overload 90

APRIL 2009 £3

A Rube-ish Square

Further investigation of the rube-ish square, with some simple group theory

If You Can't See Me, I Can't See You

Knowing where the doors are is only the start. What can you see through them?

On Management: Product Managers

We shed some light on the poorly understood mysteries of Product Management

Testing State Machines

State machines are a common design pattern. We see how to separate their concerns to make testing easier.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 90

April 2009

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 91 should be submitted by 1st May 2009 and for Overload 92 by 1st July 2009.

ACCU

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

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

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

Copyrights and Trade Marks

Back to School

Much is made of the Knowledge Economy. But just how can you keep up to date?

What do you love most about being a programmer? For me it's knowing that everyday there's going to be something different to do and find out – a new bug to track down and fix; some design work that needs thinking about; a meeting to discuss what the product requirements are; investigating the latest and greatest technology; reading books and newsgroups for new ideas; and sitting down and writing code to solve these problems.

What I dislike is when things get samey and repetitive – especially when the root cause of this repetition is known about but somehow never gets improved.

This struck me recently as I was reading Allan Kelly's book *Changing Software Development: Learning to become Agile* [Kelly], and the first few chapters are all about learning and building up knowledge. Not just how people learn, but also how the organisations we are in learn, and how the software we produce reflects this.

Indeed, a program can be considered as encoded knowledge. The understanding of the requirements by people such as the customer, product manager, and marketing, are used by developers to come up with a solution that addresses those requirements and teaches the computer makes life better for the final end user. This information is channeled through processes and eventually produce a product. That product reflects not only the information about the requirements, but also the processes used to create the final code.

This applies very much over time, which can result in what I only half-jokingly call 'Software Archeology' – when you try to understand some odd piece of code you can gain useful insights if you think of it as accumulated layers of historical code changes made by past actors each with their own motives. Sometimes an appreciation of why something was done in an initially unexpected way becomes much clearer if you can work out what problem they were trying to solve – perhaps what you are seeing is a perfectly valid work round for an awkward problem in a tight deadline situation, both of which are long gone. A classic example would be a kludgy expression to avoid a compiler bug in a long-gone version. So instead of seeing some odd looking code and thinking 'Which idiot wrote this?' it's better to think 'What odd constraints forced them to do it like that?', which is not only a more realistic way of looking at code, but also avoids a blame culture.

But the knowledge is not just in the code – it's also written in the obvious documents that are generated – this knowledge is explicit – but less well appreciated is in the minds of people and the processes that have grown around such knowledge. This knowledge is known as tacit – implied and not consciously thought about (or if you will excuse the Rumsfeldian term, it's an 'Unknown Known'). This is one reason that Documentation and Code Handovers work so badly, whether it's for something that has been outsourced, handing over from one team to another, or during the final few weeks before leaving a job. Writing documents just captures the Known Knowns, or at least the ones you remember to write down (and it might not even be read), and a huge amount of vital knowledge can be lost.

Tacit knowledge is a remarkably useful resource. For example some programmers will get to know an area of code in great detail – not so much the technical aspects, but at a deep 'gut feel' level. It might show itself in the form of a rapid evaluation of a bug symptom into a likely area of code to investigate ('Oh, the Foobaz handling has always been a bit fragile – I'd start looking there'), or a deep understanding of why some code is tricky ('Even though this area is complex, it's because it's having to deal with complex and contradictory requirements – a big rewrite won't improve things much as it will still have to reflect that complexity'). Sadly such tacit knowledge can be easily overlooked, partly because people don't realise it's there, partly because it's harder to justify hard decisions based on 'gut instincts', and partly because the people who have most first-hand experience of a system (and thus have most tacit knowledge of it) are often the people least likely to be making major decisions about it's future. (The pattern Architect Always Codes is a way of avoiding this – keeping them in touch with their own designs helps them keep their knowledge of how it really works up to date.)

> *It is not hard to learn more. What is hard is to unlearn when you discover yourself wrong.*
>
> *~Martin H. Fischer*

If we only learnt from our immediate work environment then we'd be in danger of staying in a rut, falling into groupthink problems ('We do it this way because we've always done it this way'), or just failing to take on board new ideas and new observations. One way is to keep in touch with people in similar situations (and a few wildly different ones), and exchange anecdotes, experiences, and techniques. For example, I read quite a few software and management books, programming newsgroups and mailing lists, a few blogs, and of course Overload and CVu.

Plus of course there are conferences and smaller get-togethers, where on top of the obvious value of the presentations and sessions, the informal chats in the bar can be amazing learning experiences and huge eye openers – they might not tell you a solution, but it might just make you think and realise you have a problem in the first place.

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

And learning opportunities don't have to be from software specific sources. There was an interesting thread on accu-general recently asking what non-fiction books would you recommend, things that were 'Influential classics, non-fiction, books about people's original ideas. Ideas that changed the world. Great thinkers, who influenced the way we think, with what they wrote. Politics, philosophy, science, and to a lesser extent, economics, history, psychology.' (Thanks to Thaddaeus Frogley for starting the thread)

Here I reproduce some of the ones I thought were of most interest (apologies to those I left out):

