lesser than, or in mathematical parlance dominated by, the second (Listing 16).

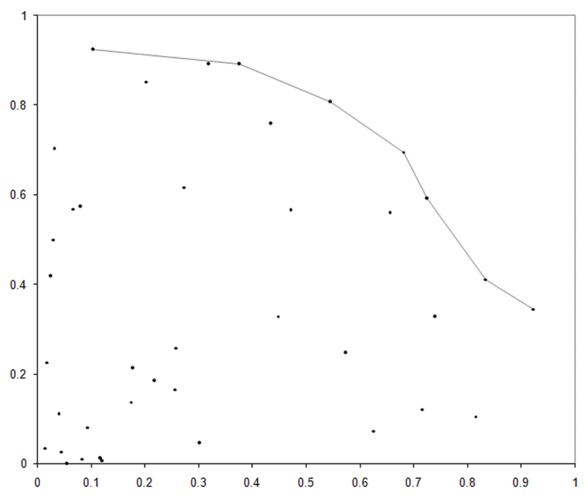


Figure 4

Given that **std::lexicographical_compare** was the inspiration for the design of this function, we might as well generalise it to cope with arbitrary comparison functions.

So, returning to the **select** member function, we ensure that we choose the best with the correct probability by setting the probability of selecting the first to **p_select** or **1-p_select** depending on whether it is the best of the pair or not. By generating a random number from zero up to, but not including, one and comparing it to this probability we can select the first exactly as often as required.

We now have all the scaffolding we need to start simulating evolution for a given **biochem**. All that's left to do is implement a **biochem** with a sufficiently complex mapping from genotype to phenotype, and we shall do that next time. ■

```
const evolve::individual &
evolve::population::select(const individual &x,
   const individual &y) const
{
  if(x.phenome().size()!=y.phenome().size())
  {
    throw std::logic_error("");
  }
  double px = 0.5;
  if(pareto_compare(y.phenome().begin(),
     y.phenome().end(), x.phenome().begin(),
     x.phenome().end()))
  {
    px = biochem_.p_select();
  }
  else if(pareto _compare(x.phenome().begin(),
     x.phenome().end(), y.phenome().begin(),
     y.phenome().end()))
  {
    px = 1.0 - biochem_.p_select();
  }
  return (rnd(1.0)<px) ? x : y;
}
```

Listing 15

Acknowledgements

With thanks to Keith Garbutt and Lee Jackson for proof reading this article.

References

[Creation] http://www.creationmuseum.org

[Dawkins86] Dawkins, R., *The Blind Watchmaker,* Longman, 1986.

[Genesis] http://www.answersingenesis.org

[MatriscianaOakland91] Matrisciana, C. and Oakland, R., *The Evolution Conspiracy*, Harvest House Publishers, 1991.

[Paley02] Paley, W., *Natural theology, or, Evidences of the existence and attributes of the Deity collected from the appearances of nature*, Printed for John Morgan by H. Maxwell, 1802.

Further reading

Frankham, R., Ballou, J. and Briscoe, D., *Introduction to Conservation Genetics*, Cambridge University Press, 2002.

Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

Hiroyasu, T. et al., *Multi-Objective Optimization of Diesel Engine Emissions and Fuel Economy using Genetic Algorithms and Phenomenological Model*, Society of Automotive Engineers, 2002.

Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

Karr, C. and Freeman, L. (Eds.), *Industrial Applications of Genetic Algorithms*, CRC, 1998.

Klockgether, J. and Schwefel, H., 'Two-Phase Nozzle and Hollow Core Jet Experiments', *Proceedings of the 11th Symposium on Engineering Aspects of Magnetohydrodynamics*, Californian Institute of Technology, 1970.

Koza, J., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

Rechenberg, I., *Cybernetic Solution Path of an Experimental Problem*, Royal Aircraft Establishment, Library Translation 1122, 1965.

