overload 88

DECEMBER 2008 £3

Generics Without Templates

Revisiting and refining a technique for generic C++ code without templates

The Legion's Revolting

An ancient Roman punishment helps to reduce excessive triangles

Understanding Who Creates Software

More On Management: asking a quite fundamental question

Iterators and Memberspaces

A C++ idiom to expose a collection's contents, with optional filtering, in a simple way

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 88

December 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 89 should be submitted by 1st January 2009 and for Overload 90 by 1st March 2009.

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

4 The Model Student: Can Chance Make Fine Things? (Part 2)
Richard Harris models the evolution of populations.

12 Model View Controller with Java Swing
Paul Grenyer redesigns his user interface.

20 On Management: Understanding Who Creates Software
Allan Kelly looks at software organisations.

24 The Legion's Revolting
Stuart Golodetz cuts his model down to size.

29 Iterators and Memberspaces
Roel Vanhout shows how to expose member data cleanly.

34 Generics without Templates - Revisited
Robert Jones re-implements parts of the STL for limited compilers.

ACCU

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

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

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

The Sweet Smell of Success

If your last project wasn't successful, you're not alone...

Apparently most software projects fail [Standish]. A scarily high statistic, but how do you decide if a project has indeed failed? What do 'fail' and 'succeed' mean?

At the most simplistic, a successful project has achieved its goals and a failing project hasn't to a greater or lesser degree. (Of course, if it doesn't have any goals then it's very hard to decide whether or not it has succeeded, but a project with no goals isn't very interesting.) So lets expand the concept of 'goals' a bit, and see how not meeting them causes failures.

The most general goal is 'deliver enough value to have been worth it'. This is still pretty vague, but it does start to produce some insights: the first thing is to deliver – a project is an obvious failure if it has never been finished, but it also won't have delivered if no one uses it, either because no users exist, or they've got it but don't use it (this latter is a bit more subtle if you sell your software – you might have got the money for the first sale, but you're unlikely to get repeat sales if the users aren't happy.)

But what of a project that is still being written? It's not yet delivering any value, so it cannot be thought of as successful or not, just 'in progress'. This is one of the problems of those over-long projects that go on for years – until the first user starts getting some value out of it, the project is just pure cost. This is one reason why early delivery of incomplete systems is such a good idea – apart from getting feedback on what you have done as soon as possible so you can adjust your future plans, the system can start earning its keep while you implement the next stage.

But what is this vague sounding word 'value'? Well, it is a measure of what is important to the person paying for the work. This could be the obvious monetary measure, but could just as easily be measured in improved performance, or just making the user's life a bit easier and making them happier with the system. But delivering a small amount of value at a large cost isn't a good trade-off, so there must be a minimum level to have made the effort worthwhile. To do that we must compare the value gained to the cost of achieving that gain.

That cost can comprise several aspects – again there's the obvious monetary cost of computers, tools and people's wages. A more subtle cost is the 'opportunity cost' – while you were implementing feature X, you have given up the opportunity to do feature Y instead. This gets really noticeable when you take the huge decision to scrap an existing system and reimplement – all the time and effort taken to redo it could have been spent enhancing the existing system with completely new features, or improving the performance and reliability. And starting from an existing system will generally mean you've an already stable, working codebase, which makes incremental improvements and delivery easier and more predictable.

"I didn't fail the test, I just found 100 ways to do it wrong" – Benjamin Franklin

So in what ways could a project fail? Remember, though, that failing is not an all-or-nothing idea, it just means that in some way it fell short of success, perhaps by only a little. A good starting point would be to consider the three classic aspects you try to manage: Features, Quality and Time.

Failing on features reduces the utility of what is delivered, and thus reduces the amount of value delivered. In project management, when something has to give, this is often the compromise of choice, especially if you make sure the vital features are done first and then you can drop the Nice-To-Haves and optional extras if things look tight.

There's a more insidious way to fail on features though – doing the wrong ones. Creating features that few or no people use is a waste of time, effort and opportunity. This is why it is vital to get a really good idea of what users actually need and their relative importance. Other problems are implementing features that you think might be useful, but no one really knows for sure. This is pretty common when you're creating something brand new, as you are having to predict based on very little evidence or experience.

Failing on the level of quality can have serious effects – buggy code can be an irritant that slows users down (so they get less value), really bad bugs can prevent the system working at all, and could even cause so much pain and effort to get to work that the net effect is negative over previous systems. Some software can be so critical that failures can cause fatalities.

Failing on time, in the form of not delivering by the promised date, is depressingly common in our industry, and I suspect accounts for most projects counted as failures. In the worst case, the project is cancelled outright, 50% to 100% overrun is not uncommon in my experience, but even delivering slightly late can have serious effects on a business, from loss of time to spend on other opportunities reducing the business' competitiveness, to contract penalty clauses, and ultimately going under because a competitor got their software out sooner.

As an aside, I suspect this effect can cause poor software to be common. Consider a market with two competitors. Company A takes six months and releases a limited and buggy product. Company B tries to do the 'right thing', and takes a year to ship a full featured, robust product. But product A has already been shipping for six months and has dominated the market, so product B sells poorly, and the company goes under. A sting in the tail is that there's no longer as large an incentive for Company A to further improve their product.

But it seems that being late is often not that serious an issue – many projects deliver late but are otherwise successes. As long as the company hasn't suffered majorly in the meantime – perhaps a previous version of the system is still selling well enough – then such delays can be absorbed.

So, what causes these failures?

Doing the wrong features can stem from poorly understanding the target market and its needs. And don't confuse what the user says they want with what they actually need – quite often a feature request is someone's idea of how they think a problem can be solved, and there can be a better solution if you can find out what is the real underlying problem. Another cause is a form of Priority Inflation – a suggestion that something might be considered is included as a Nice-To-Have, which mysteriously

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.

becomes a Priority and is then translated into a Show-Stopper. These sorts of misunderstandings could be caused by the marketing department not functioning well, having a brand new and unknown market where the customers don't even know what would work, or the developers not understanding what is needed – a lot of these are communication problems.

A hard issue to address is the natural tendency for developers to want to do something cool and new – it might well be interesting to rewrite everything in a brand new language, but can it really be justified that it is the best thing to do? Worst of all if the people suggesting such a move have the power to push it though – the hardest customer to say no to is your manager or architect who has a pet feature they want to do.

Poor quality has a myriad of causes, but the most pernicious is for the development system as a whole to not have a solid commitment to it. Even if individuals and teams want to do the right thing, it's very easy for corners to be cut especially in the face of an important deadline, or a customer offering a big order. In a sense the way that software is comparatively easy to change is a cause of this problem – quality can quickly be compromised because of a last minute 'can we just get this change in? It's a little one I promise!' Even in the best teams with the best of intentions, it's all too easy to put something in late in the development cycle with not enough time to verify that your (no doubt carefully considered) reasoning was in fact accurate and it is a safe change. Here I find Agile methods can cause problems – if your release cycle is very short then there is never enough time to flush out any subtle problems, so there's a pressure not to do anything unless it's trivial and important issues get postponed.

Being late is easy. All you have to do is promise too much and underestimate the time it takes to implement – and politically that might be done to 'sell' the project, deliberately or unconsciously. The real killer factor here is the interrelationships: the simple estimation technique I suggested last issue works well for small isolated tasks, and also extends well to combine strings of subtasks into a bigger estimate. But one thing it is easy to miss are the interactions between tasks and existing systems. If you do one thing in isolation, you have one estimate. Do two things and you have two estimates, plus one interaction that may generate a third small task to fix. Do three things, and you've three estimates, plus three interactions. And so on until the interactions dominate and you can't get anything done. This effect is why I think principles such as 'Separation Of Concerns', and 'Program to an Interface not an Implementation' work – they cause you to organise your design such that the interactions are reduced to a manageable level and things are isolated enough to get a grip on. In a similar manner, incremental implementation and delivery avoids biting off more than you can chew, and can let systems settle down, stabilise, and unforeseen problems fixed, before doing the next round.

But for all those 'failures', it is surprising just how many projects deliver something worthwhile, and are fantastic opportunities from which to learn lessons that can be applied to the next project. As Ralph Waldo Emerson put it, 'Our greatest glory is not in never failing, but in rising up every time we fail.'

Economic turmoil

While writing my last editorial I noted how things were going a little odd in the financial world, and how hopefully things would have settled down and the consequences had become clearer by the time you read it. Well, another editorial and things have got worse, with the outlook rather bleak for the next year or two (if we're lucky.) The big difference now is that this is no longer just a financial crisis, and everyday companies and people are now being affected. Otherwise healthy companies are deferring major investments in new projects, or cancelling until things become more stable; similarly with spending decisions – only buy when absolutely necessary, and so products aren't selling as well as forecast. All those nice optimistic sales forecasts are now out of date overnight, and no one has a clue what's going to happen next, and so companies understandably have to re-plan for an uncertain future.

But the personal cost of this can be great, as excellent people find themselves without a job. Having been made redundant myself in the wake of the dotcom bubble, I know how difficult it can be to know where to start sorting things out, but also how much the ACCU can be of help – just being a member looks good on a CV (and writing articles and giving talks even better!) and the networking aspects such as accu-contacts and the local meetings can really help to get the ideas and contacts that lead to that next job (and did you know there are also informal Facebook and LinkedIn groups?).

Le C++ nouveau est arrivé!

And to finish, some good news. Towards the end of September, the C++ committee met up and a major milestone happened: the draft of the new C++ standard [WG21] was published for an international ballot. This is essentially the feature-complete release, sent out to the beta testers (ie the national bodies) ready for some bugfixing. The first review is now happening, a second will happen next year and then the standard will be ratified – the intention is for it to happen towards the end of 2009, so it will indeed be C++09. Several compilers are already implementing parts – GCC has various features as options, patches and branches [GCC], Microsoft's early preview of Visual Studio 10 [Microsoft] has some too, and Codegear [CodeGear] are also working hard at the new features. Time to get experimenting!

References

[CodeGear] http://www.codegear.com/products/cppbuilder/whats-new/

[GCC] http://gcc.gnu.org/projects/cxx0x.html

[Microsoft] https://connect.microsoft.com/VisualStudio/content/content.aspx?ContentID=9790

[Standish] http://www.standishgroup.com/ produce a series of CHAOS reports into software projects. Their results fairly consistently report around 70-80% of projects fail or are challenged.

[WG21] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf

The Model Student: Can Chance Make Fine Things? (Part 2)

How well does evolution find solutions?
Richard Harris models population change.

Last time we built a library with which we will simulate the process of evolution in the hope that we can use it to gain some insight into if and how evolution works in the real world. All that was left to do was implement a biochem complex enough to act as an analogue of the relationship between the genetic makeup of a real world creature and its physical properties. What I'd really like to use would be a model of engine efficiency and power such as I used to illustrate the Pareto optimal front.

Unfortunately, such models are complicated. *Really* complicated. We're talking advanced computational fluid dynamics to even work out what happens inside a piston [Meinkel97].

So instead, I'm going use randomly generated functions with several peaks and troughs. Thankfully, it's really easy to construct such functions by adding together a set of randomly weighted constituent functions, known as basis functions. Specifically I'm going to use radial basis functions, so named because they depend only on how far an evaluated point is from a central point.

There are quite a few radial basis functions to choose from. I shall use one we have already come across; the Gaussian function. This is the probability density function of the normal distribution and has the form

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Its shape is illustrated in Figure 1.

We don't require that the function integrates to one over the entire real line, so shall replace the scaling factor with a simple weighting value. We also want to generalise to more than one dimension so must slightly change the exponentiated term. The functions we shall use will have the form

$$f_i(x) = w_i e^{-c\|x-\mu_i\|^2}$$

The $\|\boldsymbol{x}-\boldsymbol{\mu_i}\|^2$ means the square of the straight line, or Euclidean, distance between the points $\boldsymbol{x}$ and $\boldsymbol{\mu_i}$ and is easily calculated by summing the squares of the differences between each element of the vectors representing the coordinates of those points. Each function in the set will have its own weight w_i and they all share a scaling factor c applied to the squared distance. We shall restrict the arguments of our functions to values between zero and one, so I shall choose this scaling factor so that, rather arbitrarily, a distance of 0.1 corresponds to a 50% drop in the value of the function from its maximum at a distance of zero. This implies that

$$0.5 = e^{-0.01c}$$

Taking logs of both sides yields

$$\begin{aligned}-0.01c &= \ln 0.5 \approx -0.7 \\ c &= 70\end{aligned}$$

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.

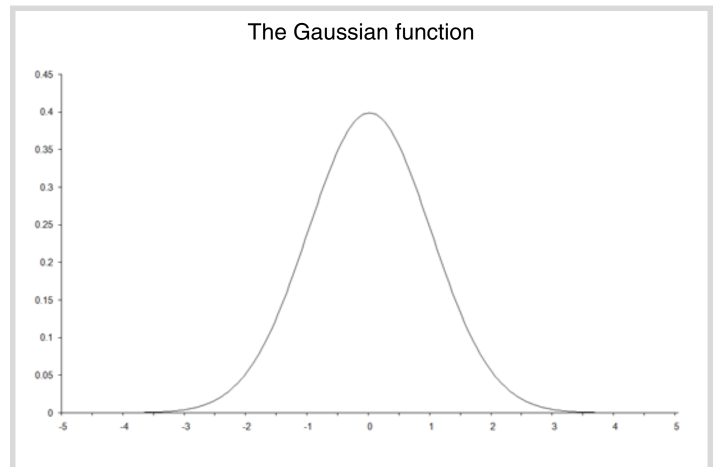


Figure 1

Figure 2 illustrates one such function, in one dimension, constructed from 8 basis functions with centres randomly chosen from the range 0.2 to 0.8 and weights randomly chosen from the range -1 to 1.

Implementing a two dimensional version of this is reasonably straightforward. Listing 1 shows the class definition for our random function.

The `node` nested class represents a single, randomly centred and weighted, basis function supporting function call semantics through `operator()`. The `random_function` class maintains a set of `node`s and sums the results of the function calls to each of them in its own `operator()`.

So let's take a look at the implementation of `node`'s member functions (Listing 2).

I'm a little ashamed to have hard coded the ranges of the central points and weights. However, we're only going to use it during our simulations, so I think it's probably forgivable in this instance.

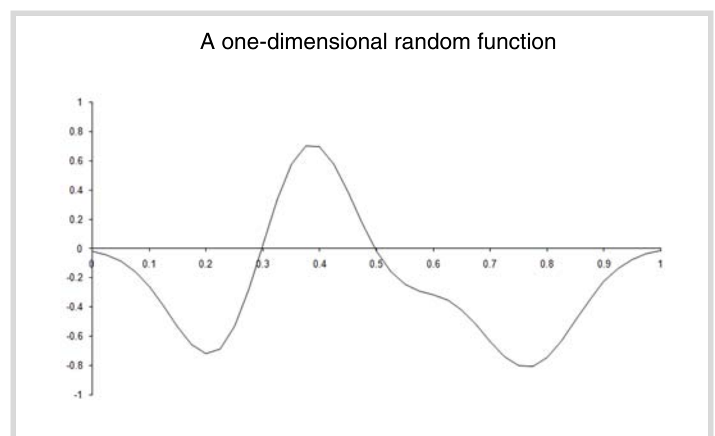


Figure 2

I'm going to use radial basis functions, so named because they depend only on how far an evaluated point is from a central point

```
namespace evolve
{
  class random_function :
     public std::binary_function<double,
     double, double>
  {
    class node
    {
    public:
      node();
      double operator()(double x, double y) const;
    private:
      double x_;
      double y_;
      double w_;
    };
    typedef std::vector<node> nodes_type;
  public:
    random_function();
    explicit random_function(size_t nodes);
    double operator()(double x, double y) const;
  private:
    nodes_type nodes_;
  };
}
```

Listing 1

```
evolve::random_function::node::node() :
   x_(rnd(0.6)+0.2),
   y_(rnd(0.6)+0.2),
   w_(rnd(2.0)-1.0)
{
}
double
evolve::random_function::node::operator()(
   double x, double y) const
{
  double d2 = (x-x_)*(x-x_)+(y-y_)*(y-y_);
  return w_*exp(-70.0*d2);
}
```

Listing 2

```
evolve::random_function::random_function()
{
}
evolve::random_function::random_function(
   size_t nodes)
{
  nodes_.reserve(nodes);
  while(nodes--) nodes_.push_back(node());
}
```

Listing 3

```
double
evolve::random_function::operator()(
   double x, double y) const
{
  double result = 0.0;
  nodes_type::const_iterator first =
     nodes_.begin();
  nodes_type::const_iterator last  =
       nodes_.end();
  while(first!=last) result += (*first++)(x, y);
  return result;
}
```

Listing 4

Now we can take a look at the implementation of **random_function**. First off, the constructors (Listing 3).

Both are pretty simple. The initialising constructor simply pushes the required number of default constructed **node**s onto the **nodes_** collection, relying upon their constructors to randomly initialise them.

The function call operator is also pretty straightforward (Listing 4).

As described before, this simply iterates over the **nodes_**, summing the results of each function call.

At long last we're ready to implement a specialisation of **biochem** (Listing 5).

```
class example_biochem : public evolve::biochem
{
public:
  explicit example_biochem(size_t nodes);
  virtual size_t genome_base() const;
  virtual size_t genome_size() const;
  virtual size_t phenome_size() const;
  virtual double p_select() const;
  virtual void   develop(
     const genome_type &genome,
     phenome_type &phenome) const;

private:
  enum {precision=5};

  double make_x(const genome_type &genome) const;
  double make_y(const genome_type &genome) const;
  double make_var(
     genome_type::const_iterator first,
     genome_type::const_iterator last) const;
  evolve::random_function f1_;
    evolve::random_function f2_;
  };
```

Listing 5

the develop member function is responsible for mapping from the genome to the phenome

```
size_t
example_biochem::genome_base() const
{
  return 10;
}
size_t
example_biochem::genome_size() const
{
  return 2*precision;
}
size_t
example_biochem::phenome_size() const
{
  return 2;
}
double
example_biochem::p_select() const
{
  return 0.9;
}
```

Listing 6

The **example_biochem** treats the genome as a discrete representation of two variables each with **precision**, or five, digits each. The two **random_functions** map these variables onto a two dimensional phenome. We hard code the **biochem** properties to reflect this, as illustrated in Listing 6.

Note that we're sticking with the selection probability of 0.9 that we used in our original model.

The constructor simply initialises the two **random_function**s with the required number of nodes.

```
example_biochem::example_biochem(
   size_t nodes) : f1_(nodes), f2_(nodes)
{
}
```

```
void
example_biochem::develop(
   const genome_type &genome,
   phenome_type &phenome) const
{
  if(genome.size()!=genome_size())
     throw std::logic_error("");
  phenome.resize(phenome_size());
  double x = make_x(genome);
  double y = make_y(genome);
  phenome[0] = f1_(x, y);
  phenome[1] = f2_(x, y);
}
```

Listing 7

```
double
example_biochem::make_x(
   const genome_type &genome) const
{
  if(genome.size()!=genome_size())
     throw std::invalid_argument("");
  return make_var(genome.begin(),
     genome.begin()+precision);
}
double
example_biochem::make_y(
   const genome_type &genome) const
{
  if(genome.size()!=genome_size())
     throw std::invalid_argument("");
  return make_var(genome.begin()+precision,
     genome.end());
}
```

Listing 8

As you no doubt recall, the **develop** member function is responsible for mapping from the genome to the phenome and our example does so using the **make_x** and **make_y** member functions to create the two variables we need for the **random_function**s from the genome. (See Listing 7.)