- *How to Read a Book*, by Doren and Adler (a cheat as it has a reading list itself)
- *The Republic*, by Plato
- *On the Origin of Species*, by Darwin
- *Lateral Thinking* by Edward de Bono
- *The Prince*, by Machiavelli
- *Predictably Irrational*, by Dan Ariely
- *Godel, Escher, Bach: An Eternal Golden Braid*, by Douglas Hofstadter
- *Relativity: The Special and General Theory*, by Einstein
- *The Design of Everyday Things*, by Donald Norman
- *The Timeless Way of Building*, by Christopher Alexander
- *How Buildings Learn*, by Stewart Brand
- *The Earth: an intimate history*, by Richard Fortey
- *The Great Crash, 1929*, by Galbraith
- *The Making of the Atomic Bomb*, by Richard Rhodes
- *The Soul of a New Machine*, Tracy Kidder
- *Seven Habits of Highly Effective People*, by Stephen Covey
- *The Science of Cooking*, by Peter Barham

and, intriguingly:

- *Principles of Helicopter Flight*, by Jean-Pierre Harrison

Interestingly enough, while none of the above are directly about software, several have strongly influenced relevant ideas, in particular Alexander's works led to the patterns movement (and in turn their need to collaborate on pattern writing led to the invention of the Wiki); Brand's ideas are about system design and ongoing maintenance; and Norman's ideas apply to UI design. We can learn from many sources.

Some people will never learn anything, for this reason,
because they understand everything too soon.
~Alexander Pope

Sometimes though, people just doesn't want to learn. A depressing story was related to my by a friend who I've been trying to persuade to come to the ACCU Conference. He was very keen, but apparently his management turned him down with the excuse that he was 'senior enough to just need books', which I thought missed the point of conferences rather spectacularly. Plus, if there's one thing I've learnt over the years, is that the 'experts' who write the books are the first ones to admit the gaps in their knowledge, but want to find out – and will write a book about it.

I wonder how many other organisations and people have such an attitude to learning? Many of the developers and companies I know are self-selecting in that regard – I know them via communities of people who do want to learn, so all I see are the 'good' ones. But I don't know of all the other communities, and ultimately if people don't want to learn and join such groups then I won't hear about them at all.

This is why there's a common interview question along the lines of 'what was the last technical book you read?', or 'how do you keep up to date with new innovations?' The answers can give some insight into how much that person approaches their work (caveat: as with all such open interview questions there are no right or wrong answers, just extra information that can be put together to form a fuller picture of that candidate, so an informed judgement of their suitability for the role can be made.)

And the questions work the other way too – if interviewing how would you answer 'How do you help developers improve?'

References

[Kelly] http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047051504X.html

Software Development in the 21st Century

What's the future of software development? Alan Griffiths and Marc Allan have a timely look at recent experiences.

Software development is an observational science rather than an experimental one: that is, it is difficult if not impossible to try varying the conditions that affect the outcome one at a time to disentangle their effects. Observing projects can be difficult though: some organisations are hostile to their intimate practices being exposed and observers filter their observations through their individual perceptions of what is important.

The current authors are as prone as others to subjective reporting, but hope that the following experience report describing a project lifting itself from primordial chaos to 21st century Agility will provide useful examples to others working in this field.

In the beginning

Despite all the progress made in the latter half of the 20th century a lot of software development is still a disorganised mess which delivers results only through heroic developers sacrificing their lives (or at least their home lives) for the good of the project. Such are the beginnings of our tale.

When one of the authors was first asked to consider working on the project it was known by management to be in difficulties – the project manager (who was based on the far side of the Atlantic to the rest of the team) had, despite frequent visits and long phone calls, difficulty understanding what work was being undertaken and why. As is often the case, the lead developer came under criticism for not communicating effectively, the situation worsened when frustration forced a senior developer on the team to tender his resignation. The brief was simple: get things back on track!

The software involved was a server-side system supporting numerous end-user applications around the globe all of whom were competing to get their favoured features implemented. Luckily most of the developers of these applications were co-located in London with the developers on the project. This meant that user requirements capture and eliciting feedback were never problematic during our time with the project.

We join the team

Timing was good – one of us had a week's handover from the departing developer which afforded a sure footing and a rapid start. The reality on the ground bore some resemblance the situation described at interview – although it has to be said that the lead developer was willing to explain what he was doing and why, the difficulties arose as he was interacting directly with the clients in the same office and, understandably, failing to report every conversation about issues, features and changing priorities across the Atlantic.

The first step was to consolidate the current issues, feature requests and state of work in a form that was accessible to everyone involved. As there was an existing project Wiki this was used for the purpose. This got a lot of positive feedback from both the project manager and from the client teams all of whom could see what was going on for the first time. It was during this exercise that the second author joined the team.

However, despite the initial enthusiasm for the Wiki based task display it slowly became obvious that these web pages only make progress visible if people visit those pages. And so, as the novelty wore off these slowly fell into neglect. Even regular planning 'meetings' (conference phone calls to review the Wiki) failed to keep the information flowing. This became dramatically apparent when emails began to circulate with the project manager demanding to know why a feature wasn't being worked on only days after he'd ran a conference call with the users that had agreed on the work for the iteration.

Organisational changes happen

While we were pondering how to address this problem some changes happened that presented an opportunity. In this case there were some changes at board level which resulted in the appointment of a new project manager located in the same office as the project team!

This made a major difference, as he could interact directly with user groups and quickly discovered that the work being done was not giving the best return. In particular, the team was working on a fourth generation implementation of the system whilst the production environment was a patchwork of the second and third generations. He decided to stop work on the fourth generation and focus effort on completing the roll-out of the third generation. The lead developer quit in disgust.

Shortly after this we moved offices and the opportunity arose to acquire a whiteboard: there were several allocated to the office, but as they had not been assembled we found them languishing in the corner of a side office. By assembling one behind our desks we established possession and moved our planning onto this display.