Vose, M., *The Simple Genetic Algorithm: Foundations and Theory*, MIT Press, 1999.

```
namespace evolve
{
  template<class InIt1, class InIt2>
    bool pareto_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
}
template<class InIt1, class InIt2>
bool
evolve::pareto_compare(InIt1 first1, InIt1 last1,
   InIt2 first2, InIt2 last2)
{
  while(first1!=last1 && first2!=last2 &&
     *first1<*first2)
  {
    ++first1;
    ++first2;
  }
  return first1==last1;
}
```

Listing 16

```
namespace evolve
{
  template<class InIt1, class InIt2, class Pred>
     bool pareto_compare(InIt1 first1, InIt1
last1,
     InIt2 first2, InIt2 last2, Pred pred);
}
template<class InIt1, class InIt2, class Pred>
bool
evolve::pareto_compare(InIt1 first1, InIt1 last1,
   InIt2 first2, InIt2 last2, Pred pred)
{
  while(first1!=last1 && first2!=last2 &&
     pred(*first1, *first2))
  {
    ++first1;
    ++first2;
  }
  return first1==last1;
}
```

Listing 17

C++ Libraries and Tools to Simplify Your Life

> **POCO C++ Libraries:** free open source libraries for internet-age cross-platform C++ applications

> **POCO Remoting:** the easiest way to distributed objects and SOAP/WSDL Web Services in C++

> **POCO Open Service Platform:** create high performance component-based, manageable and dynamically extensible applications in C++

> and much more: Fast Infoset, WebWidgets, ...

> available on many platforms – highly portable code

> scalable from embedded to enterprise applications

appliedinformatics

Free Download and Evaluation: **appinf.com/simplify**

Performitis (Part 3)

Prevention is better than cure. Klaus Marquardt suggests some treatments to avoid problems.

Last time we discussed how to overcome Performitis, and learned about technical approaches that basically resemble what I'd call sound software engineering practices. If the teams had followed them, Performitis would never have had a real chance to emerge.

However, slips in sound practice do happen – and not necessarily by novice developers. Seasoned developers abandon solid practices because of their convictions, prejudices, and assumptions.

Process and attitude

While an extensive process cannot compensate for sloppy engineering, there are process elements that can help to keep the technical issues from becoming quality issues that endanger a release.

- A Test-Oriented Process supports early feedback and fosters a quality driven attitude. While it is not necessarily directed towards performance or against Performitis, it helps to soothe most of the symptoms. When the tests include performance criteria, both the overall design quality can be improved and the performance goals met.
- Time Boxed Releases discourages spending time on activities with low return – such as too early, too detailed performance tuning measures that can lead to Performitis.

Both process elements reduce the overall project risk and are advocated by agile development methods [Agile01]. While it is always a good idea to reduce risk, these two process elements are particularly helpful in compensating for Performitis.

When Performitis occurs, how the architect interprets his role is essential, as well as how he is perceived by the other project participants. In many teams the architect is as tangled in prejudices and assumptions as any developer. A new interpretation of the architect's responsibilities can help: if the architect is not just a technical decision maker but a coach for all developers, he needs to stand back and reflect about the project and what to coach. This Architect Also Coaches stance is much less determined to result in Performitis, both due to the reflection and to the coaching.

During this coaching or in separate workshop settings, the team can explore the Synergies Between Qualities that the project offers.

Change

The technical symptoms of Performitis can be attacked by sound practice or suppressed by extensive process. The causes of Performitis can be taught and the team led to avoid and master it.

However, sometimes sound practices do not happen, the process does not help, and all costly initiatives create more friction than effect. Let's face it, it's not the technology and not the process, it's the people. A change is needed. In this situation any organizational change is good per se – as there is no point in continuing the old way.

Projects endangered by Performitis have a size that typically results in an explicit team structure. This follows either the technical domains involved, or the feature aspects of the application domain. Projects need to cover both dimensions and fill the entire matrix that is spanned, but the organization necessarily focuses on one aspect. To Reorganize The Project Team along another dimension changes the focus of the developers and can lead to remission of Performitis.

When the team is still inclined towards Performitis, Staff Exchange is the last resort of management. Remember that Performitis thrives best in teams that close themselves against external influence. When the people cannot cure the project, the project has to cure itself.

Chance removes known paths and rites, and induces further change at a smaller scope as reaction. Bringing in experience from other cultures causes friction; the project's progress will degrade before something actually improves. Combine the therapies of change with other process and technical therapies to ensure a positive learning curve with respect to performance and intrinsic qualities.

Pattern form and approach

Both diagnoses and therapies follow their own forms, including sections that contribute to the medical metaphor.

With each diagnosis, symptoms and examination are discussed and concluded by a checklist. A description of possible pathogens and the etiology closes the diagnosis.

Each diagnosis comes with a brief explanation of applicable therapies. This includes possible therapy combinations and treatment schemes that combine several therapies. These are suggested starting points for a successful treatment of the actual situation.

Therapies are measures, processes or other medications applicable to one or several diagnoses. Their description includes problem, forces, solution, implementation hints and an example or project report. Their initial context is kept rather broad since most can be applied for different diagnoses.