The **make_x** and **make_y** member functions themselves simply forward to the **make_var** member function with each of the two **precision** digit halves of the genome (Listing 8).

The **make_var** function maps an iterator range to a variable between zero and one. It does this by treating the elements in the range as progressively smaller digits, dividing the multiplying factor by the **genome_base** at each step to accomplish this.

Since we're using base 10, this is equivalent to simply putting a decimal point at the start of the sequence of digits. The elements 1, 2, 3, 4 and 5

```
double
example_biochem::make_var(
   genome_type::const_iterator first,
   genome_type::const_iterator last) const
{
  double var = 0.0;
  double mult = 1.0;
  while(first!=last)
  {
    mult /= double(genome_base());
    var += *first++ * mult;
  }
  return var;
}
```

Listing 9

would therefore map to 0.12345.

So what are the results of a simulation based on our **example_biochem**? Figure 3 illustrates the distribution of the phenomes of a **population** of 100 **individual**s from their initial state to their states after 10, 50 and 250 generations.

Well, it certainly looks like the **population** converges to a Pareto optimal front. However, we cannot be certain that there are not any better trade offs between the two functions. To check, we'll need to take a comprehensive sample of the two variables and record the optimal front.

We can do this by maintaining a list of undominated points. As we test each new point, we check to see if it is dominated by any in the list. If not, we remove any points in the list that it dominates and finally add it to the list.

In order to calculate the function values for any point, we first need to add another member function to **example_biochem**. The function call operator seems a reasonable choice and its implementation is illustrated in Listing 10.

Now we can implement a function to sample the optimal front.

```
typedef std::pair<double, double> point;
typedef std::vector<std::pair<point,
   point> > front;
front sample_front(size_t n,
   const example_biochem &b);
```

```
class example_biochem : public evolve::biochem
{
public:
  ...
  std::pair<double, double> operator()(double x,
     double y) const;
  ...
};
std::pair<double, double>
example_biochem::operator()(double x,
   double y) const
{
  return std::make_pair(f1_(x, y), f2_(x, y));
}
```

Listing 10

Note that the **std::pair** of **point**s represents the two input variables with its **first** member and the two elements of the output phenome with its **second** member.

The implementation of **sample_front** (Listing 11) follows the scheme outlined above, slicing each of the variables into **n** samples and iterating over them, filling up the sample of the optimal front.

The two helper functions, **on_sample_front** and **remove_dominated**, implement the check that a point is not dominated by any already in the list and the removal of any that it dominates respectively. (See Listing 12.)

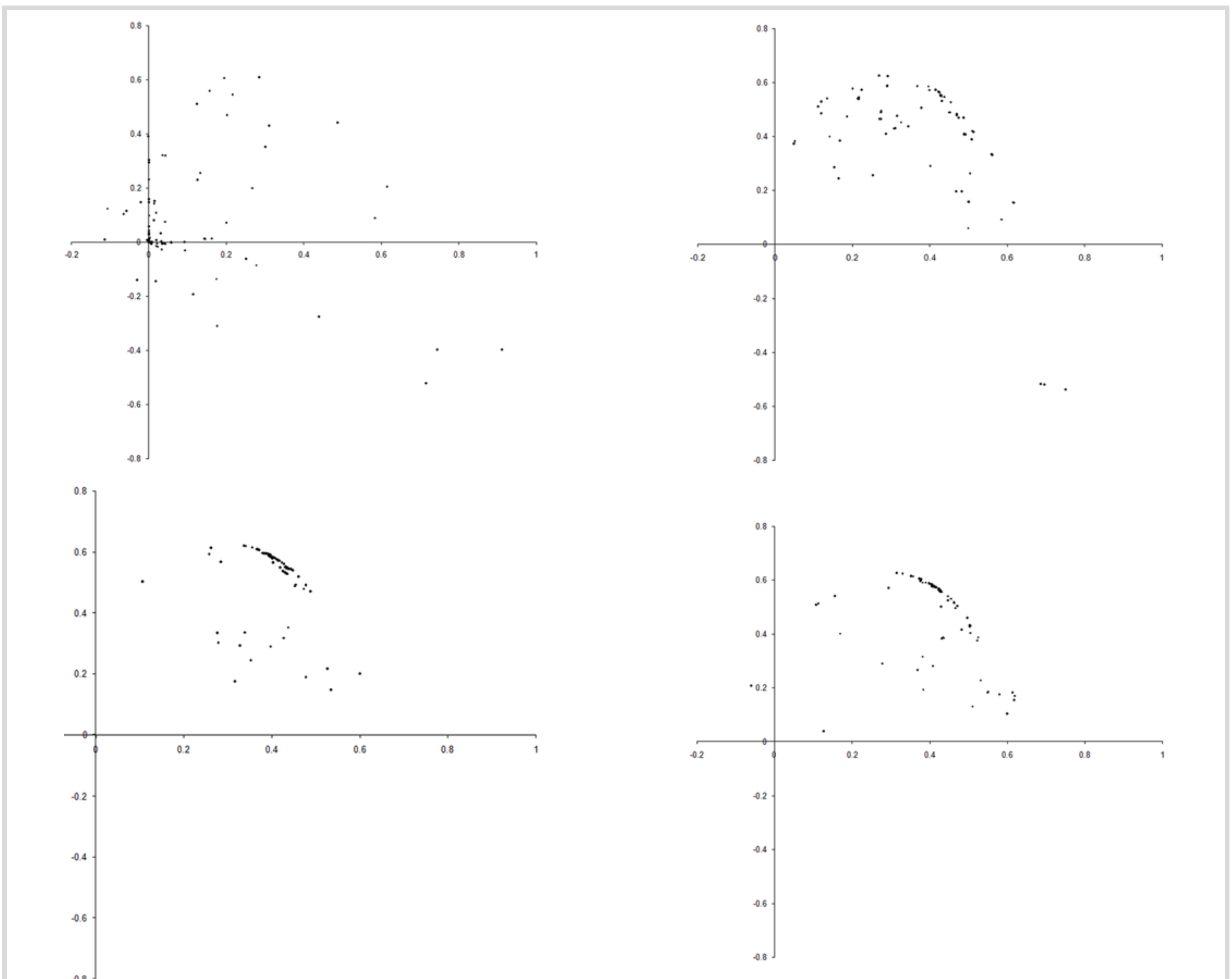


Figure 3

```
front
sample_front(size_t n, const example_biochem &b)
{
  front sample;
  for(size_t i=0;i!=n;++i)
  {
    for(size_t j=0;j!=n;++j)
    {
      double x = double(i) / double(n);
      double y = double(j) / double(n);
      point val = b(x, y);
      if(on_sample_front(val, sample))
     {
        remove_dominated(val, sample);
        sample.push_back(std::make_pair(
           point(x, y), val));
      }
    }
  }
  return sample;
}
```

Listing 11

Note that the loop condition in **on_sample_front** and the comparison at the heart of **remove_dominated** are simply hard coded versions of the complement of the **pareto_compare** function we developed earlier.

Note that for *n* samples of each variable, the **sample_front** function must iterate over n^2 points, making approximately O(*n*) comparisons at each step (since the optimal front is a line in the plane). We can't therefore afford to work to the same precision as we do in the simulation itself.

Figure 4 adds the 100 by 100 point sample of the optimal front to the final state of our earlier simulation.

```
bool
on_sample_front(const point &value,
   const front &sample)
{
  front::const_iterator first = sample.begin();
  front::const_iterator last  = sample.end();
  while(first!=last &&
           (value.first>=first->second.first ||
            value.second>=first->second.second))
  {
    ++first;
  }
  return first==last;
}
void
remove_dominated(const point &value,
   front &sample)
{
  front::iterator first = sample.begin();
  while(first!=sample.end())
  {
    if(value.first>=first->second.first &&
       value.second>=first->second.second)
    {
      first = sample.erase(first);
    }
    else
    {
      ++first;
    }
  }
}
```

Listing 12

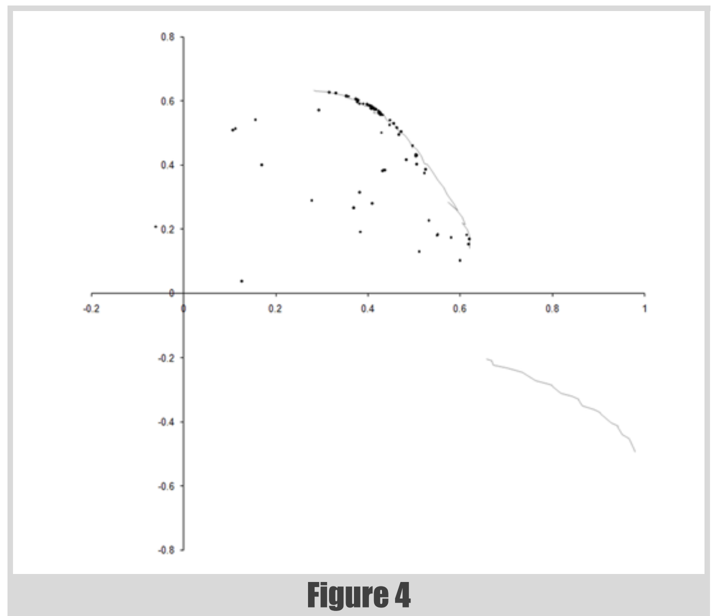


Figure 4

Well, our simulation has certainly ended up distributed along part of the optimal front, but seems to have missed an entire section of it. To see why, we need to take a look at how the points whose values lie on the optimal front are distributed.

This requires a further change to the **example_biochem**. To compute the distribution of the variables represented by the genomes, we need to make the **make_x** and **make_y** member functions public. Figure 5 compares the final distribution of points in our simulation to the set comprising the sample of the Pareto optimal front.

Clearly our **population** has converged on only one of the two regions from which the optimal front is formed. The question is, why?

To investigate this, we need to examine the properties of our **example_biochem** near each of the two regions of the optimal front. We do this by sampling squares of side 0.3 centred on the mid points of the two regions, as illustrated along with the results in Figure 6.

The 100 point uniform sample of the two regions clearly shows that points in the region of the rightmost front, marked with + symbols, are dominated by points in the region of the leftmost, marked with x symbols. The evolution process during our simulation has therefore converged on the portion of the optimal front located in the generally better region.

So presumably, if our **example_biochem** were to have several mutually dominant regions, we should expect to find individuals populating many of them. Well, sometimes, as Figure 7 illustrates.

In this case the front is formed from 3 separate regions, of which 2 are represented in the final **population**. Such regions will not necessarily persist for the duration of a simulation, however. In many cases multiple optimal regions will be populated part way through a simulation, only to disappear by the end. This is due to an effect known as genetic drift, which is also observed in the natural world [Frankham02].

In a small population, a less densely occupied region has a relatively large risk of being out competed by a more densely occupied region. The latter will likely have a greater number of dominant **individual**s and hence a better chance of representation after selection. The cumulative effect is that the population can converge on a single region, even if that region is not quantitatively better than the other. This can be mitigated with geographically separated sub **population**s or, in the terms of our model, several independent simulations. If the loss of diversity is solely due to genetic drift, different simulations will converge on different optimal regions and the combination of the final states will yield a more complete picture of the optimal front.

Returning to the argument that evolution cannot create information, only destroy it, one could argue that the success of our simulation is due to the initial random distribution of **individual**s in the **population**. Could

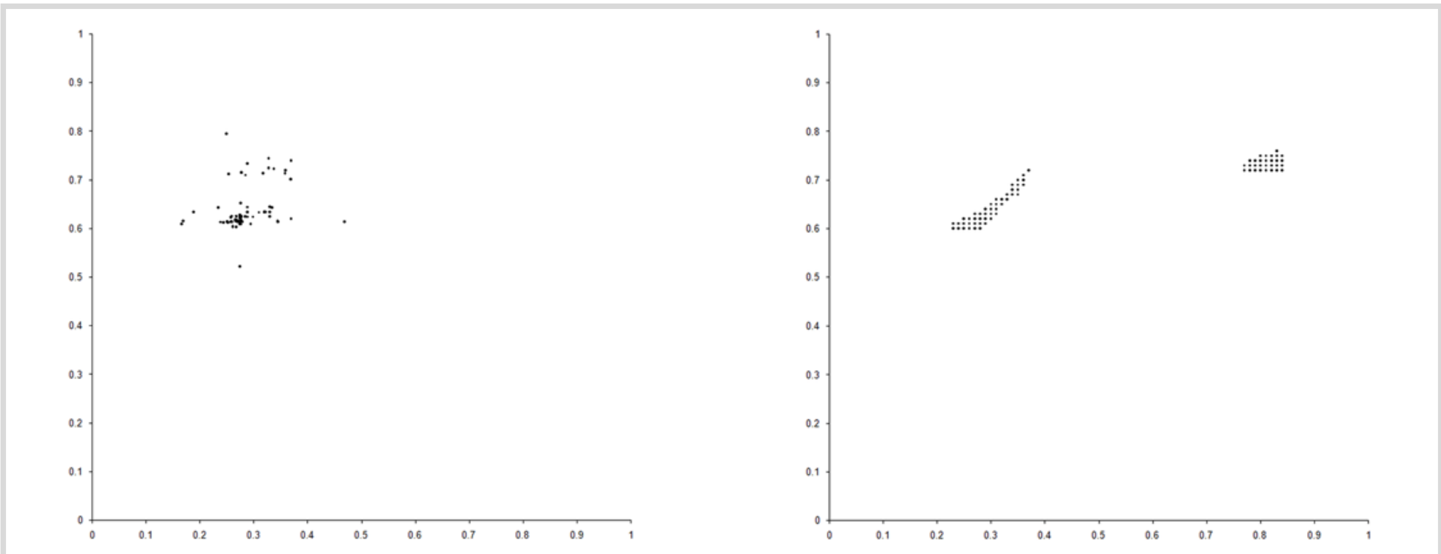


Figure 5

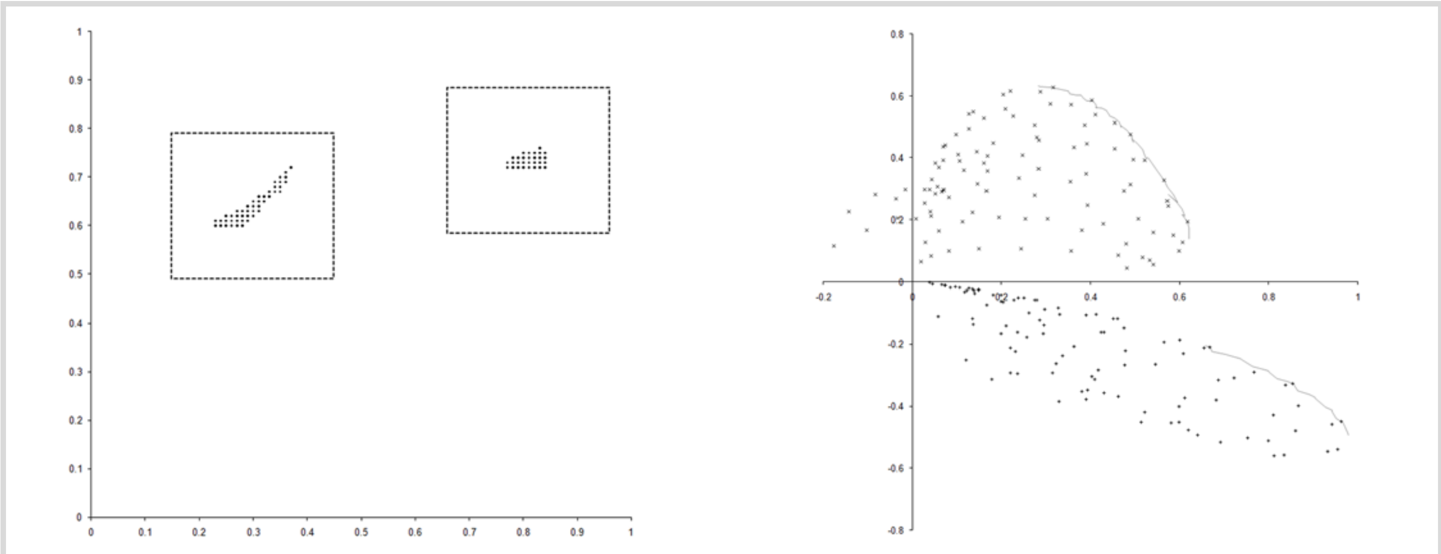


Figure 6

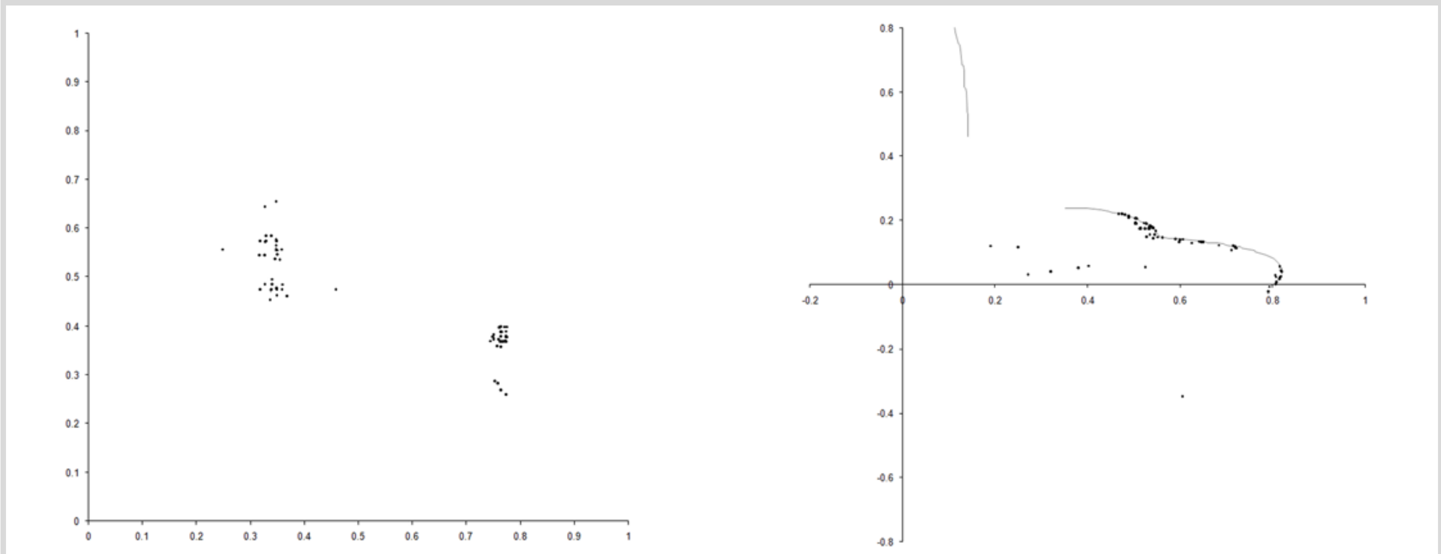


Figure 7

evolution simply be concentrating on the best region of the initial sample, rather than seeking out novelty?

This question is easily answered by forcing the **`individual`**s in the initial **`population`** to start at the same point. This requires a couple of minor changes to the **`individual`** and **`population`** classes.