We divided the whiteboard horizontally into three columns – for the release being rolled out, for the release being developed and for future activities. A post-it was used to denote each task and moved from the bottom (not started) to the middle (started) to the top (completed). Planning meetings became brief and increasingly informal and centred around the whiteboard. (Figure 1)

While this was happening we also managed to get continuous integration running. Part of the legacy of the first developer who quit was a set of functional tests – after a bit of work on these they turned into a test suite that gave us (and the other users of the UAT environment) the ability to automate the promotion of each 'successful' build into the UAT environment. This got us early feedback from other teams about new

Alan Griffiths is an independent software development practitioner who fixes development processes as well as code (in C++, Java, C#, Python and JavaScript). He is a long-standing contributor to the ACCU journals, mailing lists and conferences. For more, see www.octopull.co.uk

Marc Allen has been a professional programmer since this morning and is hoping his new-found career lasts until lunch. He helps organise the Loebner/Turning Prize whenever it visits the UK and currently writes software for a former investment bank. Marc can be contacted at marc.j.allen@gmail.com

Unfortunately, as all too often is the case, the new approach fell foul of its own success

Figure 1

features – one of whom had an independent 'acceptance test suite' that they ran regularly (and that caught our errors a couple of times before we got our own tests right – our system had features we didn't know about).

Time passed, the team grew and we successfully introduced a 'test first' culture. The legacy 'second generation' system was withdrawn from service and a raft of important new features made it into production. We had a steady 'heartbeat' of releases improving the system and good feedback from users.

Outsourcing

The next hurdle came when the manager decided that there were problems with the whiteboard planning system. These were varied: the unavailability of the whiteboard for other teams to use (particularly other ones he managed); the 'danger' of losing information if post-its fell off or were moved accidentally; and, the lack of visibly outside the office. It was probably the latter that was decisive (as his desk in a side office didn't have line of sight, there were client teams in another building, and he was considering outsourcing some of the development work to Yorkshire).

In any event, the corporate standard for project planning was a web based system: Jira. This avoided the principle problem encountered with the earlier Wiki based system (that people didn't see progress if they didn't visit the Wiki). With Jira stakeholders could subscribe to tasks that interest them and receive email notifications when they were updated.

With the arrival of the inevitable mandate to migrate the project planning system to Jira, the development team downed IDEs for web browsers for an afternoon. Converting each and every post-it to a Jira ticket was soon accomplished and the (now blank) whiteboard became (for many weeks) a monument to the past. Developers from the outsourcing company were each brought in for an introductory period to learn the system and meet the rest of the team. It seemed like no time before the goal of a geographically diverse team was accomplished.

The new approach propelled a new wave of productivity as new functionality passed through UAT into production, Jira streamlined the development team's efforts and, thanks to integrated report generation, promoted a higher degree of transparency with management and end users.

Unfortunately, as all too often is the case, the new approach fell foul of its own success. With increased productivity, release confidence and management support all on the increase – the publicity not only spurred on more interest in the project (affording a steady stream of functionality requests) but also promoted the project as a campaign poster child for Jira. (And outsourcing to Yorkshire!)

Quite frankly the support teams had understandably not requisitioned sufficient hardware resources to sustain the mushroom cloud that was to follow the eruption of Jira adopters. The knock on effect of this was that the beloved tool which had served the team so well now became a bottleneck. But success breeds success and the project budgets would now support a few upgrades – not only to servers but to desktops.

While multiple 24-inch monitors were a delight to the developers the project manager was especially pleased with getting a massive screen installed in a nearby conference room. (We didn't pay for this directly – but as a star outsourcing department we needed to demonstrate meetings with our offsite team members – the group 'outsourcing project' funded this on the basis that we were willing to share.)

April showers

Although our project continued it's meteoric progress during the recent recession this was not true of other parts of the organisation and it soon became apparent that our private conference room was going to have squatters (moved from another building that was no longer needed). So, one quiet evening we swapped the still blank whiteboard for the sexy display screen from the conference room.

We even 'found' (ok, one of us was sad enough to write it) a new screensaver for it that would show the project's Jira homepage – regaining the visibility of project progress that we'd missed since the whiteboard went blank. We could still conference with our rural colleagues – although, as a result of 'downsizing', there did seem to be fewer and fewer of them as time progressed until there were finally none left.

In addition to 'our' conference room these organisational changes also lost us the office formerly occupied by the manager. He was given, and took, the 'opportunity' to improve the in-team communications by joining the rest of us in the main office.

Jira gets an upgrade!

We were still rolling out enhancements at a steady rate when the project manager pointed out a cool new plugin for Jira: GreenHopper. 'What's GreenHopper?' we asked. Anyway, it turned out that this new plugin, amongst other things, allowed Jira to displayed tasks as 'post-its' and one could move those by 'click and drag'. Before very long our status screen

We had avoided a lot of unnecessary keystrokes and reached a pinnacle of automation