In addition to the common pattern elements, therapeutic measures contain additional, optional sections of pharmaceutical information. These are introduced by symbols:

🧩	the mechanisms of a therapy and how it works;	💵	the involved roles and related costs;
⚡	counter indications, side and overdose effects;	⇄	cross effects when combined with other therapies.

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 pattern@kmarquardt.de

An overview of the therapies discussed

	Applicability	Effect	Related therapies
TEST-ORIENTED PROCESS	Preferably early in the project	Palliative; remission possible	
TIME BOXED RELEASES	Preferably early in the project	Palliative; remission possible	Combine with TEST-ORIENTED PROCESS
SYNERGIES BETWEEN QUALITIES	Any time during the project	Remission possible	
ARCHITECT ALSO COACHES (EMPHASIZE ~ILITIES)	Preferably early in the project. Whenever team composition changes	Remission possible	Requires a DEDICATED ARCHITECT
REORGANIZE THE PROJECT TEAM	Once or twice, late in the project	Side effects of change, no deterministic result	Combine with process or technical therapies
STAFF EXCHANGE	Any time during the project	Side effects of change, no deterministic result	Combine with process or technical therapies

Table 1

TEST-ORIENTED PROCESS

Applies to projects whose major implementation decisions are derived from the developers' gut feelings. No effort is made to find out whether these decisions are appropriate.

In a development team that is overly concerned about specific system properties, expensive measures are taken without evidence of necessity or possible prove of effectiveness. Important aspects of the project goals are hidden behind the self adopted blinkers.

- Experienced developers have gone through extensive learning processes,
  ...but every new project comes with problems you have not yet experienced.
- It is unlikely that somebody learns from theoretical lessons without a match in the daily practice,
  ...but each project has stakeholders whose practice differs from the developers' world and promote their particular foci.
- Changing the development methodology can be much harder than changing some technology,
  …but tackling a process issue with process actions causes the least friction.

Therefore, focus the development on testable achievements and create a test for every development step. Consider no functionality completed until it has been tested. Explicitly include things that are hard to test, like the architecture, and the system performance.

Most blinkers are based on previous experience, much like a Pavlov reflex. Typically the developers can well describe what the actual problem back then was, how it was discovered, and what they would do differently to avoid similar problems in future projects. From here, defining some kind of test is usually a small step. Messed up dependencies can be tested by evaluating generated dependency graphs; insufficient or unfavourable architectures can be tested by reviews according to self defined criteria; failed performance goals can be tested by load tests and frequent profiling against estimated thresholds. If a blinder cannot be expressed in terms of some test, it is probably just a prejudice instead of an experience.

When changing the project's way of working, make sure that the people with the greatest level of concern feel like the winners. If possible, push them into suggesting tests themselves. Never insist that the idea comes from you, and never fully expose why you were so eager to introduce tests.

TEST-ORIENTED PROCESS is effective against all diagnoses that stem from process related blinkers. Its mechanism is to relate all experiences to a common currency called 'test' – similar to business decisions that require all known influences to be converted into 'money' for comparison. This offers a global view on a lot of different aspects of development, including missing ones as well as blinding ones.

Changing the development process requires buy-in from all project stakeholders. Like all changes to the way of working, the key costs are proportional to the resistance they create. Agree on compromises that help to reduce resistance. Suggest the changes early while the team is still small, and the blinkers have not had a severe effect.

The only real counter indication against TEST-ORIENTED PROCESS would be to introduce it near the end of the project, and it is not definitely sure that it will fail without drastic measures. The side effects are the reason you introduce it in the first place: attention to previously blind spots, and time and effort spend there. The overdose effect would be to test to an extent that is no longer cost effective – but you are not very likely to experience it.

TEST-ORIENTED PROCESS goes well together with all other therapies that do not change the development process. If you need to introduce other methodology changes as well, like a focus on architecture, it might be appropriate to find the common ground among the changes first, and then change the process according to which risks are the most important ones in the current situation.

This appears to be the obvious thing for every software developer who has seen some kind of development methodology. Testing is the broadest intersection between waterfall-like and agile ways of thinking.

Management and quality control will appreciate your initiative. The project development progress becomes visible and easily traceable, while the releases inherently bring a minimal level of quality that makes formal testing much easier.