Listing 13 shows the new constructors we need to add to these classes.

Their implementation is pretty straightforward, replacing the random initialisation of the phenome with assignment to the initial value. (See Listing 14.)

Note that we don't call **`reset`** in the new **`population`** constructor since this time we want all of the **`individual`**s to be the same.

Figure 8 shows the results of repeating our original simulation with an initial **`population`** formed entirely from **`individual`**s with zero initialised genomes.

Clearly, the initial distribution seems to have had little impact on the final state of the **`population`**.

So, given that our simulation has converged on the optimal set of trade offs, or at least the most stable of them, despite starting far from that set, where does the information actually come from?

Well, one further criticism that has been made is that these kinds of simulations somehow inadvertently encode the answers into the process; that we have unconsciously cheated and hidden the solution in the code.

You may be surprised, but I am willing to concede this point. The information describing the best **`individual`**s *is* hidden in the code. Specifically, it's in **`example_biochem`**. Whilst we may find it difficult to spot which genomes are optimal simply by examining the complex trade offs between the two functions, it does not mean that the information is not there. The Pareto optimal front exists, whether or not it's easy for us to identify.

However, I do not agree that this represents a weakness in our model.

Just as the information describing the Pareto optimal front is written into our **`example_biochem`**, so the information describing the rich tapestry of life we find around us is written into the laws of nature; into physics and chemistry. We simply do not have the wherewithal to read it.

Evolution, however, suffers no such illiteracy. It may not create information, but it is supremely effective at distilling it from the environment in which it operates.

The extraordinary power of evolution has not escaped the computer science community. From its beginnings in the 1960s and 1970s [Rechenberg65] [Klockgether70] [Holland75] the field of evolutionary computing has blossomed [Goldberg89] [Koza92] [Vose99]. Whilst the details generally differ from those of our simulation, the general principles are the same. The quality of a genome is measured by a function, or functions, that model a design problem, say improving the efficiency and reducing the emissions of a diesel engine [Hiroyasu02]. Through reproduction and selection, evolution has repeatedly proven capable of extracting optimal, or at least near optimal, solutions to problems that have previously evaded us [Hiroyasu02].

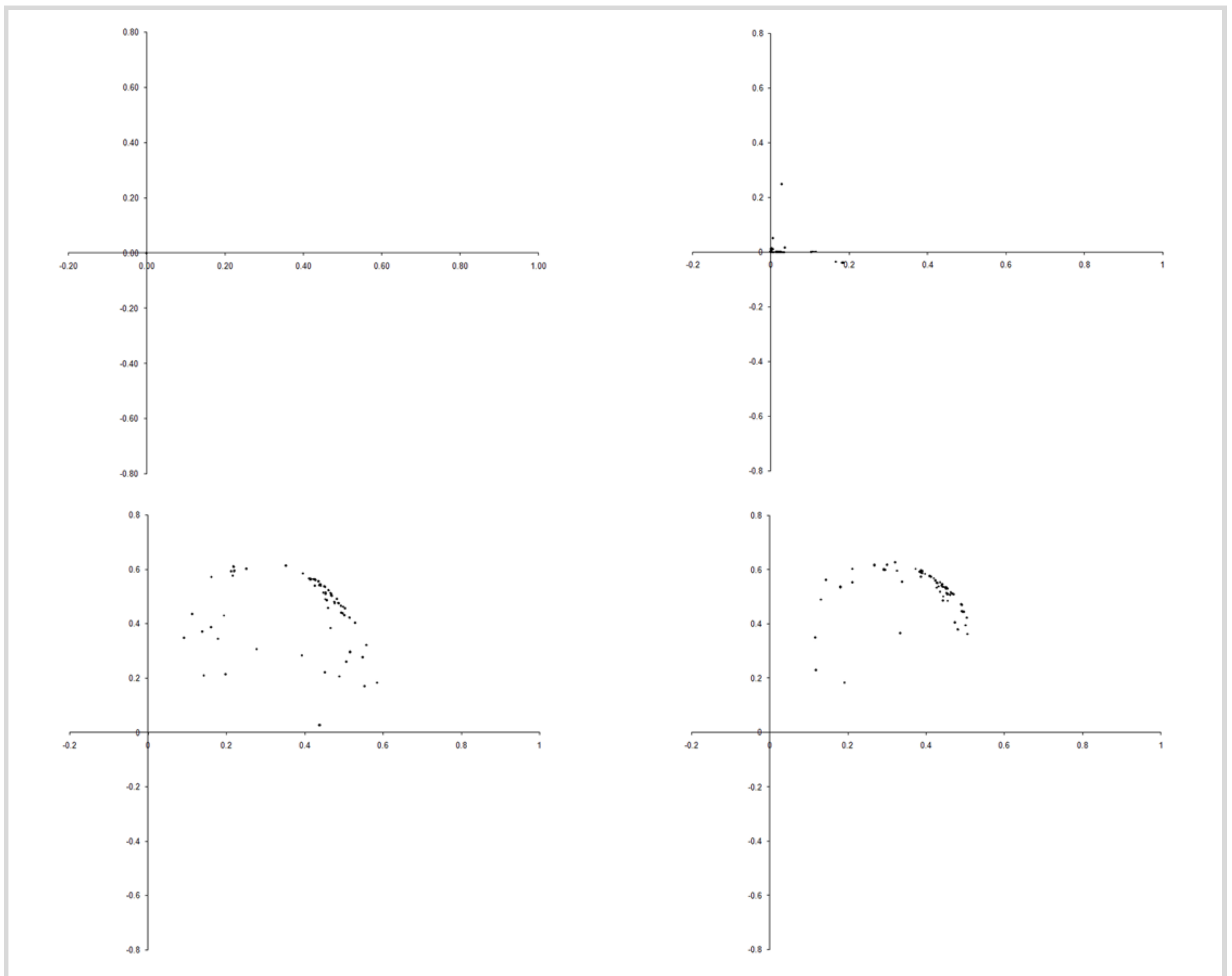


Figure 8

```
namespace evolve
{
  class individual
  {
  public:
    ...
    individual(unsigned long init_val,
       const biochem &b);
    ...
  };
  class population
  {
  public:
    ...
    population(size_t n, unsigned long init_val,
       const biochem &b);
    ...
  };
}
```

Listing 13

```
evolve::individual::individual(
   unsigned long init_val,
   const biochem &biochem) :
   biochem_(&biochem),
   genome_(biochem.genome_size(), init_val)
{
  develop();
}
evolve::population::population(size_t n,
   unsigned long init_val,
   const biochem &biochem) : biochem_(biochem),
   population_(n, individual(init_val, biochem)),
   offspring_(2*n)
{
}
```

Listing 14

There are still some aspects of our model that could use some improvement, however. For example, we are not modelling the effect of limited resources during the development of an `individual`. If you have the time and the inclination, you might consider adapting the code to reflect this, perhaps by introducing some penalty to `individual`s in densely populated regions.

If you make any further discoveries, I'd be delighted to hear about them. ■

Acknowledgements

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

References & Further Reading

[Frankham02] Frankham, R., Ballou, J. and Briscoe, D., *Introduction to Conservation Genetics*, Cambridge University Press, 2002.

[Goldberg89] Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Hiroyasu02] 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.

[Holland75] 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.

[Klockgether70] 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.

[Koza92] Koza, J., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[Meinkel97] Meinke1, M., Abdelfattah1, A. and Krause1, E., 'Simulation of piston engine flows in realistic geometries', *Proceedings of the Fifteenth International Conference on Numerical Methods in Fluid Dynamics*, vol. 490, pp. 195-200, Springer Berlin, 1997.

[Rechenberg65] Rechenberg, I., *Cybernetic Solution Path of an Experimental Problem*, Royal Aircraft Establishment, Library Translation 1122, 1965.

[Vose99] Vose, M., *The Simple Genetic Algorithm: Foundations and Theory*, MIT Press, 1999.

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

applied informatics

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

Model View Controller with Java Swing

It’s recommended to keep user interface and data model separate. Paul Grenyer looks a common solution.

In *Patterns of Enterprise Application Architecture* [PEAA] Martin Fowler tells us that the Model View Controller (MVC) splits user interface interaction into three distinct roles (Figure 1):

- **Model** – The model holds and manipulates domain data (sometimes called business logic or the back end).
- **View** – A view renders some or all of the data contained within the model.
- **Controller** – The controller takes input from the user and uses it to update the model and to determine when to redraw the view(s).

MVC is all about separating concerns. The model and views separate the data from the views and the controller and the view separate user input from the views.

Another version of the MVC pattern employs the controller as a mediator between the views and model (Figure 2).

The controller still takes user input, but now it passes it on to model. It also passes commands from the view to the model and takes events from the model and passes them on to the view. This version provides greater separation as the model and view no longer need to know about each other to communicate.

In this article I am going to describe a real problem I had and demonstrate how I solved it in Java with MVC. I am going to assume familiarity with both Java 6 and Swing.

It is very important to implement MVC carefully with a good set of tests. Benjamin Booth discusses this in his article ‘The M/VC Antipattern’. [MVC Antipattern].

The problem

As I have mentioned in previous articles (and probably to everyone’s boredom on accu-general) I am writing a file viewer application that allows fixed length record files in excess of 4GB to be viewed without loading the entire file into memory. I have had a few failed attempts to write it in C# recently (although recent discoveries have encouraged me to try again), but it was not until I had a go in Java with its `JTable` and `AbstractTableModel` classes that I really made some progress. These two classes are themselves a model (`AbstractTableModel`) and view (`Jtable`). However, in the example I’ll discuss in this article they actually form part of one of the views.

The file viewer application needs to be able to handle multiple files in a single instance. The easiest and most convenient way to do this is with a tabbed view (Figure 3).

I have completed the back-end business logic which models the file and

Paul Grenyer An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

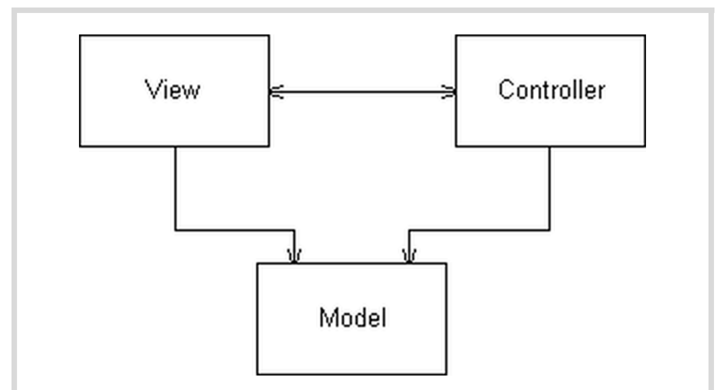


Figure 1

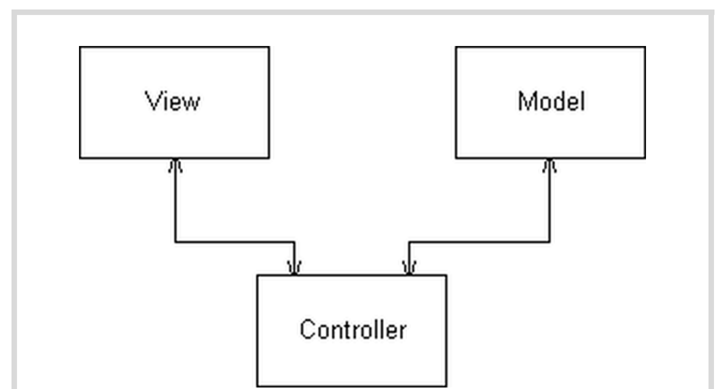
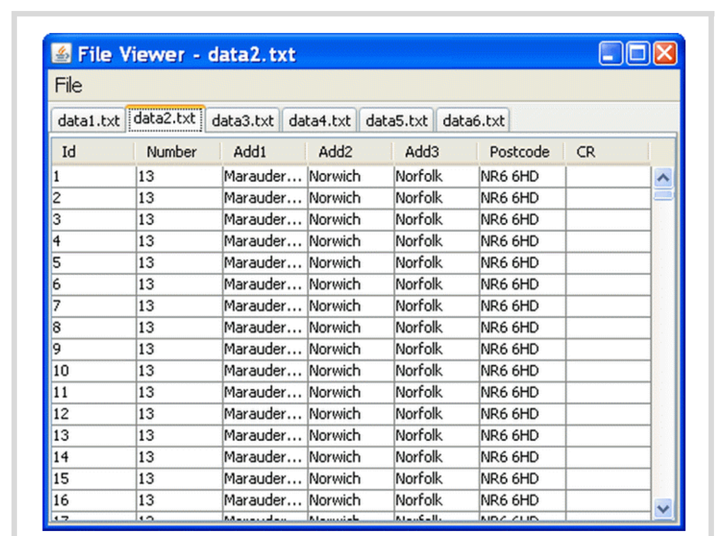


Figure 2



Id	Number	Add1	Add2	Add3	Postcode	CR
1	13	Marauder...	Norwich	Norfolk	NR6 6HD	
2	13	Marauder...	Norwich	Norfolk	NR6 6HD	
3	13	Marauder...	Norwich	Norfolk	NR6 6HD	
4	13	Marauder...	Norwich	Norfolk	NR6 6HD	
5	13	Marauder...	Norwich	Norfolk	NR6 6HD	
6	13	Marauder...	Norwich	Norfolk	NR6 6HD	
7	13	Marauder...	Norwich	Norfolk	NR6 6HD	
8	13	Marauder...	Norwich	Norfolk	NR6 6HD	
9	13	Marauder...	Norwich	Norfolk	NR6 6HD	
10	13	Marauder...	Norwich	Norfolk	NR6 6HD	
11	13	Marauder...	Norwich	Norfolk	NR6 6HD	
12	13	Marauder...	Norwich	Norfolk	NR6 6HD	
13	13	Marauder...	Norwich	Norfolk	NR6 6HD	
14	13	Marauder...	Norwich	Norfolk	NR6 6HD	
15	13	Marauder...	Norwich	Norfolk	NR6 6HD	
16	13	Marauder...	Norwich	Norfolk	NR6 6HD	

Figure 3

The Model View Controller (MVC) splits user interface interaction into three distinct roles

```
public interface Project
{
  public abstract RecordReader getRecordReader()
     throws RecordReaderException;
  public abstract Layout getLayout() throws
     LayoutException;
  public abstract String getDataDescription();
  public abstract void close();
}
```

Listing 1

its associated layout (which describes how records are divided up into fields) as a project with the interface in Listing 1.

I will look in a bit more detail at most of the methods in the interface as the article progresses, but for now the important methods are `getRecordReader` and `getLayout`. `getRecordReader` gives the `AbstractTableModel` random access to the records and fields in the file and `getLayout` gives access to the layout which allows the view to name and order its fields.

The table model implementation I have looks like Listing 2.

I've omitted exception handling and a few other bits and pieces that are not relevant to the example. Basically a `RecordGrid` object holds a reference to a `Project` object and uses it to populate the table cells and column titles on request. Rows are populated one at a time from column `0` to column x, so every time column `0` is requested a new record is loaded. This reduces the amount of file access that would be required if a record was loaded every time a cell was requested.

Every time a new project is created the code in Listing 3 is used to create a tab for the file.

Again, exception handling has been omitted. A reference to the `RecordReader` is created in order to get a name for the tab by calling the `getDataDescription` method. The new `Project` object is passed to a new `RecordGrid` object, which is then used as a `JTable` model. A new scrollable pane is created using the table and in turn used to create a

```
public class DataPane extends JtabbedPane
{
  public void addProject(Project project)
  {
    RecordReader recordReader =
       project.getRecordReader();
    TableModel model = new RecordGrid(project);
    JTable table = new JTable(model);
    this.addTab(recordReader.getDataDescription(),
       new JScrollPane(table));
    this.setSelectedIndex(this.getTabCount()-1);
  }
  ...
}
```

Listing 3

```
public class RecordGrid extends AbstractTableModel
{
  private final Project project;
  private Record record = null;
  public RecordGrid(Project project)
  {
    this.project = project;
  }
  public Project getProject()
  {
    return project;
  }

  @Override
  public int getRowCount()
  {
    return (int)
       project.getRecordReader().getRecordCount();
  }

  @Override
  public int getColumnCount()
  {
    return project.getLayout().getFieldCount();
  }

  @Override
  public Object getValueAt(int row, int column)
  {
    if (column == 0)
    {
      record =
         project.getRecordReader().getRecord(row);
    }
    return record.getField(column);
  }

  @Override
  public String getColumnName(int column)
  {
    return
       project.getLayout().getFieldName(column);
  }

  @Override
  public Class<? extends String> getColumnClass(
     int column)
  {
    return String.class;
  }
}
```

Listing 2

```
JScrollPane pane = (JScrollPane)
   tabbedPane.getSelectedComponent();
Component coms[] = pane.getComponents();
JViewport viewPort = (JViewport) coms[0];
coms = viewPort.getComponents();
JTable table = (JTable) coms[0];
RecordGrid grid = (RecordGrid) table.getModel();
project = grid.getProject();
```

Listing 4

new tab. Finally the new table is made the currently selected tab. All straight forward and not particularly complicated or problematic.

The problem comes when you want to get the **`Project`** object out of the current tab so that you can, for example, call **`close`** on it or use it to set the main window title bar. Listing 4 shows one way it can be done.

It relies on the fact that every object is a component of another object. Sometimes, as shown above, requesting a component's child component returns an array of components and, although this code doesn't show it, the required component needs to be found within the array. This is messy and potentially unreliable. After writing this code I felt there had to be a better way. So I asked some people. Roger Orr came up with a much simpler solution:

```
JTable table =
   (Jtable)pane.getViewport().getView();
```

Something still didn't feel right though. The code is querying the GUI components to navigate the object relationships. This breaks encapsulation as changing the GUI layout would break this code. There are also other ways, but none of them seemed to be the right solution either.

The solution

The general consensus of opinion was that I was mad to have a user interface component (the tabs) managing a data component (the project) and that I should be using the mediator version of the Model View Controller (MVC) described above.

I thought I pretty much had how MVC worked nailed down because of the MFC document view model stuff (basically just a model and views) I had done early in my career, but reading up on MVC and Swing in the various books I had just left me confused in terms of implementation. Then I googled and found Java SE Application Design With MVC [JADwMVC] on the Sun website. Suddenly everything was much clearer and I set about knocking up a prototype.

The main concern I had was keeping the tabbed view in sync with the model, so that when I selected a tab the currently selected project in the model was changed to reflect it correctly and when a new project was added to or deleted from the model it was also added or removed from the tabbed view. Should I remove every tab from the tabbed view and redraw when the model was updated or try and add and remove tabs one at a time in line with the model?

I decided to start developing my MVC prototype and cross that bridge when I came to it. The beauty of the mediator MVC patterns is that each component can be worked on and changed individually to a greater or lesser extent. The concerns are well separated.

The model

The file viewer model:

- Needs to handle multiple projects.
- Needs to have a mechanism for adding projects.
- Needs to have a mechanism for deleting projects.
- Needs to have a mechanism for setting the current project.
- Needs to have a mechanism for getting the current project.
- Should have a mechanism for getting the number of projects for testing purposes.
- Needs to fire events when properties change (e.g. a project is added, deleted or selected).

```
public interface ProjectOrganiser
{
  public abstract void addProject(Project prj);
  public abstract int getProjectCount();
  public abstract void setCurrentProjectIndex(
     int index);
  public abstract int getCurrentProjectIndex();
  public abstract Project getCurrentProject();
  public abstract void deleteCurrentProject();
}
```

Listing 5

The controller will have to mediate a number of these operations from views and menus to the model. So both will have similar interfaces for organizing projects within the model. The model interface looks like Listing 5.