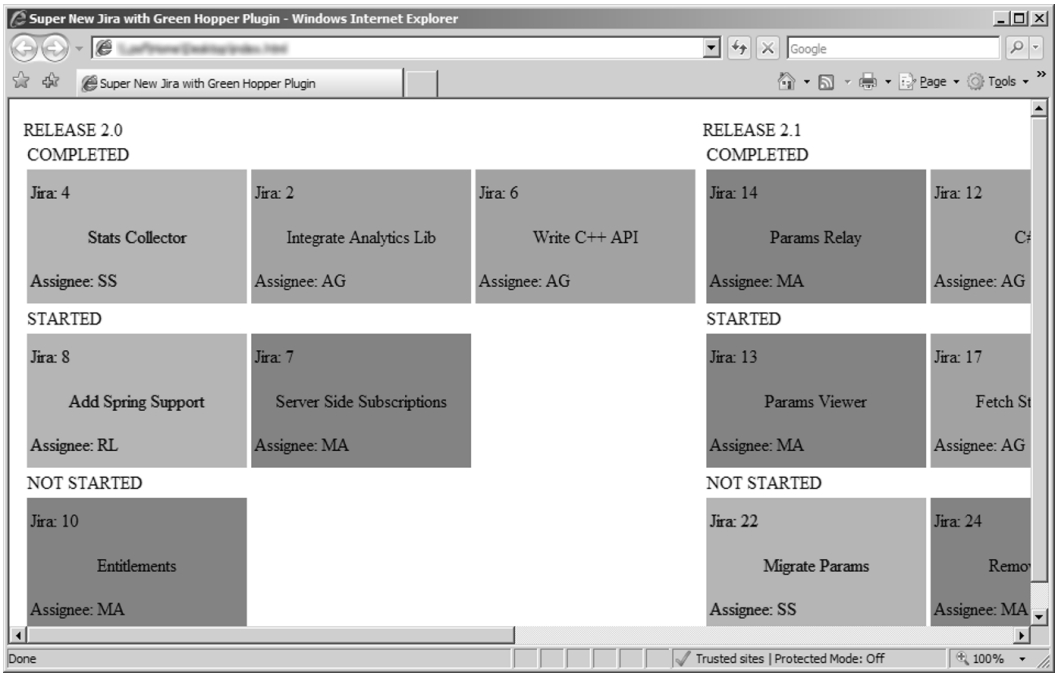

Figure 2

was demonstrating the advantages of a proper use of technology over the whiteboard based 'prototype' we'd used earlier in the project. Not only did we have a high-visibility display of the state of the project, it could be viewed over the internet and could generate the all-important status emails automatically. (Figure 2)

The one thing it didn't do 'out of the box' was to allow us to walk up to the screen and move the post-its by hand. This, naturally, was too much of a temptation – very soon some 'gesture sensitivity' was added to the screensaver. It is so cool waving a task note from 'in progress' to 'completed' – Tom Cruise eat your heart out!

Naturally this was so good that we had to make it better – we only needed to get the integration between Subversions and Jira working properly and attach webcams to all the workstations and we could automate closing tickets by smiling as we checked in the code. By pulling up the Jira page and glancing in the right direction we could allocate the next piece of work. We had avoided a lot of unnecessary keystrokes and reached a pinnacle of automation.

Not long before the end

Unfortunately and perhaps quite predictably, the state of software development nirvana was about to undergo some radical change. Whilst the project's deliveries continued to roll out to production at a steady, maintainable pace; the powers that be thought they knew better. It turns out, quite ironically, Jira was making us look bad. The magic that went on behind the scenes to ensure the team's efforts were focused on functional requirements had become an intricate part of the day's activities. The resulting stream of Jira update e-mails soon overwhelmed our stakeholders' inboxes. Very soon rules were set up to 'mark as read' and 'delete' the daily 'spam' generated by the team.

Management finally realised that something special was happening – not only were features being delivered on (or before) schedule, but despite the loss of our offsite developers the pace of deliveries was accelerating. Clearly, they thought, the only thing holding progress back was the developers. No sooner was this thought formulated than it led to action – the authors wished the project success, invited the manager to a leaving do and found alternative, less modern, employment (where their talents were still needed). We trust that the management insight was valid and the project continues to progress with even greater efficiency – we will report further if we hear details. ■

Testing State Machines

State machines are a common design pattern. Matthew Jones seperates their concerns to make testing easier.

Anyone can code up a state machine, but can you make such a machine fully testable? Can you prove that it is? Can you do this repeatably? In other words, can you present an inquisitor with a test suite that proves your state machine fully implements a given state transition diagram?

Depending on your background, you might choose to write your state machine using the C `switch()` idiom, with a `#define` per state, or better still, an `enum`. You might go so far as to have a function per state, with another `switch()` for each input event. It meets the requirements and it will probably work. But you know you can do better than that.

You might decide to go for a fully fledged GoF [Gamma] pattern-based design with a class per state and maybe a state factory. Now you are in the familiar territory of 'proper' OO and patterns. If you take this approach, surely the result will be perfect.

The trouble with testing

Whichever approach is taken, most people will naturally include code, in the state machine, that is concerned with implementing the outputs of the states. It makes a lot of sense, and it is the path of least resistance. The effort has already been put into decoding the state and handling the new input event. Having got that far it is very easy to simply add a line or two of code to finish the job, i.e. implement the 'action' part of the transition. While this code might be trivial, its impact is often not so. In the worst case (e.g. an embedded system), it might involve writing to hardware, turning on motors, lighting lights etc. Whatever the context, the state machine will be exposed to the application, and will therefore have dependencies on that application code. And this in turn makes the testing complicated.

To test a 'traditional' state machine, i.e. one where the output code is mixed with the state transition code, typically you would have to run the application, stimulate it somehow, and look for secondary evidence that the state machine is working. You might even have to resort to primary evidence: good old `printf()`. It is even worse in the embedded scenario: you could be forced to have real equipment, or an adequate simulation, just to run the code. This is obviously not good. You will have do an awful lot of work to produce any form of automated test. TDD will be hard because of the intrusion of the application. In the worst cases, where 'application' involves hardware, repeatable testing could even be impossible. At this point we would naturally give up on the goal of automated testing and resort to the bad old ways of testing the code once, manually, declaring it fit, and never going back. And with this approach comes the inevitable fear of later changes to that area since it is rightly considered fragile.