We started to introduce integration tests on a unit test granularity, for each completed function that was available on multiple platforms. These helped to limit the negative effect of changes to a common code base that were impossible to test on all systems employing the code with reasonable effort. Performance was an issue, so we expanded the test framework with time measurement. A function that had at least once caused a performance problem became accompanied by an additional unit test with no functional test criterion, but with an execution time criterion. The code had to satisfy the timing requirements of the most performance limited client, otherwise the test was not considered passed.

TIME BOXED RELEASES

Applies to projects that spend their time preparing for some future event that might never come, meanwhile neglecting sound software engineering practices.

In a development team that has lost focus of the project's purpose or the key architectural solution issues, you need to reinforce the main objectives of the project.

- Former experience leads to anticipation of problems to come,
  …but when the problems do not come, you lose the effort spent on anticipation.
- Preparing for future possibilities requires time,
  …but the time is best spent on the problems you have at hand, and on increased customer value.
- Time pressure reduces the willingness to spend effort in quality work,
  …but adequately balanced internal qualities enable the team to proceed faster, and cope with yet unforeseen situations.

Therefore, schedule the releases of your software in frequent, small intervals. Convince your management that sticking to the release dates is of major importance, just as important as keeping the internal VISIBLE QUALITIES.

When the team remains unchanged, the only variable you can negotiate is the scope, the expected functionality contained in each delivery. However, all project stakeholders will strive to have as much usable functionality with each release. This leads to a strongly perceived lack of time to care for tiny details early in the project, including early tuning measures except when well motivated as MEASUREMENT-BASED TUNING.

The mechanism of TIME BOXED RELEASES is to focus the development team, and spend only effort in those tasks that pay off within the next weeks. Implicitly, everything that has only a vague chance to pay off will not be started unless all alternatives are worse.

TIME BOXED RELEASES require management decisions and affects the entire team as well as the project's other stakeholders. Its costs are comparable to other substantial process change costs, and are not caused by the mere measures themselves. Due to reduced project risk and less effort spend in vain on irrelevant topics, it probably pays off quickly.

Counter indications to TIME BOXED RELEASES are violations to the entry conditions. Internal qualities may be hard to achieve when under time pressure, and the temptation to ignore them is high. Do not start with TIME BOXED RELEASES unless you have established some taboo on the internal qualities. Side effects are the desired emphasis changes of the projects. Some developers might feel uncomfortable with the increased time pressure, with their personal ways of reaction blocked, so prepare for some demotivation and help to establish a fearless environment. Overdose effects are reached when the time boxes are so small that your development environment does not allow for significant achievements, or when you fail to mitigate the side effect risks and induce stress and fear to individual team members.

Accompany with quality oriented process measures. It is necessary to ensure that the VISIBLE QUALITIES are established and taken seriously. A TEST-ORIENTED PROCESS helps with the necessary frequent integration, to measure progress, and to achieve internal qualities. Introduce MEASUREMENT-BASED TUNING as the standard way to motivate tuning measures.

TIME BOXED RELEASES are a two-edged sword. I have been at a team that was forced to implement features, and ignore performance for a long time. When performance had degraded to a degree that the customer could no longer reasonably execute his acceptance and usability tests, the next iteration became dedicated to performance tuning alone. TIME BOXED RELEASES are a cure for many effects, but you might experience situations where unintended scenarios emerge. In that project, we were ignoring performance issues for too long.

ARCHITECT ALSO COACHES

(Also known as: EMPHASIZE ~ILITIES)

Applies to project teams that focus their work and thoughts to a few essential ideas, but ignore all other issues that might also be or become important to the project's success.

In a development team that has an informal design and architecture process without a dedicated role assignment, and that focuses on a particular issue of development, you need to address further important system qualities that are essential to adequately manage the system architecture.

- People's experience is valuable to the project,
  …but the experience needs to fit the current project's size and criticality; building a large system requires attention to issues that hardly matter in smaller systems.
- Education opportunities for developers are readily available in courses and classes,
  …but learning as you do your job is more efficient, and has the potential to teach lessons that you never forget.
- An architect's experience and view on the software world is limited,
  …but the architect has the broadest view of the developers.
- Neglecting internal qualities can cause a large system to break under its own weight,
  …but the system can not become any better than the architect and the team know how to build it.

Therefore, the architect becomes responsible for coaching the development team about the internal qualities (the '~ilities') that are essential to crafting large software systems. This is an efficient way to succeed in his core duties – maintain the BIG PICTURE ARCHITECTURE [Marquardt02b], support management and development, balance the different forces on the architecture, ensure consistency, and broaden the architectural view – because it involves all developers and avoids resistance.