New projects will be created outside of the controller, so the **`newProject`** method takes a **`Project`** reference. Fully unit testing a model like this is relatively easy, but beyond the scope of this article. The other methods perform the other required operations, with the exception of firing events.

Implementing the model is straightforward. I'll go through the data members first and then each method in turn.

```
public class Projects implements ProjectOrganiser
{
  private final List<Project> projects =
     new ArrayList<Project>();
  private int currentProjectIndex = -1;
  ...
}
```

The model is essentially a container for storing and manipulating projects, so it needs a way of storing the projects. An `ArrayList` is ideal. The model also needs to indicate which project is currently selected. To keep track it stores the **`ArrayList`** index of the currently selected project. If the **`ArrayList`** is empty or no project is currently selected then **`currentProjectIndex`**'s value is **`-1`**. Initially there are no projects.

The **`addProject`** method adds the new project to the **`ArrayList`** and sets it as the current project:

```
public void addProject(Project prj)
{
  projects.add(prj);
  setCurrentProjectIndex(projects.size() - 1);
}
```

The **`getProjectCount`** method simply asks the **`ArrayList`** its size and returns it.

The **`setCurrentProjectIndex`** method checks the index it is passed to make sure it is within the permitted range. It can either be **`-1`** or a valid index within the **`ArrayList`**. If the index is not valid it constructs an exception message explaining the problem and throws an **`IndexOutOfBoundsException`**. If the index is valid it is used to set the current project. (See Listing 6.)

The **`getCurrentProjectIndex`** simply returns **`currentProjectIndex`**.

The **`getCurrentProject`** method relies on the fact that the **`setCurrentProjectIndex`** method has policed the value of **`currentProjectIndex`** successfully. Therefore it only checks to make sure **`currentProjectIndex`** is greater than or equal to **`0`**. If it is it returns the corresponding project from the **`ArrayList`**, otherwise **`null`** (Listing 7).

The **`deleteCurrentProject`** method is by far the most interesting in the model. It is also the most important method to get a unit test around. It checks to make sure there are projects in the **`ArrayList`**. If there are then it calls **`close`** on and then deletes the current project from the **`ArrayList`** and calculates which the next selected project should be. If, following the deletion, another project moves into the same **`ArrayList`** index it becomes the next selected project. Otherwise the project at the

```
public void setCurrentProjectIndex( int index )
{
  if (index >= getProjectCount() || index < -1)
  {
    final StringBuilder msg = new StringBuilder();
    msg.append("Set current project to ");
    msg.append(index);
    msg.append(" failed. Project count is ");
    msg.append(getProjectCount());
    msg.append(".");
    throw new IndexOutOfBoundsException(
       msg.toString());
  }
  currentProjectIndex = index;
}
```

Listing 6

```
public void getCurrentProject()
{
  Project project = null;
  if (getCurrentProjectIndex() >= 0)
  {
    project = projects.get(currentProjectIndex);
  }
  return project;
}
```

Listing 7

previous index in the **ArrayList** is used. If there are no longer any projects in the **ArrayList** the current index is set to **-1**. (Listing 8.)

As you can see, manipulating projects within the model via the **ProjectOrganiser** interface is very straight forward. However, there is currently no way of notifying the controller when a property changes. The Java SE Application Design With MVC article recommends using the Java Beans component called Property Change Support. The **PropertyChangeSupport** class makes it easy to fire and listen for property change events. It allows chaining of listeners, as well as filtering by property name.

To enable the registering of listeners and the firing of events a few changes need to be made to **ProjectOrganiser** and **Projects**. First, event names and a change listener registering method need to be added to **ProjectOrganiser** (see Listing 9).

Later versions of Java have support for enums. However, property change support does not, but uses strings instead. Enums could be used in conjunction with the **toString** method, but this makes their use overly verbose and does not give any clear advantages.

Then the **Projects** class needs a **PropertyChangeSupport** object and the methods to add listeners and fire events (Listing 10).

The **PropertyChangeSupport** object needs to know the source of the events it is firing so this is passed in. The **addPropertyChangeListener** method simply forwards to the same method of the **PropertyChangeSupport** object.

```
public void deleteCurrentProject()
{
  if (getCurrentProjectIndex() >= 0)
  {
    final int currentIndex =
       getCurrentProjectIndex();
    getCurrentProject().close();
    projects.remove(currentIndex);
    int nextIndex = -1;
    final int projectCount = getProjectCount();
    if (currentIndex < projectCount)
    {
      nextIndex = currentIndex;
    }
    else if (projectCount > 0)
    {
      nextIndex = projectCount - 1;
    }
    setCurrentProjectIndex(nextIndex);
  }
}
```

Listing 8

Any class that implements the **PropertyChangeListener** interface can receive events. The **firePropertyChange** method takes the name of the property that has changed, the property's old value and its new value. All of these, together with the source object, are passed to the event sink as a **PropertyChangeEvent**. If the old and new values are the same the event is not fired. This can be overcome by setting one or more of the old and new objects to **null**. We'll look at the implications of this shortly.

The **Projects** class now has the methods to fire events, but is not actually firing anything. The controller needs to be notified every time a project is

```
public class Projects implements ProjectOrganiser
{
  ...
  private final PropertyChangeSupport
     propertyChangeSupport =
     new PropertyChangeSupport(this);
  ...
  @Override
  public void addPropertyChangeListener(
     PropertyChangeListener listener)
  {
    propertyChangeSupport.
       addPropertyChangeListener(listener);
  }
  private void firePropertyChange(
     String propertyName, Object oldValue,
     Object newValue)
  {
    propertyChangeSupport.firePropertyChange(
       propertyName, oldValue, newValue);
  }
}
```

Listing 10

```
public interface ProjectOrganiser
{
public static final String NEW_PROJECT_EVENT = "ProjectOrganiser.NEW_PROJECT";
public static final String DELETE_PROJECT_EVENT = "ProjectOrganiser.DELETE_PROJECT";
public static final String CURRENT_PROJECT_INDEX_CHANGED_EVENT =
   "ProjectOrganiser.CURRENT_PROJECT_INDEX_CHANGED";
public static final String CURRENT_PROJECT_CHANGED_EVENT = "ProjectOrganiser.CURRENT_PROJECT_CHANGED";
public abstract void addPropertyChangeListener(PropertyChangeListener listener);
  ...
}
```

Listing 9

```
public void setCurrentProjectIndex( int index )
{ ...
  final int oldIndex = currentProjectIndex;
  currentProjectIndex = index;
  firePropertyChange(
     CURRENT_PROJECT_INDEX_CHANGED_EVENT,
     oldIndex, currentProjectIndex);
  firePropertyChange(
     CURRENT_PROJECT_CHANGED_EVENT,null,
     getCurrentProject());
}
```

Listing 11

created, deleted or selected. This means the **addProject**, **deleteProject** and **setCurrentProjectIndex** methods must be modified:

```
public void addProject(Project prj)
{
  Project old = getCurrentProject();
  projects.add(prj);
  firePropertyChange(NEW_PROJECT_EVENT,old,prj);
  setCurrentProjectIndex(projects.size() - 1);
}
```

The **addProject** method now stores a reference to the current project prior to the new project being added. The old project and the new project are both passed to the event as properties. This means that the event sink can perform any necessary clean up using the previously selected project and update itself with the new project without having to query the controller. Also, the old project and new project references will never refer to the same project, so the event will never be ignored.

The **setCurrentProjectIndex** method fires two events. The **CURRENT_PROJECT_INDEX_CHANGE_EVENT** event is fired when the current project *index* changes and the **CURRENT_PROJECT_CHANGE_EVENT** is fired when the current project changes. These are deceptively similar events. Consider when a project is deleted. If the project is in the middle of the **ArrayList** the project in front of it moves into its index. The project changes, but the index stays the same.

The **CURRENT_PROJECT_INDEX_CHANGE_EVENT** event passes both the old and new indexes. The **CURRENT_PROJECT_CHANGE_EVENT** is only passed the new project. The old value is always **null**. This is because, following a project deletion the old project no longer exists.

The **deleteCurrentProject** method requires some significant changes, shown in Listing 12.

Ideally, when a project is deleted the old value and the next project to be selected are be passed as the old and new values. The event needs to be fired before the project is actually removed so that any thing using it can clean up. This can make it difficult to work out which project is which. One way to be sure is to make a copy of the project **ArrayList**, remove the project to be deleted from it and then work out which the next selected project will be. To do this I wrote a helper class called **NextProject**. Overall it is more code, but it makes for a much neater solution to the **deleteCurrentProject** method and means the next project index only needs to be calculated once. Again, the deleted project and the next selected project will never be the same, so the event will not be ignored.

That completes the fully unit testable model. The model is by far the most complex and difficult part of the MVC to implement and get right. A good set of unit tests is essential. Once you have it right the view and controller follow quite easily.

The controller

The controller is the mediator between the model and the views. Therefore it makes sense to develop it next; otherwise the model and view could end up so incompatible that writing a controller would be very difficult.

```
public void deleteCurrentProject()
{
  if (getCurrentProjectIndex() >= 0)
  {
    final int currentIndex =
       getCurrentProjectIndex();
    final Project currentProject =
       getCurrentProject();
    final NextProject nextProject =
       new NextProject(projects, currentIndex);
    firePropertyChange(DELETE_PROJECT_EVENT,
    currentProject, nextProject.getProject());
    currentProject.close();
    projects.remove(currentIndex);
    setCurrentProjectIndex(
       nextProject.getIndex());
  }
}
```

Listing 12

The controller maintains a reference to the model and list of references to registered views. If the model changes it passes the event onto the views via the **ProjectOrganiserEventSink** interface.

```
public interface ProjectOrganiserEventSink
{
  public abstract void modelPropertyChange(
     final PropertyChangeEvent evt);
}
```

Menu items and registered views all maintain a reference to the controller and use it to pass actions to the model. Therefore the model has a number of methods that just forward to the model.

The code below shows the **Controller** properties and constructor:

```
public class Controller implements
   PropertyChangeListener
{
  private final ProjectOrganiser model;
  private List<ProjectOrganiserEventSink> views =
     new ArrayList<ProjectOrganiserEventSink>();
  public Controller(ProjectOrganiser model)
  {
    this.model = model;
    this.model.addPropertyChangeListener(this);
  }...
}
```

The **Controller** implements the **PropertyChangeListener** interface. The **PropertyChangeListener** interface allows the controller to receive events from the model. The **Controller** takes a reference to the model as a constructor parameter and uses it to register itself with the model. Views register themselves via the **addView** method:

```
public void addView(
   ProjectOrganiserEventSink view)
{
  views.add(view);
}
```

Events are passed to registered views via the overridden **propertyChange** method from the **PropertyChangeListener** interface:

```
@Override
public void propertyChange(
   PropertyChangeEvent evt)
{
  for (ProjectOrganiserEventSink view : views)
  {
    view.modelPropertyChange(evt);
  }
}
```

The model forwarding methods do just that, with the exception of the **newProject** method. The **newProject** method is different because it

has to create a project to pass to the model. The idea behind the `Project` interface is that it can be used to reference implementations for different file type and layout type combinations. Therefore I have written the `NewProjectDlg` class to allow the user to select the type of project they want. It then calls `createNew` on the project to do some project specific creation. A reference to the project can then be queried and passed to the model:

```
public void newProject(JFrame owner)
{
  NewProjectDlg d =
     new NewProjectDlg(owner,true);
  d.setVisible(true);
  Project prj = d.getProject();
  if (prj != null)
  {
    model.newProject(prj);
  }
}
```

The internal workings of the `NewProjectDlg` class are beyond the scope of this article.

The `Controller` is almost fully unit testable. The fly in the ointment is of course the `NewProjectDlg`. You just don't want it popping up in the middle of a test. There are a number of easy ways of getting round it, but they are beyond the scope of this article also. However, `Controller` is so simple that it hardly requires a unit test. Under normal circumstances I would write one anyway, but it would require some non-trivial mock objects that just do not seem worth it.

The view

Getting a view to receive events from the controller is very simple. It only requires the implementation of the `ProjectOrganiserEventSink` interface and then registering with the controller. The complexity comes with what you actually do with the view. I'm going to explain two examples. One that just updates the title of the main window when the current project changes and one that keeps tabs in sync with the model. (That was the method I decided to try and implement first. It worked as you'll see!)

Main window view

As I hinted at earlier, I come from an MFC background. Writing GUIs in Java with Swing is therefore an absolute dream by comparison. Instead of having to rely hugely on the wizard and get the linking right, with Swing a window can be created from a `main` function and a few simple objects.

The main window is a good place to create and reference the controller. In order to receive events from the controller the view must implement the `ProjectOrganiserEventSink` interface and register itself with the model (Listing 13).

When the current project changes, due to a project being added or deleted or the user selecting a different tab, the description of the project should be updated in the main window's title bar. This is done by handling the project changed event (Listing 14).

When the view receives the PropertyChangeEvent it checks to see what type of event it is. If it is a `CURRENT_PROJECT_CHANGED_EVENT` it gets the project from the event object. If the current project is null, for example if there are no projects in the model, it sets the default title, otherwise it gets a description from the project and concatenates that to the default title.

So far we have a main window that creates and handles events from a controller, but nothing that actually causes an event to be fired. To get events we need to be able to create and delete projects. One of the easiest ways to do this is via a menu. Menus are easy to create and anonymous classes give instant access to the controller. (See Listing 15.)

Projects can now be added to and deleted from the model via the main window's file menu. The main window handles an event from the controller that allows it to set its title based on the currently selected project.

Tabbed view

Projects are not much use if they cannot be viewed or selected, so what is needed is a tabbed view capable of displaying projects (Listing 16).

Swing has the `JTabbedPane` class that will do the job perfectly. Subclassing, as shown in Listing 16, gives a better level of encapsulation and control of the view's functionality. The view must also implement the

```
public class MainWindow extends JFrame
   implements ProjectOrganiserEventSink
{
  private final Controller controller;
  public MainWindow()
  {
    controller = new Controller(new Projects());
    controller.addView(this);
    this.setDefaultCloseOperation(
       JFrame.EXIT_ON_CLOSE);
    this.setSize(1000, 600);
  }
  @Override
  public void modelPropertyChange(
     PropertyChangeEvent evt)
  {
     ...
  }
  public static void main(String[] args)
  {
    javax.swing.SwingUtilities.invokeLater(
       new Runnable()
    {
      public void run()
      {
        new MainWindow().setVisible(true);
      }
    });
  }
}
```

Listing 13

```
public class MainWindow extends JFrame
   implements ProjectOrganiserEventSink
{
  private static final String TITLE =
     "File Viewer";
  ...
  @Override
  public void modelPropertyChange(
     PropertyChangeEvent evt)
  {
    if (evt.getPropertyName().equals(
      ProjectOrganiser.
      CURRENT_PROJECT_CHANGED_EVENT))
    {
      StringBuilder builder =
         new StringBuilder(TITLE);
      Project prj = (Project) evt.getNewValue();
      if (prj != null)
      {
        builder.append(" - ");
        builder.append(prj.getDataDescription())
      }
      setTitle(builder.toString());
    }
  }
  ...
}
```

Listing 14

```
public class MainWindow extends JFrame
   implements ProjectOrganiserEventSink
{
  private final Controller controller;
  public MainWindow()
  {
    controller = new Controller(new Projects());
    controller.addView(MainWindow.this);
    buildMenu();
  }
  private void buildMenu()
  {
    JMenu fileMenu = new JMenu("File");
    fileMenu.setMnemonic(KeyEvent.VK_F);
    JMenuItem newItem = new JMenuItem("New");
    newItem.setMnemonic(KeyEvent.VK_N);
    newItem.addActionListener(
       new ActionListener()
    {
      @Override
      public void actionPerformed(ActionEvent e)
      {
        controller.newProject(MainWindow.this);
      }
    });
    fileMenu.add(newItem);
    JMenuItem closeItem = new JMenuItem("Close");
    closeItem.setMnemonic(KeyEvent.VK_C);
    closeItem.addActionListener(
       new ActionListener()
    {
      @Override
      public void actionPerformed(ActionEvent e)
      {
        controller.deleteCurrentProject();
      }
    });
    fileMenu.add(closeItem);
    JMenuBar menuBar = new JMenuBar();
    menuBar.add(fileMenu);
    this.setJMenuBar(menuBar);
  }
}
```

Listing 15

ProjectOrganiserEventSink, maintain a reference to the controller, which must be passed into the constructor, and register itself with the controller. In order to be seen and used the **DataPane** also needs to be added to the main window:

```
public class DataPane extends JtabbedPane
   implements ProjectOrganiserEventSink
{
  private final Controller controller;
  public DataPane(Controller controller)
  {
    this.controller = controller;
    initialise();
  }
  private void initialise()
  {
    controller.addView(this);
  }
  @Override
  public void modelPropertyChange(
     PropertyChangeEvent evt)
  { ... }
}
```

Listing 16

```
public class MainWindow extends JFrame
   implements ProjectOrganiserEventSink
{
  ...
  public MainWindow()
  {
    ...
    this.add(new DataPane(controller));
    this.setDefaultCloseOperation(
       JFrame.EXIT_ON_CLOSE );
    this.setSize(1000, 600);
  }
  ...
}
```

As well as responding to events fired by the controller, the view also needs to notify the controller when a user has selected a different tab. This is done by writing a change listener:

```
public void initialise()
{
  controller.addView(this);
  addChangeListener(new ChangeListener()
  {
    @Override
    public void stateChanged(ChangeEvent arg0)
    {
      controller.setCurrentProjectIndex(
         getSelectedIndex());
    }
  });
}
```

Every time the tabbed control's state changes, for example the user selects a different tab, the change listener's **stateChanged** method is called. As the change listener is implemented as an anonymous class within **DataPane** it has access to **DataPane**'s properties. Therefore it can query the tabbed view for the current index and pass it to the controller.

The **modelPropertyChange** override handles events from the controller (Listing 18).

As before, the event's name is used to determine what sort of event it is. The **DataPane** handles the new project, project deleted and project index change events, passing where appropriate, the event's new value to another method for handling. The new value is a new project that has been created, the newly selected project following a project deletion, or the new index following a newly created or deleted project. **setSelectedIndex** is a method inherited via **JTabbedPane** and does exactly what you would

```
@Override
public void modelPropertyChange(
   PropertyChangeEvent evt)
{
  if (evt.getPropertyName().equals(
     ProjectOrganiser.NEW_PROJECT_EVENT))
  {
    addProject((Project)evt.getNewValue());
  }
  else if(evt.getPropertyName().equals(
     ProjectOrganiser.
     CURRENT_PROJECT_INDEX_CHANGED_EVENT))
  {
    setSelectedIndex((Integer) evt.getNewValue());
  }
  else if(evt.getPropertyName().equals(
     ProjectOrganiser.DELETE_PROJECT_EVENT))
  {
    deleteCurrentTab();
  }
}
```

Listing 17

expect. The **addProject** method is implemented as follows, with exception handling omitted for clarity:

```
  private void addProject(Project project)
  {
    if (project != null)
    {
      RecordReader recordReader =
         project.getRecordReader();
      TableModel model = new RecordGrid(project);
      JTable table = new JTable(model);
      this.addTab(
         recordReader.getDataDescription(),
         new JscrollPane(table));
    }
  }
```

The code should look familiar. It is almost identical to part of the code in 'The problem' section above. The only difference is that the newly added tab is not selected as the current tab. That is now set following a project index changed event.

```
private void deleteCurrentTab()
{
  if (getTabCount() > 0)
  {
    remove(this.getSelectedIndex());
  }
}
```

The **deleteCurrentTab** method checks to make sure there is at least one tab to delete, gets the index of the current tab and deletes it.