There are any number of tricks to get round this problem, but they will all emit 'bad smells' [Fowler]. You might stub the application by substituting a test version of `application.cpp`. You might add test instrumentation to the application. It might even be conditionally compiled so you can switch it off in the 'real' system. These are all poor solutions and will have you be tying yourself in knots of test-only code which will pollute the deliverable code and make it hard to read, understand, and maintain.

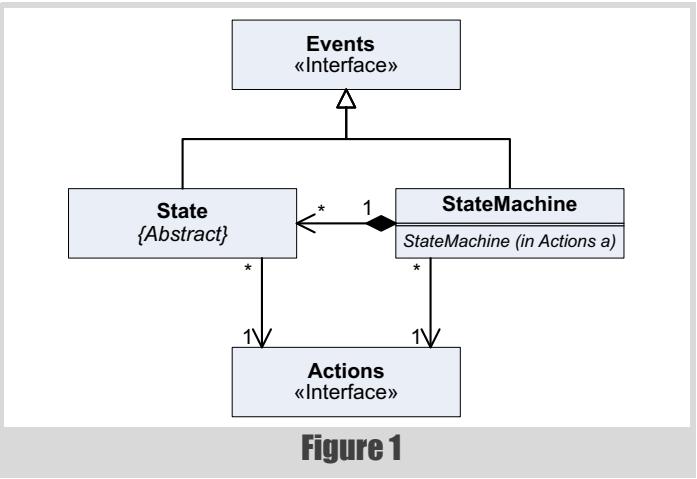

Figure 1

So by starting out innocently enough and harmlessly mixing state machine logic with the application, you can easily end up seriously compromising your development. But there is a better way. Of course there is: there are probably many, and if you are already a master or mistress of testable state machines, congratulations: stop reading now. For everyone else, the rest of this article describes an approach I developed recently. The background to this work is embedded software, and so the problem of testing a state machine is far more apparent than where hardware is not involved.

What is a state machine?

If we ask ourselves 'what is a state machine?', the answer (in a software context) should be something like 'code that manages the state of something, responding to external events, and translating them into actions to be implemented by the system'. State transitions will result in output actions that are communicated to the system, but only in an abstract, or event-like way. The detail of carrying out these actions is not part of the state machine, because 'detail' implies exposure of the state machine to knowledge of the application. It is this last point that is usually overlooked, leading to the blurring of the state machine and the application. It might appear to be somewhat picky, but if we allow the state machine to do two jobs (state management and controlling the application), we lose separation of concerns [Wikipedia1] and reduce cohesion [Wikipedia2].

This stripped down definition translates perfectly to an object oriented approach: we have an interface describing the input events, and an interface describing the output actions. The state machine implements the events interface, and the application implements the actions interface. It really can be as simple as that. See Figure 1.

Matthew Jones started programming with BBC Basic, and then learned C on a summer job between school and VI form. He has been in programming professionally for over 15 years, having moved on to C++, and is happiest working on large embedded systems. He can be contacted at m@badcrumble.net

We have isolated the state machine from the application with two interfaces: Events and Actions. This is one of the fundamental principles of good design: partitioning [Griffiths]. We reduce the coupling between the state machine and the application to two simple interface classes. This allows us to test the state machine with mock, or test, objects [Mackinnon]. Later, we can implement the 'proper' version of the interface in all its application-ridden glory, safe in the knowledge that the state transition logic is perfect. It also allows the application to be tested with a mock state machine, should we wish to, by substituting the implementation of the Events interface.

An example

At this point we need to introduce an example and start talking in more direct terms. Figure 2 shows a state transition diagram for an external security light. The example is obviously trivial but I tried working through a few larger ones (e.g. 10 states) and it quickly turns from a useful example to 500+ lines of code showing most of a real system. Crucially, this example also includes interaction with hardware, so that a traditional implementation would require manual testing.

The security light moves between two high level states: day, when the lamp is off; and night, when the lamp is controlled by a movement sensor. Transition between these states is controlled by an ambient light level sensor. In the night state, when movement is sensed, the lamp is turned on and a timer is started. When movement ceases, the lamp is turned off by the timer. Note that although the sensors and timers might have thresholds, or return variable readings, in the realm of this state machine they are reduced to valueless events.

To turn this into code, we need four main classes: the **`Events`** interface, the **`Actions`** interface, the **`StateMachine`** and a **`State`** base class.

The **`Events`** interface declares the events that stimulate the state machine. These are the state machine inputs.

The **`Actions`** interface declares the actions that the state machine may cause. These are the state machine outputs.

The **`State`** class is the base class from which all states are derived. It inherits the **`Events`** interface because every state must be able to react to every event. There are a lot of details missing here. (If you are really interested, the fully worked example is available here: http://accu.org/content/journals/ol90/TestableStateMachines.zip). For instance the state must have some way to change to a new state. In practice each state should be constructed with a **`StateContext`**, which includes a **`StateFactory`** for creating new states; a **`StateChanger`**, to allow the new state to be passed to the state machine; and an **`Actions`** instance. There is one important detail, though, and that is that all **`State`** classes are themselves state-less. They are constructed with sufficient context to function, but no more. It might be that in a more complex system this would not be practical,