This sounds simple, but its implementation requires serious effort. While most project situations can live with a developer informally taking that role, ARCHITECT ALSO COACHES requires significant time and effort. You need to have the architect's role defined and assigned, as in DEDICATED ARCHITECT.

The internal qualities the architect needs to emphasize correspond to the size and complexity of the solution under construction. Testability is a favourite one that pays off quickly. Ensuring testability is closely related to a design with a clear distribution of responsibilities and strict adherence to dependency rules. These two are also needed to allow parallel development of several tasks and potentially several teams. Maintainability is another key quality for a large project, as during initial development the first of its components are already being maintained.

The mechanism behind ARCHITECT ALSO COACHES is respect and trust, and to spread knowledge. The team will respect an architect that knows the system, has a stake in it, and solves the day-to-day problems. Trust is necessary to learn and to change the own behavior. Spreading the knowledge helps convincing developers and avoiding resistance.

ARCHITECT ALSO COACHES involves the architect and potentially all team members. The costs can become significant because you need to dedicate a lot of time to it. However, the costs for education and consistency will reduce the project risk and likely pay off over the project's course.

If individual developers send signals that they would not accept coaching, this might be a counter indication. ARCHITECT ALSO COACHES has side effects on the work load that the team can manage. It will decrease in the short term, but eventually increase in the mid to long term.

The acceptance of the architect coaching is likely fostered when you apply ARCHITECT ALSO IMPLEMENTS [Coplien04]. Pair programming [Beck99] or JOINT DESIGN [Marquardt02] are good opportunities to start coaching. When some team members do not accept any coaching, consider STAFF EXCHANGE.

After the business case was established, the project had to change. The effort and schedule estimations demanded that the team of initially ten developers, located in two sites in different countries, had to grow to sixty within one year. While one department started growing the way management expected, the other team just grew to sixteen developers within thirty months. Most developers came straight from university, and the local architect and his manager took significant time to coach them. Years later, the project had missed the initial expectations. However, that smaller team was still working with a high quality and at a good pace. The other department had collapsed due to the mismatch between expectations and fulfilment, and most of the developers had been fired.

SYNERGIES BETWEEN QUALITIES

Applies to projects in domains that require high system responsiveness.

In a development team that is blinded by a particular experience and tends to ignore or even deny all issues that do not fall into the selected category, you need to address architectural needs that are critical to the success of the project.

- People tend to concentrate on a single issue, neglecting everything else,
  …but a broad overview uncovers connections among different system qualities.
- Some external qualities are more relevant to reach than other external or internal ones,
  …but in most cases achieving some quality does not necessarily prevent other qualities.

Therefore, identify the relation between different qualities, and separate actual contradiction from developers' superstition. Outline which technical methods and means would foster which internal and external qualities, and which would prevent you from achieving them. Initiate those actions that support multiple quality aspects of your system and that complement each other. Teach your peers about the mutual amplification of qualities, and show that many assumed or perceived contradictions are not existent in reality.

SYNERGIES BETWEEN QUALITIES works by focusing attention at the self-sustaining quality gains, and reflecting on unspoken assumptions that are blinkers to developers and managers.

All developers need to be involved, and it is helpful to include technical management as well to avoid contradicting your efforts by restrictive management policies. The costs come from the two phases of the therapy. Identifying measures and effects to qualities require some preparation that depends on the architect. The ongoing teaching is a mentoring effort that needs to last for some time; its effort is similar to other mentoring or coaching techniques.

There are no counter indications to SYNERGIES BETWEEN QUALITIES. Possible side effects or overdose effects are not attributable to SYNERGIES BETWEEN QUALITIES directly.

Combine SYNERGIES BETWEEN QUALITIES with VISIBLE QUALITIES and with ARCHITECT ALSO COACHES. PERFORMANCE-CRITICAL COMPONENTS gives some concrete examples of frequently perceived contradictions.

The therapy patterns PERFORMANCE-CRITICAL COMPONENTS and ARCHITECTURE TUNING exhibit one of the most popular virtual contradictions. If you choose a clear, concise structure, you are (a) more likely to quickly locate performance relevant issues, and (b) be able to fix them with much less effort. In the end, you get a system that runs faster and shows a better internal structure, increasing testability and maintainability.