That completes the implementation of the **DataPane** view. Projects can now be added to and deleted from the application, tabs can be changed and the changes reflected in the main window title and the client area tabbed view (see Figure 4).

Conclusion

By implementing MVC and demonstrating how easy it is to manipulate, control and display the projects, I believe I have demonstrated the advantages of MVC over the original design I had, where the model was effectively buried in the view and difficult to get hold of when needed.

The mediator pattern keeps the model and views nicely decoupled. There is slight coupling of the view to the **ProjectOrganiser** interface as the event names are part of the interface. If this became an issue it would be simple to move the event names to their own class. I believe this is unnecessary at this stage.

I was also concerned about keeping the projects in the model and the associated tab in the view in sync. However, this problem was easily overcome:

- By relying on the fact that new tabs are always added at the end of the tabbed view and new projects are always added at the end of the model **ArrayList**.
- Using an event from the model to set the currently selected project in the tabbed view.
- Using an event from the model to remove the tabs for deleted projects from the tabbed view.

Finally I'll leave you with a comment from a colleague of mine:

> "I really like the idea of Model View Controller, but it takes so much code to do very little."

I think I've shown here that there is quite a lot of code involved in the MVC design compared to the model within the view design, but the separation and the ease with which the data can be manipulated is a clear advantage. ■

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**

Acknowledgments

Thanks to Roger Orr, Tom Hawtin, Kevlin Henney, Russel Winder and Jez Higgins for general guidance advice and patience and to Caroline Hargreaves for review.

References

[PEAA] Patterns of Enterprise Application Architecture by Martin Fowler. ISBN-13: 978-0321127426

[MVC Antipattern] The M/VC Antipattern http://www.benjaminbooth.com/tableorbooth/2005/04/m_c_v.html

[JADwMVC] Java SE Application Design With MVC http://java.sun.com/developer/technicalArticles/javase/mvc/

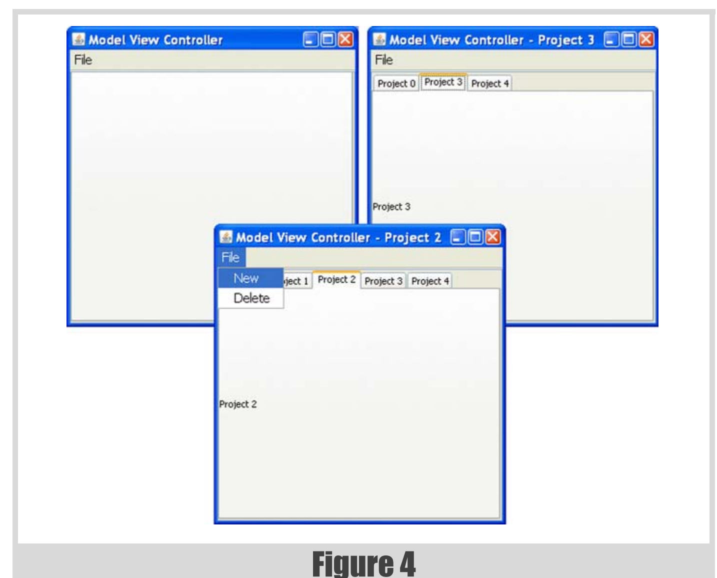


Figure 4

On Management: Understanding Who Creates Software

Software development organizations vary greatly. Allan Kelly considers what this means for managers.

When I was at school, studying for my Computer Studies GCE 'O' level I was issued with a little green book of computing terms published by the British Computer Society. Amongst other things this book described the titles and roles found in a computing department.

The book is long gone and so too are some of the titles – I've never met a 'Head of Data Processing' and don't expect to. This is shame because I find job titles more and more confusing. What one company calls a 'Development Manager' another calls a 'Project Manager', what one calls a 'Product Manager' another calls a 'Business Analyst' and so on.

Being a manager is different to being a developer. As a developer I could buy Scott Meyers' *Effective C++* and stick to his 50 rules. Yes C++, Java and C# are hard to use and there is a lot to learn but there are important differences between developing code and managing the activity.

Managers work in a far more ambiguous environment than developers. Not only are the parameters within which they work unclear and changing but the actual practice of management is ambiguous.

Neither is it clear what managers actually do. Newly promoted managers often tell me they go home at night wondering what they have done. As a developer it's possible to measure the lines of code written today, or the bugs fixed, or even the UML diagrams drawn. A bad day is one spent in meetings and discussions without any code cut. Measuring management work is more difficult.

Lesson 1: Managing software development is different to developing software and requires different skills. It is a mistake to manage people and processes in the same way as files and systems.

Few management books are of help. Books for managers often focus on a specific idea – 're-engineering', 'knowledge management' or 'outsourcing'. General management books discuss the things managers *should* be doing – thinking big thoughts, setting strategy, objectives and measuring value. In fact most days are spent in a constant round of *fire fighting* and crisis control.

Discussing 'managers' in general is not very insightful. Management roles are not equal. In this article and future articles I would like to discuss what managers do, and what they 'should' be doing. Thus, in this and future articles, I would like to look at the role of management in software development projects by looking at some of the roles managers undertake.

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.

In the same way that management roles differ, so too do organisations. Therefore, before looking at management roles it is necessary to consider organizations. Superficially similar roles, with the same title, can be very different in different types of organisation. In order to understand any given role one has to understand the organization, in order to understand the organization one has to know what types of organization there are.

Lesson 2: One size does not fit all; different types of organizations require different structures and different roles. And each individual organization will have its own unique differences. Don't try to force one company into the mould of another.

Thus this article will look at a few of the most common types of organizations that develop software. As such it will set the stage for future discussion. It is not meant to represent any kind of *best practice* but it is meant to describe a reference model.

Types of organization

Broadly speaking there are three types of organization which develop software:

- Independent Software Vendor (ISV) or Software Product Company: A company that produces and sells the same software products to customers. For example: Microsoft, Oracle, Symbian and SalesForce.com. If these companies did not produce software they would not have a business.
- Corporate IT / In house: A company that develops software to support its activities. For example: Barclays Capital, Unilever/Lever Brothers and British Airways. These companies sell a product or service that is not software but develop software to make the product or service. To keep things simple the term Corporate IT is used to include both central corporate IT functions and distributed IT activities, e.g. a bank may well have developers working for the equities trading desk.
- External Service Providers (ESP): A company that develops software for customers. For example Accenture, EDS, Infosys and Thoughtworks. Such companies may also provide additional IT services such as operations control and data centres. For some customers they may provide such services but not develop software. Most customers are corporate who need some IT services.

ISVs do on occasion contract ESPs but since their business depends on their ability to create software this is uncommon. ISVs also have internal IT needs and may contract an ESP to run aspects of the computing system, e.g. Microsoft have outsourced some of their internal support operations to an ESP.

The three categories outlines will serve for reference. This list is by no means exhaustive and new business models are constantly arising which obscures the boundaries. For example, Software as a Service (SaaS) pioneer SalesForce is included here as an ISV but their business model

Internal development groups have different objectives, roles and processes to those which produce software for sale

rests on providing a service in a similar way an ESP might. The difference is that the SaaS model offers the same software to all customers.

Other companies produce products that would not be possible without a software element but would not think of themselves as an ISV. For example the makers of digital radio sets are dependent on software but are clearly not an ISV. As more and more products contain complex software – cars, televisions, alarm clocks – more and more companies will be dependent on software for their key products.

The rest of this article will focus on the ISV and corporate IT models. This is not because ESPs are any less worthy but because ESPs usually operate either as an extension of a corporate IT (think outsourcing) or are used to provide a specific product (similar to an ISV).

One of the biggest mistakes made by young ISVs is operating as a Corporate IT department and not an ISV. Software engineering skills are largely the same inside corporate IT departments and ISVs. If you can code Java in a bank you can code Java for a company that sells software. But the same is not true at management level. Managing the creation and delivery inside a corporate requires different skills and judgements to those required to successfully manage the delivery of software products.

Lesson 3: Know what type of software producer you are, and which you are not. Understand what role your software plays in delivering the final product to the customer and generating revenue.

Internal development groups have different objectives, roles and processes to those which produce software for sale. Particularly in the UK where most IT is based insides corporates this is a common mistake. In the US were there is a longer tradition of software product companies there are more role models available to ISVs.

The reverse mistake is not so often made and is actually less dangerous. ISVs live and die by their ability to deliver software, therefore their practices need to deliver and they have a software-centric culture. The same is not true in corporate where the culture will come from the main business. If a software development project is late or fails, the business will usually carry on as before, in other words the company can afford a few IT failures.

I would like to make the very broad generalisation that ISVs tend to have better practices than corporate IT groups. Since ISVs depend on selling software in order to survive one might expect that their development practices are better than corporate IT departments. After all, if an ISV cannot create good software the business cannot continue.

However, experience shows that even poor ISVs can survive for a surprising amount of time. If they have an existing customer base, or a product that is genuinely innovative they can often scrape together enough money to continue for some time. In the extreme these companies can even trade on their poor quality by selling customer maintenance contracts and upgrades to fix faults.

Title inflation

Senior managers are now more likely to be called Directors whether they sit on the company board or not. Other managers may manage people, things, or simply organise their own time.

As in program code, the term ‘manager’ is often used as a general catchall name. The term itself implies a degree of seniority and authority. Rather than actually managing something many ‘managers’ would be better thought of as ‘specialists’. I once worked with several ‘Product Managers’ at a telecoms firm; these managers would have been better described as ‘Mobile telephone radio specialists’.

Small companies with big companies as customers tend to suffer more than most from title inflation. There is a need to appear big, to send people of equivalent ‘rank’ to meetings, so titles are often aggrandised for marketing reasons. And all companies are prone to offering title enhancements in place of financial rewards.

Such practices are harmless as long as the individuals and their co-workers don’t attach too much significance to the title. One developer of my acquaintance acquired the title ‘Chief Software Architect’. This would have been harmless enough if the individual concerned had carried on as before but with only about 10 developers the company hardly needed a Bill Gates type figure to direct the architecture. What it did need was a software engineer who understood how things hung together; helped more junior engineers design and got their hands dirty with code.

For those looking to improve the performance of their software development activities high-performing ISVs are a good place to look for practices and techniques. These techniques can then be transplanted to the corporate IT world provided account is taken of the differences.

Traditionally corporate IT departments only dealt with internal customers. If they produced a program which was difficult to use, the users had little choice but to use it. The IT department could always offer training courses, tell people to read the manual or simply refuse to support other systems. However this is no longer true and corporate IT department need to take lessons from ISVs.

For example, until ten years ago the IT department of a package holiday company only had internal users, and perhaps a few travel agents. Today they may be asked to build a website to be used directly by customers to book holidays. If the customers find the website hard to use, confusing or slow they may go elsewhere. Ten years ago, the only customers had no choice – they had to wait.

So corporate IT departments face changing times. Together with the traditional internal only systems they are used to developing and supporting they are also being asked to develop customer facing systems. These systems need a different approach and require different skills to build.

Corporate IT roles

Inside a corporation the IT department is just one more function alongside Marketing, Finance and so on. Such a group will be lead by a Director of IT or Chief Information Officer (CIO) – or, if I remember my BCS booklet,

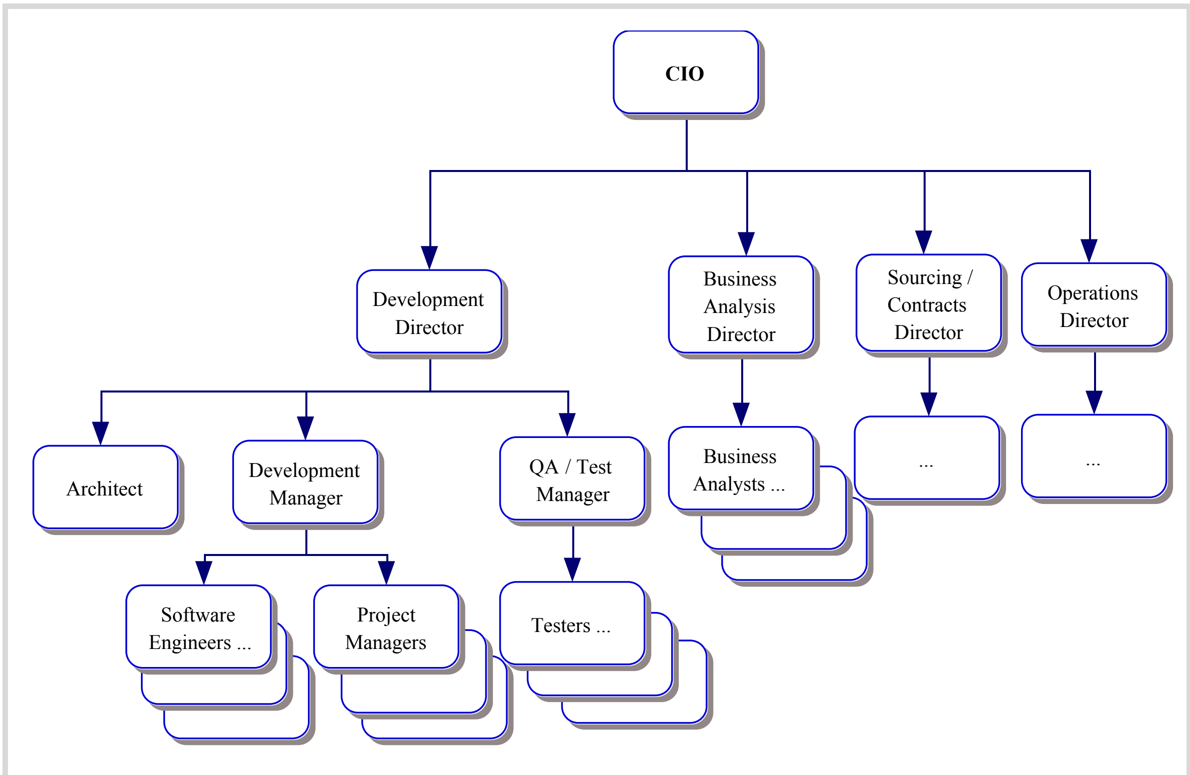


Figure 1

This description is static and shows reporting lines. Actually creating and maintaining software requires that a team is brought together from different groups. Corporate IT groups may have several software development teams – as in Figure 2. Project teams may be short-lived, lasting a matter of months, or they may last for years building and supporting the application.

As a result of this structure the actual workers – engineers, business analysts, testers, etc. – are considered a pool of resources and matrix management is common. So for example, a Software Tester would report to the Test Manager for line (or personnel) matters and professional test issues but to a Project Manager for the specific project.

the Head of Data Processing. The structure of the group may look something like Figure 1. The size of an organization will have an obvious effect on this chart. Larger organizations may have more levels and smaller companies may combine roles and groups.

Software development is only one part of the CIO's responsibilities. Quite likely there will be an operations group and maybe a business analysis group. Software Architects may report directly to the CIO or they may report to another senior manager, it all depends on the type and role of the architect.

Where an organisation uses external suppliers – ESPs, ISVs or rents data centre space – there may be a group to co-ordinate this work too.

The development of new systems is just one part of the CIO's responsibilities and would normally be headed by a senior manager such as a Development Director. Reporting to this director would be one or more development managers and one (or even more) QA/Test Managers.

On this diagram, project Managers, Testers and Business Analyst have been shown at different levels. The important point is that each of these groups exists as a group in their own right, the level at which the head of the group reports varies.

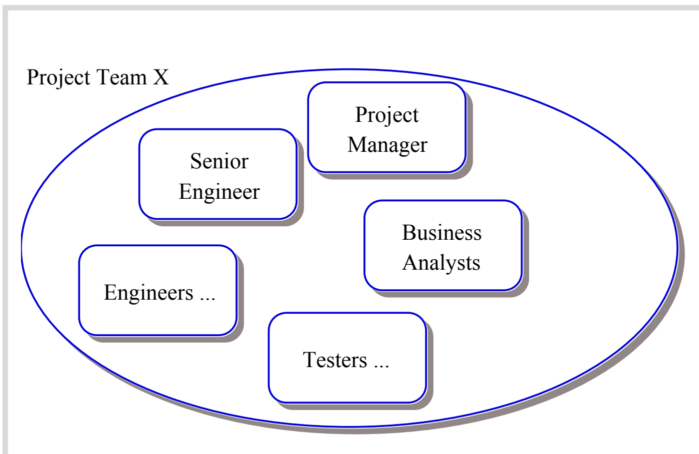


Figure 2

The three key features of this reference model are:

- CIO is head of the organizations
- Project teams are drawn from resource pools
- Matrix management

Lesson 4: Corporate IT departments exist to support a business. Software development is not the business; it is only a means to an end.

Independent software vendor

Technology companies may well have a CIO role as described above. Like all other companies – especially when they get large – there are corporate information needs, and the need for corporate IT services. However, when a company's life blood is technology itself – and specifically software vendors – there is a need for another role, the Chief Technology Officer or CTO.

While the CIO role is internally focused on processes and systems to support the business, the CTO role and the organization they lead are focused on the application of technology to create products. This creates a different organization with different roles.

Things get a little confusing when the company does not sell technology but sells a product or service that is inherently dependent on technology. For example an online retailer like Amazon or a travel company like Expedia.

A technology company has roles that don't exist – or don't exist in the same way – as a non-technology corporate and this is reflected in the corporate structure shown in Figure 3.

Some CTOs choose to take a hands-on role in managing the development department. Other CTOs define their role as architects and involve themselves directly with the development of products – even coding – while leaving it to a Development Director, or Vice-President of Engineering to organize the department. Another type of CTO concentrates their efforts in the board room and may spend most of their time making strategy or evaluating merges and acquisitions.

Lesson 5: The CTO's role is what the CTO and other senior managers choose to make it.

Companies with multiple products may have product heads that run their own organizations within organizations. Development teams and other resources working on one product may have little involvement with those working on a different product.

Traditionally, software companies that delivered software on a disc had no need of an operations department. Now when software is delivered online (as a service) there is an operations element thus there is usually a Technical Operations group ('TechOps') also reporting to the CTO.

Whether delivering on a disc or online there is support to users so there are often support desk operations. These sometimes report to the CTO but more often report elsewhere, say to sales or client services.