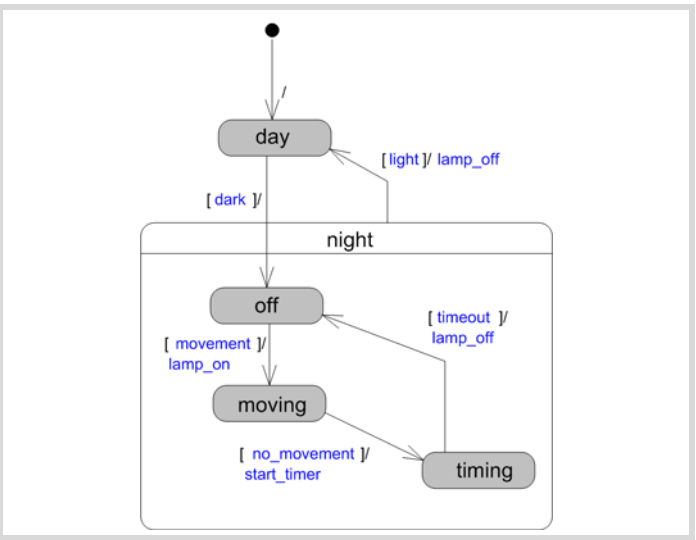

Figure 2

```
class Events
{
public:
    virtual ~Events() {};
    virtual void Dark () = 0;
    virtual void Light () = 0;
    virtual void Movement () = 0;
    virtual void NoMovement () = 0;
    virtual void Timeout () = 0;
};
class Actions
{
public:
    virtual ~Actions() {};
    virtual void LampOn () = 0;
    virtual void LampOff () = 0;
    virtual void StartTimer () = 0;
};
class State : public Events
{
private:
    Actions &actions;
public:
    State (Actions &a) : actions(a) {}
};
class StateMachine : public Events
{
private:
    Actions &actions;
    State *currentState;
public:
    StateMachine (Actions &a)
    :
    actions(a),
    currentState (day)
    {}
    // Implement Events interface
    void Dark ()        {currentState->Dark();}
    void Light ()       {currentState->Light();}
    void Movement ()    {
       currentState->Movement();}
    void NoMovement () {
       currentState->NoMovement();}
    void Timeout ()     {currentState->Timeout();}
};
```

Listing 1

but in this example, and all my real world implementations so far, it has held true. Incidentally, having stateless **`State`** classes also simplifies the problem of creating and changing state: one permanent instance of each State can be created by the StateFactory, and repeatedly handed out when required. There is no need to create new objects dynamically.

The **`StateMachine`** class brings everything together. It inherits the **`Events`** interface so that the application can signal events to it. Every event is delegated to the current **`State`**. This is the classic **`State`** pattern [Gamma]. The **`StateMachine`** must be constructed with an **`Actions`** instance. The **`Actions`** instance is added to the **`StateContext`** (not shown) which is passed to every **`State`** on construction.

The key classes are summarised in Listing 1.

Given this framework, and a number of helper classes already alluded to, we can concentrate on implementing the state transition diagram correctly. The realisation of the state transition diagram is the implementation of the **`Event`** interface, in each of the **`State`** class. Given the **`State`** class hierarchy in Listing 2, the translation of Figure 2 into code is completed in Listing 3.

The simple example has turned into two interfaces and eight classes. It should already be obvious that there is one thing missing: the application,

```
// Basic implementation of State interface giving
// default behaviour.
class StateImpl : public State
{
    // ...
};

class Day : public StateImpl
{
public:
    void Dark ();
};

class Night : public StateImpl
{
public:
    void Light ();
};
class Off : public Night
{
public:
    void Movement ();
};

class Moving : public Night
{
public:
    void NoMovement ();
};

class Timing : public Night
{
public:
    void Timeout ();
    void Movement ();
};
```

Listing 2

```
void Day::Dark ()
{
    context.changer.ChangeState(
       context.factory.OffState());
}
void Night::Light ()
{
    context.actions.LampOff();
    context.changer.ChangeState(
       context.factory.DayState());
}
void Off::Movement ()
{
    context.actions.LampOn();
    context.changer.ChangeState(
       context.factory.MovingState());
}
void Moving::NoMovement ()
{
    context.actions.StartTimer();
    context.changer.ChangeState(
       context.factory.TimingState());
}
void Timing::Timeout ()
{
    context.actions.LampOff();
    context.changer.ChangeState(
       context.factory.OffState());
}
```

Listing 3

```
class TestActions : public Actions
{
public:
    enum ActionType
    {
        LAMP_ON,
        LAMP_OFF,
        START_TIMER
    };
    std::vector <ActionType> v;
    void LampOn () { v.push_back(LAMP_ON); }
    void LampOff () { v.push_back(LAMP_OFF); }
    void StartTimer () {
       v.push_back(START_TIMER); }
};
```

Listing 4

and this is precisely the point of this whole approach. Describe and write the state machine in terms of state transition logic and nothing more.

Testing the example

All we need to do now is write a test implementation of the **`Actions`** interface, and then we can start some serious testing. Listing 4 shows one way to do this.

In our test harness, we are now able to construct a **`StateMachine`** and pass in a **`TestActions`** object. We can then devise a set of state transition tests, run them, and inspect the contents of **`TestActions::v`**. The term 'devise' it rather strong, in fact, since we should be pedantic and test all the inputs to all the states, there isn't much to tax the imagination. In other words we should extract a complete state transition table from the code and compare this to what is expected. Listing 5 shows an abbreviated version of such a harness. It makes assumptions about a number of features to aid testability, such as **`StateMachine::ChangeState()`** and **`StateMachine::ReportState()`**. Although it is clearly excessive to test such a simple example, it scales very well to realistic levels of complexity. The important point to note is that given the **`enum`**, **`struct`**, and helper functions, **`main()`** is straightforward, clear, and self-documenting. With a bit more effort, the helper functions can also output helpful information when tests fail, helping debugging.