Furthermore, performance and the qualities fostered by separation of concerns can go nicely together. However, you need to separate along the lines that help increase responsiveness - which is probably not the way you would initially decompose a system. Scalability immediate relates to system performance. Testability can also increase performance and responsiveness, both directly and indirectly by enabling layers that can be mocked for testing and later replaced with an optimized implementation. Even portability does not need to impose a performance penalty depending on the techniques the team chooses.

Some years ago I worked for a business unit that was supposed to build both a domain specific framework, and the first product based on that framework. My role was product family architect, so I was in close contact to the management and to future users of the framework. Sensing tough decisions, I asked the management for their priorities. Of the choices I offered to them, they selected two things being both on top of the list: quality, and time to market. At first, I was frustrated from not getting a clear priority. After some time I learned that this priority combination strengthened the position of the architecture a lot, and was a perfect motivation to emphasize a healthy, consistent, concise and thus respected system architecture – the best thing one could do to reach both goals.

After project failure (due to non-technical reasons), the intellectual property and most of the framework team became absorbed by other projects. While not initially intended, the architecture and the team building started to pay off. More projects became more successful than expected. Currently, its results are implicitly reused in several products and form the basic of a now successful framework.

REORGANISE THE PROJECT TEAM

Consider a project that is implemented by a team of more than a dozen developers.

A software project team that is structured into several sub-teams, the distribution of team responsibilities can only follow one possible view on the system decomposition. Each project must satisfy a number of different aspects and cover a multi dimensional decomposition.

- The organization into sub-teams enables a focused work,
  …but each project needs to have different foci, and priorities change over time.
- Different foci could be supported by having a multi dimensional team structure,
  …but reporting to different leads obtains more overhead than even most large projects can afford.
- Changes cause friction in reestablishing working teams,
  …but important goals need to be reflected in the organization to get significant attention.

Therefore, change your project organization occasionally during the project's course in a way that it reflects the highest project risk.

Dividing the project team into sub-teams according to functional components or layers is a very natural thing for architects to suggest, and can be highly effective in technical domains. Dividing the project team according to user visible function and workflow enables the team to deliver quickly what the user expects. All significant systems need to cover both views, but the organization cannot reflect both at the same time (Conway's Law, see [Coplien04]).

A deliberate change in the organization forces all project participants to think in multiple dimensions, those of the new and the former organization. The implementation and architecture follows this change with a delay, a phase shift in time. The possibility of repeated organization reversion gets the participants used to multilateral thinking and minimizes the influence of Conway's Law.

The main mechanism is change, resulting in reactions and further changes. Different aspects are addressed in the most effective way, by changing the organization.

Reorganize the Project Team requires management decision and can only be suggested by subordinates. Its costs are similar to other costs caused by change and should be seen as an insurance fee.

Reorganize the Project Team has a number of side effects including communication changes, irritation, and tighter integration. If the risks associated are higher than the chances it is counter indicated. Overdose effects occur when you reorganise too often: hidden communication due to fear, ineffectiveness due to uncertainty, and an increase in staff turnover.

An alternative team structure would be Team per Task [Coplien04] that avoids a breakup into sub-teams and forms teams for each small task. The tasks may both be technical or application bound.

Reorganize the Project Team when other therapies have failed to change the prevalent mindset. The maximum dosage is twice during the project's course. Do not apply it in the first quarter of the estimated project time.

The new team organization needs to support the area of the highest project risk. This will likely compensate for the costs and friction of the restructuring. However, be aware that initiated changes cause further changes in possibly unexpected areas, and that none of the symptoms of the actual disease is directly addressed.

The initial prototype phase ended with a team of five that did not need further substructure. When more developers joined the project team, the tasks and later the teams were split into different areas: database, GUI, and network. Further teams were established for quality and for connection to particular devices. When field tests began, the workflows slightly beyond the trivial standards failed or were unstable. To overcome this deficiency, the team focus was shifted towards making the workflow operable, and the team structure was reorganized according to the workflows. The workflow teams were composed so that each technical competence was represented.

Staff Exchange

Applies to projects stuck with old ideas that work to some extend, but lead to unsatisfying results.

In a development team that is stuck, caught within their own ideas, and blinded by their own limited experience, you need to bring in new ideas.

- People's experience is valuable to the project,
  …but repeated similar experiences can blind you and let you ignore new possibilities.
- Changing the staff of your project is risky both socially and by means of the technical and organizational learning curve,
  …but new people bring new ideas and different blinkers.

Therefore, suggest exchanging some members of the development team for developers new to the domain or the company. Make sure that management replaces at least one of the key team members, and looks for replacements that are personally able to become key players within a short time. Look for developers that bring experience, a strong personality and good communication skills, so that the project really profits from their knowledge.