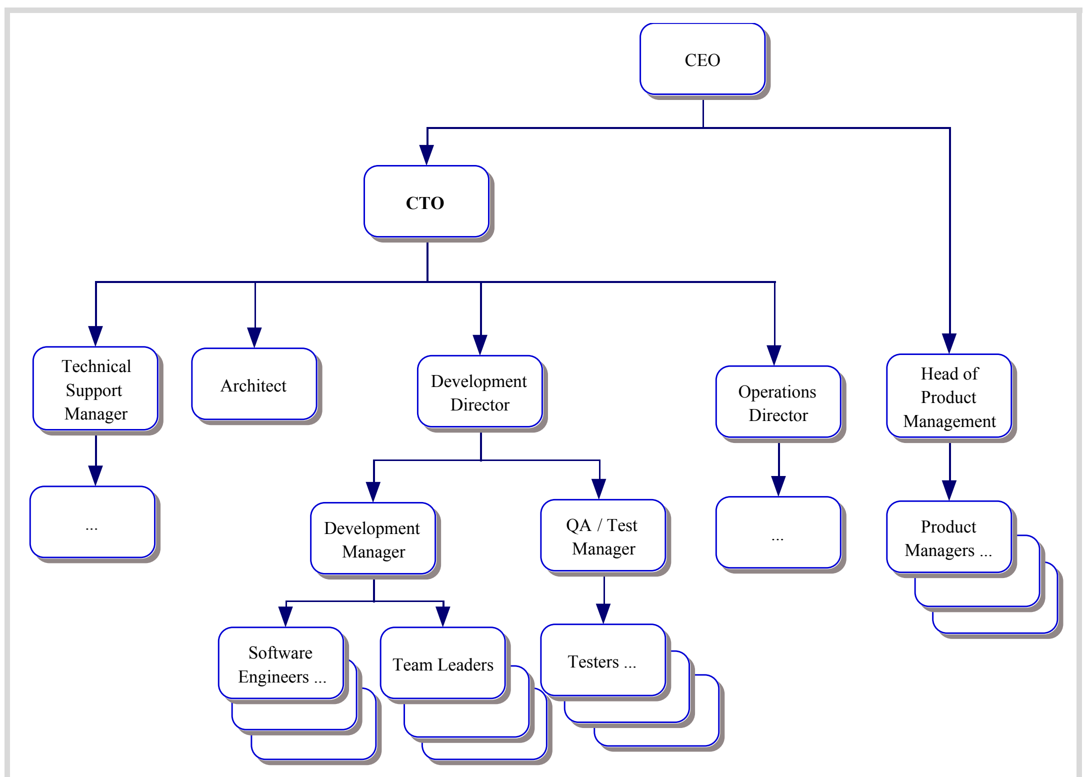


Figure 3

Perhaps the biggest difference is the replacement of Business Analysts reporting to the CIO with Product Managers reporting to the CEO. In a technology company knowing what technology products to develop is very important thus they report to the CEO – of course some companies will have them reporting to the CTO.

The Business Analyst and Product Manager roles will be discussed in future articles but for the moment it is enough to say that while the BA is inward focused, looking at systems and processes within the organization, the Product Manager is outward-focused looking at what customers want. Both roles feed the development teams with requirements and requests but they discover their requirements in a different fashion.

The inward looking nature of business analysis makes it well suited to the corporate IT world where systems are developed for internal users and to change company processes, while the customer facing nature of Product Managers makes them more suitable to ISVs.

Lesson 6: Both Product Managers and Business Analysts can create requirements for development teams. In an ISV it is typically outward-looking Product Managers who supply requirements while in corporate IT departments it is typically inward looking Business Analysts.

In recent years Scrum [Highsmith02] has popularised the term 'Product Owner'. This role decides what the development team should be working on but it does not prescribe how the decision is arrived at. Making these decisions required the skills of a Product Manager or Business Analyst.

Whatever the role is called it is important that someone is concerned with what the software will do, that someone is asking what the customer or user needs and that that person is directing the team on what should be developed and when.

Lesson 7: Software Developers should not be deciding what to develop and when to develop it. This is a separate role and should be filled by someone with the appropriate skills.

When this role is not filled explicitly there is a void. Nature abhors a vacuum and this is no exception. Sooner or later someone steps in to fill this void even when they are not explicitly tasked to do so. With luck this person is motivated to do the job, is knowledgeable about the field and has the right skills. Unfortunately it is also possible the person who steps in is not knowledgeable, has the wrong skills and has their own agenda.

Conclusion

The structures given here are examples to help discuss the role and responsibilities of management. They are also intended to highlight the difference in organizations that develop software.

There are countless variations on these models – not least those caused by culture and national differences, and business trends and fashions. The arrival of CEO – and other 'C' level – officers in British companies is relatively new. Not long ago the top person would be the Managing Director.

These models categorise development according to business model. Alternative categorisations could be by size – small or large – or according to project or product focus. As a general rule-of-thumb, ISVs are considered as product focused and corporate IT as project focused.

At a technology level – Java, Windows and such – there is often little difference between the technology company and the corporate IT department. But in terms of managing there is a world of difference. Creating systems to support a business is very different to creating systems to sell are two different tasks.

Unfortunately it is becoming harder to determine which is which. As companies come to depend on technology to deliver their products they take more of the characteristics of technology companies, but if they continue thinking like a corporate IT department the results will be disappointing at best. ■

References

[Highsmith02] Highsmith, J. 2002. *Agile Software Development Ecosystems*: Addison-Wesley.

The Legion's Revolting!

3D meshes can be too large to deal with efficiently. Stuart Golodetz applies some ancient discipline.

In my last article, I explained the first stage of the multiple material marching cubes (M3C) algorithm [Wu03] for generating polygonal meshes from labelled volumes. For my own work in medical image analysis, I make use of this algorithm to generate 3D models from a series of segmented (labelled) CT slices. The meshes produced by this first stage of the process are reasonable, but (as I noted last time) they tend to suffer from 'stair-stepping', due to the inter-slice distance being substantially greater than the intra-slice distance (the distance between the centres of adjacent pixels in a slice). They also contain far too many triangles.

I will discuss solutions to both of these problems in this article. Stair-stepping (and indeed lack of smoothness in general) can be mitigated by a technique called *Laplacian smoothing*; excessive triangle counts can be cured by a process known as *decimation*.

Stage 2: Laplacian smoothing (overview)

For single-material meshes, Laplacian smoothing is actually a remarkably simple 'averaging' process. The idea is essentially to iteratively move each mesh node a small distance towards its neighbours, using the equation:

$$x_i^{t+1} = x_i^t + \frac{\lambda}{|N(x_i)|} \sum_{x_j \in N(x_i)} (x_j^t - x_i^t)$$

In other words, the position of x_i at iteration $t+1$ is calculated by adding offset vectors in the directions of each of its adjacent nodes in $N(x_i)$. The following things are worth noting:

- λ is known as the *relaxation factor* and can be set by the user: it is usually defined to be somewhere between 0 and 1
- The adjacent nodes don't change between iterations, hence we can write $N(x_i)$ rather than $N(x_i^t)$
- The equation in [Wu03] misses the division by $|N(x_i)|$: it turns out to be important

Smoothing multiple-material meshes is a similar process, but with the difference that in this case we only smooth certain nodes in order to preserve the overall topology of the mesh. Nodes can be classified into three types:

- *Simple* nodes are those that have exactly two material IDs
- *Edge* nodes are those that have three or more material IDs and are adjacent to exactly two nodes with at least those same material IDs
- *Corner* nodes are those that have three or more material IDs and are not edge nodes

Having classified the nodes in the mesh, we then smooth the three types differently. *Simple* nodes are smoothed straightforwardly using the single-material equation given above. *Edge* nodes (in my implementation) are smoothed similarly, but using a neighbour set consisting of only the two nodes with at least the material IDs of the node being smoothed: note that this differs from the method described in [Wu03], where they restrict edge nodes to motion along the edge path (I tried both, and noticed very little difference in practice). *Corner* nodes are topologically important and thus not subjected to smoothing at all.

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

Stage 2: Laplacian smoothing (implementation)

Implementing Laplacian smoothing in C++ is quite straightforward. The process is initialised with a value for lambda and an iteration count, and the `iterate()` method is then called the specified number of times on the mesh (see Listing 1, which performs a single iteration of Laplacian smoothing). Each iteration proceeds by classifying each node and then smoothing it according to its classification; all the new positions are copied across en-masse at the end of the iteration because the old positions are required for the smoothing process. The function for actually classifying each node as a simple, edge or corner node is shown in Listing 2.

Stage 3: Mesh decimation (overview)

In Ancient Roman times, decimation was the barbaric punishment occasionally meted out to rebellious troops, or those who were considered to have shown cowardice in the face of the enemy. A military unit selected for decimation would have its number reduced by a tenth: one in ten men would be selected by lot and clubbed to death by his comrades. The idea was to terrify the remaining troops into obeying orders and fighting courageously: needless to say, however, the process more often reduced morale than increased it. In the case of modern computing, however, decimation is a much more benign procedure. Here, decimation refers to the process of selectively removing triangles and nodes from a mesh until the triangle count has been reduced to the desired level. (The triangle count can often be reduced by as much as 80-90% without greatly affecting the appearance of the mesh!).

There are various different algorithms for this, but the one referred to in [Wu03] works as follows: first of all, a decimation metric is calculated for each simple or edge node that indicates how good a candidate it is for removal from the mesh. For simple nodes, the metric is the perpendicular distance of the node from the average plane of its surrounding loop of adjacent nodes (I described how to calculate the average plane in [Golodetz08]). For an edge node, it is the perpendicular distance from the line joining the two adjacent nodes which have at least the same material IDs [Wu03], but I will only focus on simple nodes for the purposes of this article.

All the nodes are then placed into a 'priority queue' that supports post-insertion updates of element keys (note that `std::priority_queue` is not such a data structure); the decimation metric for each node is used as its key, so that the first node to be extracted from the queue will be a 'most suitable' candidate for decimation (I say 'a' rather than 'the' because several nodes can be equally good candidates).

```
template <typename Label>
void LaplacianSmoother<Label>::iterate(
   const Mesh_Ptr& mesh) const
{
  NodeVector& nodes = mesh->writeable_nodes();
  int nodeCount = static_cast<int>(nodes.size());
  std::vector<Vector3d> newPositions(nodeCount);
  // Calculate the new node positions. Note that
  // they have to be stored separately since we
  // need the old node positions in order to
  // calculate them.
  for(int i=0; i<nodeCount; ++i)
  {
  // holds the neighbours which might affect a
  // node (depends on the node type)
    std::vector<int> neighbours;
    // Determine the type of node: this affects
    // how the node is allowed to move.
    NodeType nodeType = classify_node(
       i, nodes, neighbours);
    newPositions[i] = nodes[i].position;
    switch(nodeType)
    {
      case SIMPLE:
      case EDGE:
      {
        for(
           std::vector<int>::const_iterator
           jt=neighbours.begin(),
           jend=neighbours.end(); jt!=jend; ++jt)
        {
          Vector3d offset =
             nodes[*jt].position -
             nodes[i].position;
          offset *= m_lambda / neighbours.size();
          newPositions[i] += offset;
        }
        break;
      }
      case CORNER:
      {
        // Corner nodes are topologically
        // important and must stay fixed.
        break;
      }
    }
  }
  // Copy them across to the mesh.
  for(int i=0; i<nodeCount; ++i)
  {
    nodes[i].position = newPositions[i];
  }
}
```

Listing 1

Nodes are then iteratively extracted from the queue and the local mesh around them decimated, until some user-defined reduction threshold is reached. In the case of simple nodes, this local decimation process consists of removing the node in question, and all triangles that use it, and retriangulating the loop of adjacent nodes around it using the Schroeder triangulation process introduced in [Golodetz08]. (It is important to note that the adjacent node loop will generally be non-planar, which necessitates a more complicated triangulation scheme than might otherwise be necessary.) The metrics of adjacent nodes are then recalculated, which may cause them to move around in the priority queue.

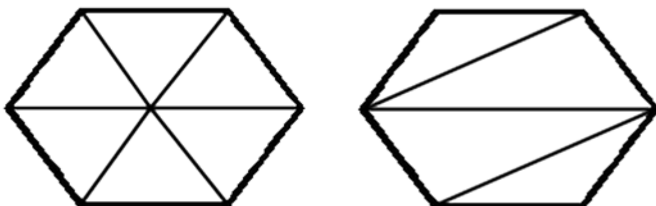


Figure 1

```
template <typename Label>
NodeType classify_node(int i,
   const std::vector< Node<Label> >& nodes,
   std::vector<int>& neighbours)
{
  NodeType nodeType = SIMPLE;
  if(nodes[i].labels.size() == 2)
  {
    // We're dealing with a simple node.
    neighbours.swap(
       std::vector<int>(
       nodes[i].adjacentNodes.begin(),
       nodes[i].adjacentNodes.end()));
  }
  else
  {
    // Count the number of adjacent nodes with at
    // least the same labels as this one. If it's
    // equal to two, we're dealing with an edge
    // node. Otherwise, we have a corner.
    int edgeCriterion = 0;
    for(std::set<int>::const_iterator
       jt=nodes[i].adjacentNodes.begin(),
       jend=nodes[i].adjacentNodes.end();
       jt!=jend; ++jt)
    {
      std::set<int> commonLabels;
      std::set_intersection(
         nodes[i].labels.begin(),
         nodes[i].labels.end(),
         nodes[*jt].labels.begin(),
         nodes[*jt].labels.end(),
         std::inserter(commonLabels,
         commonLabels.begin()));
      if(commonLabels.size() ==
         nodes[i].labels.size())
      {
        ++edgeCriterion;
        neighbours.push_back(*jt);
      }
    }
    if(edgeCriterion == 2) nodeType = EDGE;
    else nodeType = CORNER;
  }
  return nodeType;
}
```

Listing 2

This process reduces both the triangle and node counts of the mesh. The node count obviously decreases by 1, since a single node has been removed. The triangle count decreases by 2: consider the example of meshing a hexagaon with a central node (6 triangles) compared to one without (4 triangles): see Figure 1.

Stage 3: Mesh decimation (implementation)

The implementation of mesh decimation relies on a peculiar type of priority queue that supports the updating of priorities while elements are still in the queue (see Listing 3). The priority queue itself is represented as a heap (as usual); the only difference is that we also maintain a dictionary to allow elements in the heap to be referenced and their keys modified. In our case, we use each node’s global ID as the ID, its decimation metric as

```
template <typename ID, typename Key,
   typename Data, typename Comp = std::less<Key> >
class PriorityQueue
{
public:
  class Element
  {
  private:
    Data m_data;
    ID m_id;
    Key m_key;
  public:
    Element() {}
    Element(const ID& id, const Key& key,
       const Data& data) : m_id(id), m_key(key),
       m_data(data) {}
    Data& data()             { return m_data; }
    const ID& id() const     { return m_id; }
    const Key& key() const   { return m_key; }
    friend class PriorityQueue;
  };
private:
  // The dictionary maps IDs to their current
  // position in the heap
  typedef std::map<ID,size_t> Dictionary;
  typedef std::vector<Element> Heap;
  // Datatype Invariant: m_dictionary.size()
  // == m_heap.size()
  Dictionary m_dictionary;
  Heap m_heap;
public:
  void clear()
  {
    m_dictionary.clear();
    m_heap.swap(Heap());
  }
  bool contains(const ID& id) const
  {
    return m_dictionary.find(id) !=
       m_dictionary.end();
  }
  Element& element(const ID& id)
  {
    return m_heap[m_dictionary[id]];
  }
  bool empty() const
  {
    return m_dictionary.empty();
  }
  void erase(const ID& id)
  {
    size_t i = m_dictionary[id];
    m_dictionary.erase(id);
    m_heap[i] = m_heap[m_heap.size()-1];
    if(m_heap[i].id() != id)  // assuming the
    // element we were erasing wasn't the last one
    // in the heap, update the dictionary
    {
      m_dictionary[m_heap[i].id()] = i;
    }
    m_heap.pop_back();
    heapify(i);
  }
  void insert(const ID& id, const Key& key,
     const Data& data)
  {
    if(contains(id))
    {
```

Listing 3

```
      throw Exception("An element with the
         specified ID is already in the priority
         queue");
    }
    size_t i = m_heap.size();
    m_heap.resize(i+1);
    while(i > 0 && Comp()(key,
       m_heap[parent(i)].key()))
    {
      size_t p = parent(i);
      m_heap[i] = m_heap[p];
      m_dictionary[m_heap[i].id()] = i;
      i = p;
    }
    m_heap[i] = Element(id, key, data);
    m_dictionary[id] = i;
  }
  void pop()
  {
    erase(m_heap[0].id());
    ensure_invariant();
  }
  Element top()
  {
    return m_heap[0];
  }
  void update_key(const ID& id, const Key& key)
  {
    size_t i = m_dictionary[id];
    update_key_at(i, key);
  }
private:
  void heapify(size_t i)
  {
    bool done = false;
    while(!done)
    {
      size_t L = left(i), R = right(i);
      size_t largest = i;
      if(L < m_heap.size() &&
         Comp()(m_heap[L].key(),
         m_heap[largest].key()))
        largest = L;
      if(R < m_heap.size() &&
         Comp()(m_heap[R].key(),
         m_heap[largest].key()))
        largest = R;
      if(largest != i)
      {
        std::swap(m_heap[i], m_heap[largest]);
        m_dictionary[m_heap[i].id()] = i;
        m_dictionary[m_heap[largest].id()] =
           largest;
        i = largest;
      }
      else done = true;
    }
  }
  static size_t left(size_t i) { return 2*i + 1; }
  static size_t parent(size_t i)
  {
    // Precondition: i > 0
    return (i+1)/2 - 1;
  }
  void percolate(size_t i)
  {
    while(i > 0 && Comp()(m_heap[i].key(),
       m_heap[parent(i)].key()))
```

Listing 3 (cont'd)

```
    {
      size_t p = parent(i);
      std::swap(m_heap[i], m_heap[p]);
      m_dictionary[m_heap[i].id()] = i;
      m_dictionary[m_heap[p].id()] = p;
      i = p;
    }
  }
  static size_t right(size_t i){ return 2*i + 2; }
  void update_key_at(size_t i, const Key& key)
  {
    if(Comp()(key, m_heap[i].key()))
    {
      // The key has increased.
      m_heap[i].m_key = key;
      percolate(i);
    }
    else if(Comp()(m_heap[i].key(), key))
    {
      // The key has decreased.
      m_heap[i].m_key = key;
      heapify(i);
    }
  }
};
```

Listing 3 (cont'd)

```
template <typename Label>
typename MeshDecimator<Label>::Mesh_Ptr
MeshDecimator<Label>::operator()(
   const Mesh_Ptr& mesh) const
{
  m_trisToRemove = static_cast<int>(
     m_reductionTarget *
     mesh->triangles()->size());
  m_trisRemoved = 0;
  PriQ pq;
  construct_priority_queue(pq, mesh);
  while(!pq.empty() &&
     m_trisRemoved < m_trisToRemove)
  {
    PriQ::Element e = pq.top();
    pq.pop();
    std::list<TriangleL> tris =
       e.data()->decimate();
    int triCount = static_cast<int>(tris.size());
    if(triCount > 0)
    {
      int index = e.data()->index();
      NodeVector& nodes = mesh->writeable_nodes();
      NodeL& n = nodes[index];
      m_trisRemoved += static_cast<int>(
         n.adjacentNodes.size()) - triCount;
     // Mark the node as no longer valid and remove
    // references to it from the surrounding nodes.
      n.valid = false;
      for(std::set<int>::const_iterator it=
         n.adjacentNodes.begin(),
         iend=n.adjacentNodes.end(); it!=iend;
         ++it)
      {
        nodes[*it].adjacentNodes.erase(index);
      }
      // Add any new edges introduced by the
      // retriangulation.
      for(std::list<TriangleL>::
         const_iterator it=tris.begin(),
         iend=tris.end(); it!=iend; ++it)
```