At this point we have exhaustively tested our state machine, which is designed to control real hardware, in a software-only test harness. We can prove that it is a faithful implementation of the original design. Armed with this powerful approach to testing, we can start to write state machines with a new level of confidence.

When I was working all this out for the first time, I stopped at this point and offered up my 'perfect' new module of code for system testing. It worked, of course, but system testing revealed a number of subtle defects in the design of the state machine itself. I had perfectly implemented a flawed design, and I could prove it. Fortunately, the solution was close at hand.

Testing transition sequences

The test mechanism can be very easily extended to provide a second extremely useful facility. If more than one input event is allowed in a test vector, and it tests more than one expected output action, we can test sequences of transitions. This means we can test what amounts to use cases for the state machine. For example a single sequence test might be:

day --> off(night) --> moving --> timing --> off --> day

For realistic levels of complexity this testing offers more value than simply proving correct transition logic. Of course we would still retain the simple transition tests. This is what I did for my development system: I worked out the normal, and abnormal, routes round the state transition diagram, and added them to the test harness. And the problems jumped out immediately in the form of unexpected actions. Although each transition on the diagram seemed right, I had not worked through real examples, and the results of combinations of transitions. The original design allowed the

```
enum StartingState { START_DAY, START_OFF, START_MOVING, START_TIMING };
struct TestVector
{
    // Type for a pointer to void (void) member function of StateMachine.
    typedef void (StateMachine::* SMFunctionPointer) (void);
    const char *        testTitle;
    StartingState       startingState;
    SMFunctionPointer   eventFunctionToApply;
    const char *        expectedState;
    const TestActions::ActionType *firstAction;
    // etc. for other actions if required...
};
void TestOneAction (TestActions &actions, const TestActions::ActionType &expected)
{
    if (!actions.v.empty())
    {
        TestActions::ActionType &action = *actions.v.begin();
        assert (action == expected);
        actions.v.erase (actions.v.begin());
    }
    else
        assert ("action list unexpectedly empty");
}
void TestOneTransition (const TestVector &v)
{
    TestActions resultingActions;
    StateMachine uut (resultingActions);
    switch (v.startingState)
    {
    case START_DAY:    uut.ChangeState (uut.DayState());    break;
    case START_OFF:    uut.ChangeState (uut.OffState());    break;
    case START_MOVING: uut.ChangeState (uut.MovingState()); break;
    case START_TIMING: uut.ChangeState (uut.TimingState()); break;
    default:
            assert ("Unknown starting state");
            return;
    }
    // clear the results of changing to the starting state.
    resultingActions.v.clear();
    cout << "Starting state is: " << uut.ReportState() << endl;
    (uut.*(v.eventFunctionToApply))();
    if (v.firstAction)
        TestOneAction (resultingActions, *v.firstAction);
    while (!resultingActions.v.empty())
    {
        assert ("Found unexpected action");
        resultingActions.v.erase (resultingActions.v.begin());
    }
    cout << "Resulting state is " << v.expectedState << endl;
    assert (strcmp (uut.ReportState(), v.expectedState) == 0);
}
static const TestActions::ActionType L_On (TestActions::LAMP_ON);
static const TestActions::ActionType L_Off (TestActions::LAMP_OFF);
static const TestActions::ActionType Start_T (TestActions::START_TIMER);
int main (void)
{
  TestActions ta;
  StateMachine sm(ta);
  const TestVector v[] = {
  { "Day + dark -> off",                START_DAY, StateMachine::Dark,       "Off", 0 },
  { "Day + light -> no change",         START_DAY, StateMachine::Light,      "Day", 0 },
  { "Day + movement -> no change",      START_DAY, StateMachine::Movement,   "Day", 0 },
  { "Day + no_movement -> no change",   START_DAY, StateMachine::NoMovement, "Day", 0 },
  { "Day + timeout -> no change",       START_DAY, StateMachine::Timeout,    "Day", 0 },
  { "Off + dark -> no change",          START_OFF, StateMachine::Dark,       "Off", 0 },
  { "Off + light -> lamp off; day",     START_OFF, StateMachine::Light,      "Day",    &L_Off },
  { "Off + movement -> lamp on; moving", START_OFF, StateMachine::Movement,   "Moving", &L_On },
  { "Off + no_movement -> no change",   START_OFF, StateMachine::NoMovement, "Off", 0 },
```

Listing 5

```
  { "Off + timeout -> no change",         START_OFF, StateMachine::Timeout,       "Off", 0 },
  { "Moving + dark -> no change",         START_MOVING, StateMachine::Dark,       "Moving", 0 },
  { "Moving + light -> lamp off; day",    START_MOVING, StateMachine::Light,      "Day",    &L_Off },
  { "Moving + movement -> no change",     START_MOVING, StateMachine::Movement,   "Moving", 0 },
  { "Moving + no_movement -> timing",     START_MOVING, StateMachine::NoMovement, "Timing", &Start_T },
  { "Moving + timeout -> no change",      START_MOVING, StateMachine::Timeout,    "Moving", 0 },
  { "Timing + dark -> no change",         START_TIMING, StateMachine::Dark,       "Timing", 0 },
  { "Timing + light -> lamp off; day",    START_TIMING, StateMachine::Light,      "Day",    &L_Off },
  { "Timing + movement -> no change",     START_TIMING, StateMachine::Movement,   "Timing", 0 },
  { "Timing + no_movement -> no change",  START_TIMING, StateMachine::NoMovement, "Timing", 0 },
  { "Timing + timeout -> lamp off; off",  START_TIMING, StateMachine::Timeout,    "Off", &L_Off },
  // terminate the tests
  { 0, START_DAY, StateMachine::Dark, 0, 0 } };
  for (unsigned i = 0; v[i].testTitle; i++)
      TestOneTransition (v[i]);
  return 0;
}
```

Listing 5 (cont'd)

state machine to get the application into an illegal state. But now that we had automatic transition and use case tests, it was very easy to change the design, and then prove it again.