Ensure that you have a stake in the job interviews. Be clear about your goals and the difficult situation during the job interview, to avoid later disappointment that could counter your intentions. Management could make the first raise depending on the influence the newcomer gains during his first months.

The mechanism of Staff Exchange is to influence the development team by a factor they cannot ignore: new colleagues. These bring new knowledge and different experience and inject this into the developers'minds while the entire team is going through all team building phases.

Staff Exchange is beyond the scope of an architect and can only be suggested to management. There is no one-fits-all answer on the related costs, but the required learning curve and the intended friction make it expensive.

A strong counter indication to Staff Exchange is when at least some of the key developers are willing to learn and accept offered opportunities. A side effect is that you run into more discussions than you really desire. Besides all irrationalities of the newly started team forming process, the arising discussions will cover development practice and methods, coding and quality standards, architectural ideas, just to mention a few. An overdose can cause the team to get lost in discussions, break its motivation, and eventually loss of the project and valuable employees. Another overdose effect could be that the company loses the expertise it once had, without being able to adequately replace it.

Staff Exchange is kind of an entropic therapy causing undirected activities. You need to complement it by problem and goal oriented therapies to focus the direction of its effect.

Before you consider exchanging the entire team, think of exchanging just the architect. An architect in the wrong place can do more harm than good. For indications of such a step, see the diagnoses in [Marquardt03].

Another important variant is to expel the consultants. Having consultants in the role of an architect is particularly dangerous as significant competency, the reasoning behind decisions, and key knowledge will be gone at a time you cannot predict. Consultants that follow their own agenda are also an obstacle.

A division of a consulting company specialized in technical projects was particularly good at taking the entire technical leadership of their customers projects. While about half the staff in the development team was new to that kind of project, a large number of experienced developers were distributed over the teams. They also came to join particular projects as senior consultants when problems occurred. The internal turn-over made sure that the knowledge was spread, and the new developers became familiar with different projects quickly while maintaining a common understanding and corporate identity with the company.

A contractor had managed to become the mind monopole at one of my customers. He motivated his queer data model with reasons about local performance gains. When the system went productive, it was a factor of 100 slower than comparable systems, which would have caused annual operation costs of several 100,000 . Confronted with radical ideas to save a factor of 1000, the contractor reacted with denial, without being able to give reasons. Due to cost saving measures, the contractor finally had to leave the project, and system responsibility was passed to an internal team. For political reasons, only the simplest of the suggested changes became implemented, and these confirmed the initial performance gain estimation.

Suggested treatment schemes

Now that the individual therapies are known in detail, it is time to think about which therapy to apply when.

While some therapies are known to be risky, there is good news about therapy combinations: no combination of the suggested therapies can be harmful to the project. They all have mutually increasing effect. However, too many therapeutic changes at the same time might break more than they heal. Take your time to introduce one after another, and anticipate which one will have the most effect in the current situation.

Figure 1 is one treatment scheme that might be useful for you if the project has been on its way for a significant time. In this case, you need to focus on therapies that can be used any time during the project. Which of these you can trigger depends on your role and your ability to convince other project participants.

First, establish MEASUREMENT-BASED TUNING – this one imposes small costs, does not require major project changes, and is the basis for further tuning measures. In case it is not accepted, you may need to wait until the problems of the project are more severe. Then you can offer a DEDICATED ARCHITECT role as an approach with a clear responsibility to ensure adequate system performance.

With the acquired data, start ARCHITECTURE TUNING on the basis of the observations and their interpretation. With this initiative you can prove that performance is taken seriously and covers most concerns.

The following steps depend strongly on whether you succeed, or at least perceive a growing acceptance. If you fail to get that acceptance, you should consider addressing a missing DEDICATED ARCHITECT very soon. In case some key developers remain ignorant, consult with management and suggest process changes that help to reduce the overall project risk, like TIME-BOXED RELEASES. Process changes possibly happen at the expense of the developers' motivation. So alternatively you could try to influence QA to demand a TEST-ORIENTED PROCESS. You need to consider that such a therapeutic schema against the team's adopted habits might lead to an unintended STAFF EXCHANGE, and possible drawbacks will be attributed to some scapegoat.

With growing acceptance, you can in parallel introduce VISIBLE QUALITIES to start a team learning process. Over time, those developers eager to learn will notice the SYNERGIES BETWEEN QUALITIES and will have learned for their professional careers. Depending on the reactions, be relaxed on the strict timing related to VISIBLE QUALITIES, it might be OK to check them with every release only.