Listing 4

```
      {
        int i0 = it->indices[0],
           i1 = it->indices[1],
           i2 = it->indices[2];
        NodeL& n0 = nodes[i0];
        NodeL& n1 = nodes[i1];
        NodeL& n2 = nodes[i2];
        n0.adjacentNodes.insert(i1);
        n0.adjacentNodes.insert(i2);
        n1.adjacentNodes.insert(i0);
        n1.adjacentNodes.insert(i2);
        n2.adjacentNodes.insert(i0);
        n2.adjacentNodes.insert(i1);
      }
      // Splice the new triangles onto the end of
      // the triangle list.
      TriangleList& meshTriangles =
         mesh->writeable_triangles();
      meshTriangles.splice(meshTriangles.end(),
         tris);
     // Recalculate the metrics for the surrounding
     // nodes and update their keys in the priority
      // queue.
      for(std::set<int>::const_iterator it=
         n.adjacentNodes.begin(),
         iend=n.adjacentNodes.end(); it!=iend;
         ++it)
      {
        if(pq.contains(*it))
        {
          PriQ::Element& adj = pq.element(*it);
          adj.data()->calculate_details();
          if(adj.data()->valid()) pq.update_key(
             *it, adj.data()->metric());
          else pq.erase(*it);
        }
      }
    }
  }
  clean_triangle_list(mesh);
  rebuild_node_array(mesh);
  return mesh;
}

template <typename Label>
void MeshDecimator<Label>::clean_triangle_list(
   const Mesh_Ptr& mesh) const
{
  const NodeVector& nodes = *(mesh->nodes());
  TriangleList& triangles =
     mesh->writeable_triangles();
  for(TriangleList::iterator it=triangles.begin(),
     iend=triangles.end(); it!=iend;)
  {
    int i0 = it->indices[0], i1 = it->indices[1],
       i2 = it->indices[2];
    if(!nodes[i0].valid || !nodes[i1].valid ||
       !nodes[i2].valid)
    {
      it = triangles.erase(it);
    }
    else ++it;
  }
}

template <typename Label>
void MeshDecimator<Label>::
   construct_priority_queue(PriQ& pq,
   const Mesh_Ptr& mesh) const
```

Listing 4 (cont'd)

```
{
  const NodeVector& nodes = *(mesh->nodes());
  int nodeCount = static_cast<int>(nodes.size());
  for(int i=0; i<nodeCount; ++i)
  {
    std::vector<int> neighbours;  // holds the
   // neighbours which might affect a node (depends
    // on the node type)
    NodeType nodeType = classify_node(
       i, nodes, neighbours);
    switch(nodeType)
    {
      case SIMPLE:
      {
        NodeDecimator_Ptr nodeDecimator(
           new SimpleNodeDecimator(i, mesh, pq));
        if(nodeDecimator->valid()) pq.insert(i,
           nodeDecimator->metric(),
           nodeDecimator);
        break;
      }
      case EDGE:
      {
        //...
        break;
      }
      case CORNER:
      {
        // Corner nodes should not be decimated.
        break;
      }
    }
  }
}
template <typename Label>
void MeshDecimator<Label>::rebuild_node_array(
   const Mesh_Ptr& mesh) const
{
  NodeVector& nodes = mesh->writeable_nodes();
  TriangleList& triangles =
     mesh->writeable_triangles();
  int nodeCount = static_cast<int>(nodes.size());
  // Rebuild the node array.
  std::vector<int> mapping(nodeCount, -1);
  NodeVector newNodes;
  for(int i=0; i<nodeCount; ++i)
  {
    if(nodes[i].valid)
    {
      mapping[i] =
         static_cast<int>(newNodes.size());
      newNodes.push_back(nodes[i]);
    }
  }
  nodes = newNodes;
  // Update the indices in the adjacent node sets
  // using the mapping.
  nodeCount = static_cast<int>(nodes.size());
  for(int i=0; i<nodeCount; ++i)
  {
    std::vector<int> adjacentNodes(
       nodes[i].adjacentNodes.begin(),
       nodes[i].adjacentNodes.end());
    int adjacentNodeCount =
       static_cast<int>(adjacentNodes.size());
    for(int j=0; j<adjacentNodeCount; ++j)
    {
      adjacentNodes[j] =
         mapping[adjacentNodes[j]];
```

Listing 4 (cont'd)

Figure 2

the key, and store a mesh decimation functor (to perform the local decimation work) as the data payload.

The key part of the decimation code, then, is shown in Listing 4. It works as follows: provided there are still undecimated nodes remaining, and we haven't yet reached our reduction target, nodes are repeatedly extracted from the priority queue and the mesh around them decimated. This decimation is handled in practice by marking the node to be decimated as invalid and retriangulating the node loop around it. Some book-keeping is necessary to update the various mesh data structures, after which the new triangles are added to the mesh and the metrics for the surrounding nodes are recalculated. At the end of the process, the triangle list is cleaned by removing any triangles which mention invalid nodes. The node array is then cleaned by removing any nodes which are invalid: this involves renumbering all the remaining nodes, so a map is created from old to new indices, and this is used to update the triangle list accordingly.

Figure 2 shows the end result.

Summary

In this article, we have seen how to smooth and decimate the intermediate meshes produced by the first stage of the multiple material marching cubes (M3C) algorithm [Wu03]. This allows us to generate smooth, reasonably-sized meshes that are suitable for real-time visualization. In the medical imaging domain, these meshes can be used to allow doctors to view a patient's state from any angle (and indeed to 'fly around' the patient's model), greatly aiding their understanding of the 3D situation involved. ■

References

[Golodetz08] Golodetz, SM, Seeing Things Differently, Overload 86, August 2008.

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


```
      }
      nodes[i].adjacentNodes =
         std::set<int>(adjacentNodes.begin(),
         adjacentNodes.end());
    }
    // Update the indices in the triangle list using
    // the mapping.
    for(TriangleList::iterator it=triangles.begin(),
       iend=triangles.end(); it!=iend; ++it)
    {
      for(int j=0; j<3; ++j)
      {
        it->indices[j] = mapping[it->indices[j]];
      }
    }
}
```

Listing 4 (cont'd)

Iterators and Memberspaces

Exposing a compound object's collections can be messy. Roel Vanhout introduces a powerful idiom.

This article describes a C++ idiom to expose an object's collection member variables in a way that is easy to use and read for users of the object. It allows users to iterate over filtered and/or modified elements of a collection without the user having to be aware of the implementation or type of the collection. It describes the problem, several possible solutions and their drawbacks and proposes a technique that uses the 'memberspaces' idiom and the **`Boost.Iterator`** library to provide object designers with a mechanism to allow users of their objects flexible access to the data of those objects.

It contains introductions to the memberspace idiom and Boost.Iterator's **`filter_iterator<>`** and **`transform_iterator<>`**.

Introduction

A common scenario in object-oriented development is that an object has a collection (for example a **`std::vector<>`**) as a member that the object's author wants to expose in the public interface of the object. Consider an object like this one:

```
class A {
  protected:
    std::vector<int> values;
};
```

This object has a member with a collection in it (values) in which **`int`**s can be stored. As far as the internal implementation goes, this works fine; however when the contents of values need to be exposed, dilemmas arise. There are several solutions to most of those dilemmas, all with their advantages and drawbacks.

Potential solutions

The first solution is to expose the collection itself directly:

```
std::vector<int>& getValues();
```

or (depending on whether read/write or read-only access is appropriate):

```
const std::vector<int>& getValues();
```

The disadvantages to this are clear: when the type of the collection or of its contents changes internally, users will have to change their code as well. Secondly, there is no granularity in what users of the object can do to the collection. When exposing the non-const version above, users can change the contents of the vector but also its size. There is no way to limit them to do just one of those two, nor does it have a mechanism to do validation of the vector's contents after the user performed an operation on it.

A second solution is to expose functions to query and modify the collection:

```
size_t getNValues() const {
   return values.size(); }
int getValue(int index) const {
   return values[index]; }
void setValue(int index, int new_value) {
   values[index] = new_value;
}
```

This solves the encapsulation issue of the first solution, and provides a way for the object author to put in validation of the input. On the other hand, it's inconvenient to use for both the author and the user of the object because there are so many ways to name and implement this solution. Although it is possible to enforce naming standards within the same class library (but even within a team this gets harder as the team grows and the codebase ages), invariably at one point differences in naming will emerge (**`getNumValues()`**, **`GetValuesCount()`**, …) and users of the object will have to start refering to the documentation or header file of an object every time they want to use it; not to mention the inevitable differences in conventions between different libraries. Worse than this issue, users cannot use stl-style generic algorithms on collections that are exposed this way.

A third solution is to expose the stl interface of the collection directly:

```
typedef std::vector<int>::iterator iterator;
typedef std::vector<int>::const_iterator
   const_iterator;
iterator begin() { return values.begin(); }
iterator end() { return values.end(); }
```

This solution is starting to look better but is still not without its flaws. First, it makes the object directly 'iterable', which may not be intuitive (depending on the relationship between the object and its collection). Furthermore it restricts the number of exposed collections to one. Thirdly, it re-introduces the problem of the first solution of having no granularity in the access to the collection: users can for example change values and assign any value to the elements of the collection, without regards of whether or not those values make sense in the context of the object. As long as those values remain compatible with the collection's data type, they can do anything they want with them. Lastly, it relies on the collection that is being exposed to already have an stl-compatible interface with functions that provide iterators.

The goal

The ideal solution to the described problem has the following properties:

- It exposes iterators so collections can be used by generic, stl-compliant algorithms.
- It can be used to expose all sorts of collections, not only stl collections.
- It allows multiple collections to be exposed by an object.

Roel Vanhout Roes Vanhout is a software developer at a small company in the Netherlands that develops and commercializes Spatial Decision Support Systems. He spends most of his time in C++ but likes to reminisce about his times with PHP too. He can be contacted at roel.vanhout@gmail.com

A problem arises when you want to access the variables of the class in which a member space is contained

- It's intuitive and idiomatic: it allows client code to express clearly what is happening.
- It provides object authors with finely grained control over the way the collection is accessed or modified.

Ideally, client code could look like this:

```
Classroom c;
BOOST_FOREACH (Desk d, c.Desks) {
  // do something with Desk d here
}
BOOST_FOREACH (Student s,
   c.StudentsInClassroomRightNow) {
  // do something with Student s here
}
```

This code uses the boost `Foreach` construct that wraps iterators to allow direct iteration over stl-compliant containers; the corresponding way to do this directly with iterators is:

```
typedef Classroom::Desks::iterator TDeskIter;
for (TDeskIter it = c.Desks.begin() ;
     it != c.Desks.end() ;
     it++) {
    Desk d = *it;
    // do something with Desk d here
}
```

Fortunately there is a way to make this happen. By combining the freely available Iterator library from Boost and using an idiom called 'memberspaces', the code above can be made into compilable and working C++ code.

Memberspaces and two techniques for implementing them

Memberspaces (a name I first found mentioned in [LópezMuñoz02]) are essentially a technique to subdivide the 'space' in which the member variables of an object live into separately named 'subspaces', like namespaces do for freestanding functions and classes. They are implemented as member structs, hence the name 'memberspace'. When used with only variables (i.e., not member functions), they look like this[1]:

```
class PirateShip {
  struct Deck {
    int nCannons;
  }
  struct UpperDeck : public Deck {
  } UpperDeck;
  struct LowerDeck : public Deck {
  } LowerDeck;
}
```

This allows users of the `pirateShip` class to write code like this:

```
PirateShip adventure_galley;
…
std::cout << "There are "
         <<  adventure_galley.UpperDeck.nCannons
          << " cannons on the upper deck and "
          << adventure_galley.LowerDeck.nCannons
          << " cannons on the lower deck."
          << std::endl;
```

Note the naming convention. In this article I use a style where class names are in CamelCase and variables are in all lowercase with underscores. In the `PirateShip` class above though, `UpperDeck` and `LowerDeck` are variables; why are they not in lowercase with underscores? The reason is in their use. They are not used as 'regular' member variables, but as pseudo-namespaces; therefore they use the naming convention of namespaces, which is CamelCase.

Also note that the class name and the memberspace (variable) name are the same (`UpperDeck` and `LowerDeck`). This is legal and although it doesn't serve an explicit purpose, I find it an easy way to recognize memberspaces in the code. It also prevents you from having to make up another name for the class: it could be called `UpperDeckT`, `UpperDeckClass`, `UpperDeck_`, … but by making it be the same as the memberspace variable itself, it is easy to be consistent across your project.

After a little getting used to, this is a very readable and natural way of working with the properties of an object.

A problem arises when you want to access the variables of the class in which a member space is contained. If you wanted to implement member functions in the `Deck` class and for that wanted access to variables of the `PirateShip` class, you'd need a way to get to those variables. There are two easy ways to implement this, one of which I've coined 'non-intrusive' because it doesn't require changing the containing class, the other one 'intrusive' because it does.

'Non-intrusive'

To write a memberspace that can handle access to the containing class on its own, without any help from the containing class, you can use some pointer arithmetic to calculate the start address of the containing class and cast that to the correct type. To calculate this start address, you take the address of the memberspace struct and deduct from it the offset of the memberspace struct within the containing class – which you can get with the `offsetof()` macro.

1. History buffs will shake their heads at this example, as pirates (the sea-faring type) generally preferred the highly manoeuvrable schooners or sloops, which typically only had one level of artillery. In this article, it is assumed that the `PirateShip` class is part of a highly extensible pirate-modelling framework that allows for flexible configuration of pirate paraphernalia and puts responsibility for historical accuracy in the hands of the modeller.

you'll have to check yourself whether this technique falls within the 'acceptable' bounds of the coding standards of your environment

The drawback of this technique is that it relies on potentially unportable assumptions, like `offsetof()` being implemented in a way that it accounts for things like virtual functions. More specifically, the standard requires `offsetof()` to work only on POD types. This (roughly) means that for classes you are restricted to those that are 'struct-like': no virtual functions, no constructor or destructor and only public functions. For more information.

Recent `offsetof()` implementations of the Visual Studio compiler and gcc work also in (some) cases not covered by the standard, but if you require guaranteed portability this technique is not the right one to use.

This technique may give others reviewing your code the jitters because it uses `reinterpret_cast<>`, a construct that is often indicative of code of highly questionable quality. In my opinion the case presented here is one of the few cases in which it is ok to use `reinterpret_cast<>`, but you'll have to check yourself whether this technique falls within the 'acceptable' bounds of the coding standards of your environment.

An example is shown in Listing 1.

```
class A {
  public:
    void doSomething() {}
    struct MyMemberSpace {
    public:
        void someFunction()
        {
            owner().doSomething();
        }
    private:
        A& owner() {
            return *reinterpret_cast<A*>(
                reinterpret_cast<char*>(this) -
                offsetof(A, MyMemberSpace));
        }
    } MyMemberSpace;
}

A a;
a.MyMemberSpace.someFunction();
```

Listing 1

'Intrusive'

A more portable way is to keep a reference to the containing object in a member variable and let the containing object initialize it in its constructor. This is most easily explained with a code example (Listing 2).

The main disadvantage of this technique is that it requires a variable in the memberspace – adding to the size of the object. When the first technique is used, no such member is needed and no price in space overhead needs to be paid. There will be few circumstances in which this overhead is prohibitive, though.

```
class A {
    A() : MyMemberSpace(*this) {}
    void doSomething() {}
    struct MyMemberSpace {
        void someFunction() {
            m_A.doSomething();
        }
    private:
        friend A;
        MyMemberSpace(A& a) : my_A(a) {}
        A& my_A;
    } MyMemberSpace;
}
A a;
a.MyMemberSpace.someFunction();
```

Listing 2

Exposing a collection through a memberspace

We've already fulfilled one of the original design goals we started with: we have a way to expose multiple collections, through multiple memberspaces that all have their own name. The next thing we want to do is make the memberspaces iterable, that is, make them compliant with stl-like containers. Luckily, it is easy to make them work for the most basic cases. All that is required is that we provide `begin()` and `end()` functions and an iterator `typedef` that indicates the type of iterator that is returned by those functions. If the collection we are exposing is already an stl-compatible container, we can use the iterator type and `begin()` and `end()` iterator accessors to implement our own function. A simple example is shown in Listing 3.

```
class A {
  public:
    A() : MyMemberSpace(*this) {}
    typedef std::vector<int> myCollection;
    myCollection collection;
    struct MyMemberSpace {
        friend A;
        typedef myCollection::iterator iterator;
        iterator begin() { return
           m_A.collection.begin(); }
        iterator end() { return
           m_A.collection.end(); }

    private:
        MyMemberSpace(A& a)
            : m_A(a)
        {}
        A& m_A;
    } MyMemberSpace;
};
```

Listing 3

The technique ... works fine as long as the only thing you want to do is to directly allow all elements of the container to be visible from the outside

This can now be used like this:

```
A a;
BOOST_FOREACH (int i, a.MyMemberSpace) {
  // do something with int i here
}
```

Exposing a filtered collection through a memberspace

The technique explained above works fine as long as the only thing you want to do is to directly allow all elements of the container to be visible from the outside. But there are scenarios in which you want to provide an easy way for clients to go through a subset of the elements in the container – either as an addition to or as a replacement of the access to all elements. Consider a class like the following:

```
struct Employee {
  enum ESeniority { E_SENIOR, E_JUNIOR };
  std::string name;
  ESeniority seniority;
};
class Staff {

protected:
  std::vector<Employee> employees;
};
```

If we could find a way to expose an iterator over employees that could leave out (or 'filter') certain elements that don't fulfill a certain criterion, we could use that iterator to go over all those that do fulfill it. Such an iterator is called a 'filtering iterator'. The design of one is described in detail in [Higgins08].

Here however a filtering iterator is realized through the freely available Boost.Iterator [Abrahams03] library. This library provides, roughly speaking, templates to easily implement new iterators and to modify existing ones. By using the `filter_iterator<>` template, we can modify the behaviour of the iterators that we can get from `std::vector<>` so that for every element in it, a function is called: if that function returns `true`, the element is included when advancing the iterator; otherwise it is skipped. This template provides us with a tool to easily implement iterators for the memberspaces in the example above (see Listing 4).

Here is what is happening. First, we define an object that will serve as the 'decision-making point' for whether or not a certain `Employee` will be considered part of the `SeniorEmployees`. We can determine this by testing the seniority member variable of the object. The object needs to be callable (have the `()` operator defined) and should take the type of the elements of the collection we want to iterate over – in this example, an `Employee`.

For the implementation of the memberspace proper, we start with providing a `typedef` for the iterator type that will be exposed. We can use the type of the `boost::filter_iterator` template for this. It takes two template arguments: the object that will provide the 'decision-making' functionality and the iterator that is being 'wrapped' or modified. Since we're exposing an stl collection, we can re-use the iterator type from the `std::vector<>`. Although not strictly needed, we also provide a `const_iterator` – it is needed by the `BOOST_FOREACH` construct that I'm rather fond of. Many algorithms use the `const_iterator` so it's good practice to always define it.