A very short treatment scheme (Figure 2) can be recommended whenever the team composition changes. Take your time to coach new members for a while and EMPHASIZE THE ~ILITIES of system architecture.

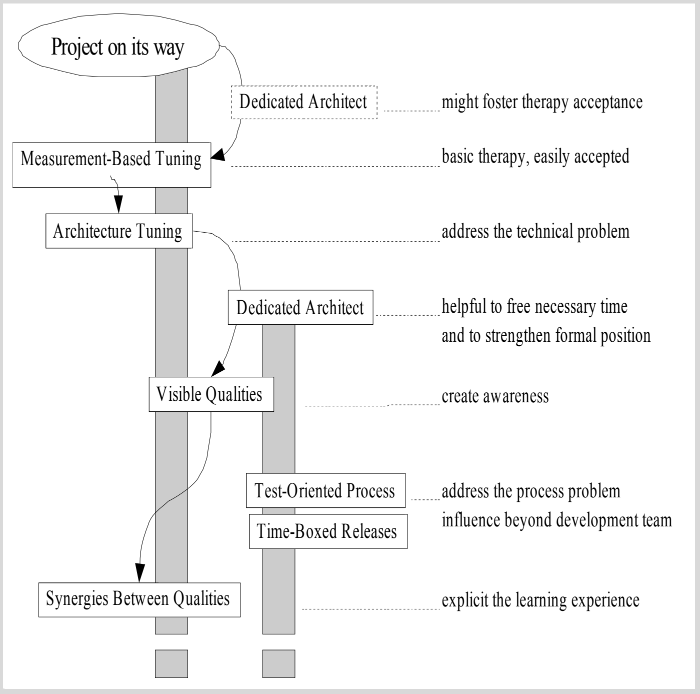

Project on its way
Dedicated Architect
might foster therapy acceptance
Measurement-Based Tuning
basic therapy, easily accepted
Architecture Tuning
address the technical problem
Dedicated Architect
helpful to free necessary time and to strengthen formal position
Visible Qualities
create awareness
Test-Oriented Process
Time-Boxed Releases
address the process problem
influence beyond development team
Synergies Between Qualities
explicit the learning experience

Figure 1

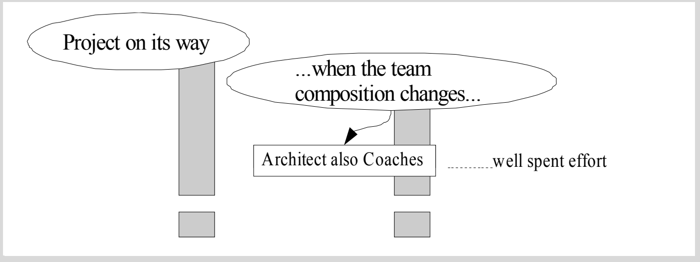


Figure 2

Annual health check

As with all diseases, the success of the healing depends on the patient, not the doctor. While the doctor applies due diligence, offers support and adequate medication, therapeutic success is the key interest of the patient and ultimately his responsibility. The healing depends not only on the quality of medication, but on compliance to the advice and on the willingness to change personal habits and attitude.

The doctor is now satisfied with your situation. Please come back next year for your annual health check! ■

Acknowledgements

Many thanks to Ric Parkin and the Overload team for their feedback and engagement.

A word amongst colleagues: Dear fellow medical architects, if you could report on your results with different therapies and patients, I'd be interested to add this to the body of knowledge. Please contact me at pattern@kmarquardt.de. Many thanks in advance!

References

Agile01 http://www.agilemanifesto.org
'The best architectures […] emerge from self-organizing teams'

Beck99 Kent Beck: E*xtreme Programming Explained: Embrace Change*. Addison-Wesley 1999

Coplien04 James Coplien, Neil Harrison: *Organizational Patterns of Agile Software Development*. Prentice-Hall 2004

Marquardt02 Klaus Marquardt: 'Supporting the Software Architect: Selected Patterns Covering Different Perspectives'. In: *Proceedings of EuroPLoP 2002*

Marquardt02b Klaus Marquardt: 'Patterns for the Practicing Software Architect'. In: *Proceedings of VikingPLoP 2002*

Marquardt03 Klaus Marquardt: 'Neglected Architecture'. In: *Proceedings of VikingPLoP 2003*