Next, we come to the implementation of the `begin()` and `end()` functions which should return the actual iterators. Boost.Iterator provides a utility function to make iterators of the `filter_iterator<>` type. It is (unsurprisingly) called `make_filter_iterator` and takes three arguments, of which one takes the form of a template parameter. This template parameter is the type of the class that will decide whether or not to include a certain element in the collection to be included when the iterator is used. The two 'normal' arguments that are passed to the

```
#include <boost/iterator/filter_iterator.hpp>
struct isSeniorEmployee {
  bool operator()(Employee& e) {
    return e.seniority == Employee::SENIOR;
  }
};
struct SeniorEmployees {
  typedef
     boost::filter_iterator<isSeniorEmployee,
     std::vector<Employee>::iterator> iterator;
  typedef
     boost::filter_iterator<isSeniorEmployee,
     std::vector<Employee>::const_iterator>
     const_iterator;
  iterator begin() {
    return boost::make_filter_iterator<
       isSeniorEmployee>(
       owner().employees.begin(),
       owner().employees.end()
    );
  }
  iterator end() {
    return boost::make_filter_iterator<
       isSeniorEmployee>(
       owner().employees.end(),
       owner().employees.end()
    );
  }
private:
  Staff& owner() {
    return *reinterpret_cast<Staff*>(
        reinterpret_cast<char*>(this)
        - offsetof(Staff, SeniorEmployees));
  }
} SeniorEmployees;
```

Listing 4

constructor are a range of elements, embodied by two iterators. The `end()` iterator is needed because the filter iterator needs to know when to stop going over the elements of the source collection.

The result is a construct where we can iterate over only the senior employees by writing:

```
Staff s;
BOOST_FOREACH(Employee e, s.SeniorEmployees) {
  std::cout << e.name << std::endl;
}
```

Exposing a modified collection through a memberspace

The previous section covered modifying the appearance of a collection by filtering out certain elements. We can also use iterators to modify not the collection, but rather the individual elements of a collection. The crucial element to do so is another iterator adapter: boost's `transform_iterator<>`.

Let's add another potentially useful memberspace to the `Staff` class introduced above:

```
struct EmployeeNames {
} EmployeeNames;
```

To get access to only the names of the employees (and to be able to iterate over them), we have to get access to the 'name' member variable of every `Employee` object. The `transform_iterator<>` mentioned earlier will apply a user-defined operation on an object to transform it into a different type. In the `EmployeeNames` example, what we want to do is 'transform' an `Employee` struct to its name, in `std::string` form. The 'transformer' struct to do this looks very similar to the struct that determined whether or not to include an object in an exposed collection. The actual conversion is done in the `()` operator, which should return the type to be transformed to and takes as a reference the object to be transformed. The code is shown in Listing 5.

The `begin()` and `end()` iterators work in the same way as those for the `filter_iterator`. You use the `make_transform_iterator` utility function provided by the library, which takes the beginning of the iterator range you want to transform as an input. You don't need to specify an end this time as the `transform_iterator` iterates over all the elements. The second argument is an instance of the class you wrote to transform your input object.

```
#include <boost/iterator/transform_iterator.hpp>
struct getEmployeeName {
  std::string operator()(Employee& e) const {
    return e.name;
  }
};

struct EmployeeNames {
  typedef
     boost::transform_iterator<getEmployeeName,
     std::vector<Employee>::iterator> iterator;
  typedef iterator const_iterator;
  iterator begin() {
    return boost::make_transform_iterator(
       owner().employees.begin(),
       getEmployeeName());
  }
  iterator end() {
    return boost::make_transform_iterator(
       owner().employees.end(),
       getEmployeeName());
  }
  private:
    Staff& owner() {
        return *reinterpret_cast<Staff*>(
        reinterpret_cast<char*>(this)
        - offsetof(Staff, EmployeeNames)); }
} EmployeeNames;
```

Listing 5

Suggestions for further experimentation

It is easy to see that the previous two constructs can be combined: for example providing a way to iterate over only the names of the junior employees. For those trying to implement this, such an iterator declaration can quickly get unwieldy – it is advisable to use lots of `typedef`s for the iterator types to keep on top of things.

The `transform_iterator` can be used not only to query properties of objects. A useful 'transformer' class is the following:

```
template<typename TObj>
struct getObjectNr<boost::shared_ptr<TObj> > {
  typedef TObj* result_type;
  TObj* operator()(
     boost::shared_ptr<TObj> sh_ptr) const {
    return sh_ptr.get();
  }
};
```

This allows for making an iterator over a collection of `boost::shared_ptr`'s where the raw pointers themselves are exposed, rather than the shared pointers. This comes in handy in a codebase where smart pointers were added later but many parts of the code still only work with raw pointers.

We've only explicitly covered a part of the initial qualities that we sought in a solution to the original problem. Specifically, we haven't yet discussed using Boost.Iterator to make new iterators that work on non-stl types and we haven't done any input validation or value restrictions. Writing custom iterators is a topic of its own; a task that is simplified by the use of other parts of the Boost.Iterator libraries yet complex enough to warrant an eventual separate article.

Conclusion

Memberspaces are a powerful idiom to provide interfaces to collections. They make for expressive, easily readable code. With the help of the Boost.Iterator library, we can extend their functionality to more than readability: they can provide users of objects with an easy way to access that object's properties in a myriad of different ways – at little to no cost in terms of memory usage or speed. It provides class designers with ways to give users access to an object's properties while maintaining control over the way this is done. ■

Note

During review, a third technique to deal with this problem was mentioned that may be of interest. See [Bass05].

References

[Abrahams03] 'The Boost.Iterator Library', David Abrahams, Jeremy Siek, Thomas Witt, 2003: http://www.boost.org/doc/libs/1_36_0/libs/iterator/doc/index.html

[Bass05] Phil Bass, 'The Curate's Wobbly Desk', Overload 70 (December 2005)

[Higgins08] 'Custom Iterators in C++', Jez Higgins, *CVu* Volume 20 Issue 3 (June 2008), ACCU.

[LópezMuñoz02] 'An STL-like bidirectional map', Joaquín M López Muñoz, 2002: http://www.codeproject.com/KB/stl/bimap.aspx

Generics without Templates – Revisited

Can you use the STL if your compiler lacks templates? Robert Jones implements iterators and alogorithms.

In my last article on this topic [Jones08], I explored a method of creating an STL-like vector container, without making use of templates, for circumstances in which the template features C++ are not available.

In this article I shall extend the techniques explored previously, to include other containers and more importantly their ability to inter-operate, and finally examine a restricted, but still useful, implementation of some basic STL algorithms using these containers.

Review

By the conclusion of the first part of this article we had seen how to create an STL-like vector container, using a macro to generate a typed interface and then exploiting code-hoisting techniques to abstract a single, common implementation.

The resulting **`vector`** type could be used in this fashion, and offered the dynamic resizing and iteration capabilities of the STL vector (Listing 1).

Using similar techniques it is possible to create an analogue of the STL **`list`** container, and then things become rather more interesting.

Both the **`vector`** and **`list`** containers have methods such as

```
void insert( position, first, last );
```

in which the range **`first...last`** may be bounded by iterators of any type, including pointers. In the vector, iterators are indeed **`typedef`**'d as pointers (although there are exceptions to this, e.g. checked iterators), so the issue does not usually arise, but for other containers the semantics of their iterators are quite different. Iterators themselves must contain the semantics of how to move and compare them.

Towards a better Iterator

The behaviour required of iterators is essentially polymorphic – the concept of moving or comparing is common to all iterators, but the implementation of that concept for different iterators (aka different containers) varies.

The natural approach to implementing polymorphic behaviour would be to use inheritance – an abstract base iterator class, and various concrete derived classes to iterate over the various containers; however this approach is fatally flawed. Firstly, iterators are passed by value, and to do otherwise would be a sufficiently subtle departure from expected behaviour based on the STL to be folly, and consequently suffer from the slicing problem [Meyers]. Secondly, the ultimate intent of this work includes implementing some elementary STL-like algorithms, such as **`find()`**. **`find()`** returns an iterator of the same type as its range parameters. While, in the absence of templates, it is inevitable that **`find()`** must be overloaded for containers of many different types, it is undesirable to have to overload **`find()`** for many different containers of many different types, since this is an unbounded group in two orthogonal directions. Consequently, iterators over a collection of a single given type must be of a single given type.

However, inheritance still provides the mechanism for customization of behaviour, through a policy class wrapped by an iterator facade. The **`policy`** class expresses the essential operations necessary to implement a sufficiently general iterator (Listing 2).

A pointer to policy is then wrapped by a generic **`void *`** iterator core (Listing 3).

And finally a type-aware macro is wrapped around the core iterator to provide the necessary type information, which turns out to be just the size of the type for iterators. See Listing 4.

For the sake of brevity much has been omitted from these code snippets; both iterator core and wrapper must also exist in const versions, both need the full gamut of comparison operators, and a chain of methods must also be present to facilitate dereferencing, which must also be supported in the

Robert Jones Robert has been programming in C++ for many years, since the early days when C++ was only available as a cross compiler. His experience has been primarily in embedded environments, especially telecoms. Robert attended his first ACCU conference in 2007, and found the experience utterly engaging. He can be contacted at robertgbjones@gmail.com

Compilers

C++ compilers for embedded environments often lack support for some of the relatively recent additions to the C++ language, most notably templates and exceptions. There can be a number of reasons for this.

Some embedded environments are now very mature, and may now have only a small active development community. As a result it may not be commercially viable to update their compilers to support these features. Supporting templates and exceptions adds significantly to the complexity of the compiler, so the commercial case to make the investment may not be attractive.

However, probably the most common reason for not supporting these features is the obscurity they introduce to the runtime performance. Even with a basic C++ implementation, the apparently simple act of exiting a scope can cause a great deal of activity as a result of stack unwinding and all the destructor calls that may be made. By including exceptions and templates this unseen activity penalty is exacerbated. In the kind of environments where the system must respond within a handful of microseconds this kind of 'under-the-hood' activity can be problematic.

```
#include "vector.hpp"
typedef vector( unsigned ) IntVector;
IntVector v;
// Populate v;
for (
   IntVector::iterator i=v.begin();
   v!=v.end(); ++i )
{
  printf( "%u\n", *i );
}
```

Listing 1

```
struct AnyIteratorPolicy
{
  typedef int difference_type;
  virtual void const * client(
     void const * ) const = 0;
  virtual void increment(
     void const * & ) const = 0;
  // & decrement, advance etc.,
  // all const & non-const
  virtual bool equal(
     void const *, void const * ) const = 0;
  virtual difference_type distance(
     void const *, void const * ) const = 0;
  virtual ~AnyIteratorPolicy( ) { }
};
```

Listing 2

```
struct iterator_core
{
  iterator_core( AnyIteratorPolicy const *,
     void * );
  iterator_core & operator ++ ( );
  iterator_core & operator -- ( );
  // & +=, -=, +, -, equal etc.
  void * value( ) const;
private:
  AnyIteratorPolicy const * m_policy;
  void * m_value;
};
```

Listing 3

```
#define DEFINERANDOMITERATOR( const_iterator,  \
   iterator, type )                            \
struct iterator                                \
{                                              \
  typedef type value_type;                     \
  typedef iterator_core core_type;             \
  typedef core_type :: difference_type         \
     difference_type;                          \
  iterator(                                    \
     value_type * value,                       \
     AnyIteratorPolicy const * policy ) :      \
     m_core( policy, value )                   \
     { }                                       \
  iterator & operator ++ ( )                   \
     { ++ m_core; return * this; }             \
  /* also --, +=, -=, +, -, ->, deref etc. */  \
private:                                       \
  core_type m_core;                            \
}
```

Listing 4

policy since dereferencing a vector iterator may be very different to dereferencing a list iterator.

Naturally nothing comes for free, and there is a price to pay for this flexibility. Iterators are now at least twice the size of a pointer, and every dereference introduces at least an extra virtual function call.

All of this can now be packaged to make it more useable with a few macros (Listing 5).

Subsequently, when creating a new type over which it may be required to iterate, all that is required for a type **`A`** is that its header file **`#include`**s the iterator header, and appends a single line:

```
ITERATORS( A );
```

Inspection of the names chosen in the macros above suggests that in practice the definition of the full set of iterators is a little more complex, and this is true. However, the additional complexity is essentially rote-repetition of the approach described, and presents no new issues.

```
#define concat3( a, b, c ) a ## _ ## b ## _ ## c
#define i_category( category, type ) concat3(    \
   category, type, iterator )
#define const_i_category( category, type )       \
   concat3( category, type, const_iterator )
#define random_iterator( type ) i_category(      \
   random, type )
#define random_const_iterator( type )            \
   const_i_category( random, type )
#define ITERATORS( type )                        \
   DEFINERANDOMITERATOR(
   random_const_iterator( type ),
random_iterator( type ), type );                 \
   DEFINERANDOMCONSTITERATOR(                    \
   random_const_iterator( type ),
random_iterator( type ), type );
ITERATORS( bool );
ITERATORS( char );
ITERATORS( short );
ITERATORS( int );
ITERATORS( unsigned );
ITERATORS( long );
ITERATORS( float );
ITERATORS( double );
```

Listing 5

Iterator policy lifetime

The core iterator illustrated above contains a pointer to iterator policy, which naturally raises the issue of ownership and lifetime of the pointee. The policy may not, of course, be a contained member of the iterator, since the precise type of the policy varies for different containers, whereas the iterator must be a constant type for all containers of a given type. For iterators over containers the policy may be a member of the container, since the valid lifetime of the iterator can never exceed the lifetime of the container over which it iterates. Native pointers converted to iterators present a special case which will be examined separately.

Container changes

Changes to the container interfaces are now very minor, although reflecting more substantial changes to the implementation. Previously the vector container simply **`typedef`**'d pointers to create its nested iterator type (Listing 6). Now, using the macros above this becomes Listing 7.

With iterators as common types across all containers, and with each iterator containing the knowledge of how to iterate over its owning container, we can now use our containers in this fashion. (See Listing 8.)

```
#define vector ( TYPE )                         \
  class VectorOf##TYPE                          \
  {                                             \
  public:                                       \
    // ...                                      \
    typedef value_type * iterator;              \
    typedef value_type const * const_iterator;  \
    // ...                                      \
  }
```

Listing 6

```
#define vector ( TYPE )                          \
  class VectorOf##TYPE                           \
  {                                              \
   public:                                       \
     // ...                                      \
     typedef random_iterator( TYPE ) iterator;   \
     typedef random_const_iterator( TYPE )       \
     const_iterator;                             \
     // ...                                      \
   }
```

Listing 7

Algorithms

The containers and iterators of the STL would be of little use without the algorithms that operate on them. Simulating STL algorithms without using templates in general is beyond the scope of this article, and presents some difficult challenges. The root cause of difficulty is that algorithms are functions, in contrast to containers which are classes, and so benefit from implicit instantiation.

However by imposing some fairly draconian restrictions on the facility, and explicitly specifying types in circumstances where a templated facility could deduce them, some usefulness can be gained. Specifically, many useful algorithms, such as **find_if()**, accept unary predicates, i.e., functions taking one parameter and returning **bool**, and that is a sufficiently specific signature to make a macro generated base class acceptable. (Listing 9.)

This kind of approach can be extended arbitrarily, but quickly becomes unwieldy and difficult to maintain. However, defining sufficient unary predicates to implement some basic algorithms is manageable and useful.

With solid iterators and predicates written, algorithms follow a similar pattern, and are fairly straightforward. (Listing 10.)

These, as with functors, must be explicitly 'instantiated' before use. There are some subtleties here to do with constness of iterators, predicate arguments, and whether the algorithm is a mutating one, which makes a full working implementation of these techniques quite extensive. However, with care, and a few encapsulated const casts, STL-like behaviour can be substantially achieved.

Dealing with pointers

Pointers present some special challenges because the semantics of their operators cannot be changed. One possible approach would be to make iterators implicitly constructible from the corresponding pointer, but this would require the iterator to 'own' its own pointer policy, and so necessitate either copying the policy each time the iterator is copied, or holding it by counted pointer, which would involved a memory allocation for each iterator construction. Neither of these is considered an attractive option.

Instead, it is quite simple to overload all algorithms taking a range, and similarly all container methods taking ranges, with pointer versions. These versions are thin wrappers, but most importantly supply the iterator policy, and use the knowledge that the policy need only exist for the lifetime of the algorithm call.

Conclusion

With all of these facilities defined, some remarkably STL-like code can be compiled.

```
#include "vector.hpp"
int a[] = { 0, 1, 2, 3, 4 };
vector(int) v( a, a + sizeof(a)/sizeof(int) );
v.erase( remove( v.begin(), v.end(), 3 ),
   v.end() );
```

To this extent this work has met the original objectives of providing STL-like containers and the algorithms to use them in environments that do not support templates. The alternative techniques employed, those of nested macros and virtual functions, have their own costs and drawbacks, and in the environments where templates are unavailable may be equally unacceptable. The resulting code is brittle, and the macro generation techniques that make this even moderately amenable to comprehension produces a lot of redundant code artefacts.

However it is achieved, the flexibility and interoperability of iterators comes at a cost. ■

Author's note

Since producing the code described in this article I have moved on to other work, and in particular have become very much more familiar with the state of the art of template programming, and in particular the amazing work that has been done in the Boost libraries. Having had the opportunity to understand the full capabilities of templates, the work described here seems a very poor imitation of compiler templates.

If I were in this position again, I would push much harder for adoption of templates, with far greater understanding of the productivity benefits they can bring.

References

[Jones08] 'Generics Without Templates', *Overload 83*
[Meyers] *Effective STL*, item 38

```
#include "vector.hpp"
#include "list.hpp"

list( unsigned ) l;
vector( unsigned ) v;
//...populate l...
v.assign( l.begin( ), l.end( ) );
```

Listing 8

```
#define unary_function( arg, ret ) concat4(     \
   unary, ret, fn, arg )
#define unary_functor( arg, ret )  concat4(   \
   unary, ret, ftor, arg )
#define DEFINE_FUNCTION( fn, arg, ret )       \
    typedef ret ( * fn )( arg & );            \
#define DEFINE_FUNCTOR( STRUCT, arg, ret )    \
  struct STRUCT                               \
  {                                           \
    typedef arg argument_type;                \
    typedef ret return_type;                  \
    virtual return_type operator( )(          \
    argument_type & ) = 0;                    \
    virtual ~STRUCT( ) { }                    \
  }
#define DEFINE_FUNCTORS( arg_type, ret_type ) \
DEFINE_FUNCTION(                              \
   unary_function( arg_type, ret_type ),      \
   arg_type, ret_type );                      \
DEFINE_FUNCTOR(                               \
   unary_functor( arg_type, ret_type ),       \
   arg_type, ret_type );
DEFINE_FUNCTORS( bool, bool );
DEFINE_FUNCTORS( char, bool );
DEFINE_FUNCTORS( short, bool );
DEFINE_FUNCTORS( int, bool );
DEFINE_FUNCTORS( unsigned, bool );
DEFINE_FUNCTORS( long, bool );
DEFINE_FUNCTORS( float, bool );
DEFINE_FUNCTORS( double, bool );
```

Listing 9

```
#define DEFINE_FIND_IF_FTOR( Iterator,          \
  Predicate )                                   \
  Iterator find_if(                             \
  Iterator first,                               \
  Iterator last,                                \
  Predicate & p                                 \
  )                                             \
  {                                             \
    for ( ; first != last; ++ first )           \
      if ( ( p( * first ) )                     \
        break;                                  \
      return first;                             \
  }
```

Listing 10