Like all good examples, our simple security light has a bug, and the tests in Listing 5 do not reveal it. A carefully chosen extended sequence test would show that it does not restart the timer each time movement is detected while the lamp is still on. There should be a transition from timing to moving for the movement event. For example the sequence of events dark, movement, no_movement, movement, no_movement, timeout (i.e. a second movement while the lamp was still on) would result in lamp_on, start_timer, lamp_off, when it should cause lamp_on, start_timer, start_timer, lamp_off. Therefore we find that we need to add:

```
void Timing::Movement (void)
{
  context.changer.ChangeState(
     context.factory.MovingState());
}
```

Dealing with values

The overriding theme throughout has been to keep the application at arms length. Reducing the world outside the state machine to void (void) actions and events is an extreme simplification. In many cases it might appear to be a step too far. What about actions that need parameters? What about events that carry information? I would argue that a state machine deals with logic, not quantities. The code immediately surrounding the state machine, its immediate context, needs to deal with these quantities, and translate them to events and from actions on behalf of the state machine.

A very contrived example might be that our security light should control the brightness of the lamp according to the speed of movement. This simple control function would sit outside the state machine, storing speed and converting it to brightness when `LampOn` is called.

It would be feasible to allow properly encapsulated application logic inside the state machine, but validating the outputs would turn a simple test harness into a monster. I suspect the resulting pressure to revert to the bad old ways would be great when faced with such a complex task.

Further work

The example above has rather a lot of code for such a simple state machine. This is because it is a condensed version of a real implementation that was complex enough to warrant that approach. Now that we have a complete regression test harness, we could easily, and safely, refactor it into to a leaner and more concise version.

Something I have not tried yet is to add the `Actions` and `Events` interfaces to an existing state machine as a way to instrument it and help to bring it under better coverage of unit tests. Once a full set of transition tests has been written they provide enough of a safety net to allow refactoring.

Conclusions

This all started as an innocent attempt to write a 'nice clean' state machine using principles of coding to interfaces, and good separation of the roles of classes. It quickly turned into a revelation that 'there is a better way' to approach state machines and their testing in general. We all think we know how to write a state machine, but it is healthy to challenge this every now and then.

In the embedded world that spawned this work, faulty state machines are often the root cause of defects. It is the inability to test them effectively and repeatably that is the root cause of their unreliability. Eliminating this problem yields a significant improvement in the intrinsic quality of the code. By lifting the state machine up to the same level as more general application code, to which TDD is easily applied, it is no longer a poor relation, and can be treated equally.

I have only applied this technique a couple of times so far, but with no problems. I would be foolish to assume it will always work, but I look forward to confirming the assumption. ■

References

[Fowler] *Refactoring: Improving the Design of Existing Code*, Fowler, Beck, Brant, Opdyke and Roberts.

[Gamma] *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson and Vlissides.

[Griffiths] 'Separating Interface and Implementation in C++', Griffiths and Radford. http://www.twonine.co.uk/articles/SeparatingInterfaceAndImplementation.pdf

[Mackinnon] 'Endo-Testing: Unit Testing with Mock Objects', Mackinnon, Freeman and Craig. http://connextra.com/aboutUs/mockobjects.pdf

[Wikipedia1] http://en.wikipedia.org/wiki/Separation_of_concerns

[Wikipedia2] http://en.wikipedia.org/wiki/Cohesion_(computer_science)

Acknowledgments

Ric Parkin for guidance and support throughout the 'ordeal' of writing my first article.

My family, and the Overload review team, for constructive comments on the drafts.

di.fm, snakenet, and beer, for accompaniment whilst writing.

If You Can't See Me, I Can't See You

Knowing where the doors are is only the start. Stuart Golodetz works out what he can see through them.

In my last article, I talked about how to generate a set of portals (doorways) in a 3D world using a BSP tree. These portals can be very useful on their own (e.g. it's possible to write a *portal engine*), but they also have an application in determining a potential visibility relation between the empty leaves of the BSP tree, and it's that application that I will discuss in this article.

One of the early challenges faced when writing a 3D engine is how to avoid rendering your entire level when you can only see a small bit of it, since this slows your frame-rate to a crawl. This was a serious problem for the developers of the original *Quake* [Abrash]. The solution eventually adopted by id Software's John Carmack (and thereby popularized) was to precompute the set of empty leaves potentially visible from each empty leaf (also known as the PVS, or potentially visible set, of each leaf), using a method originally described in [Teller]. This solves the scale problem and allows you to have large levels with a high frame-rate (of course, it still doesn't mean that you can have large numbers of polygons all in the same room and maintain your frame-rate, but that's an entirely separate challenge). The idea is essentially that once the potentially visible sets for each leaf have been calculated, you can get away with rendering only the polygons in the leaves which are in the PVS of the current viewer's leaf: in other words, you render only the polygons that can potentially be seen from somewhere in the current room in which the viewer resides. This greatly reduces the number of polygons that need to be rendered each frame (see Figure 1: only the grey leaves need to be rendered from the specified viewer position).

Door-to-door

The challenge here is in how to calculate the PVS in the first place. Clearly, since it's computed offline, we could theoretically sit down for each level and fill in the $O(n^2)$ entries by hand, but this isn't a very attractive proposition when n (the number of empty leaves in the level) is even moderately large.

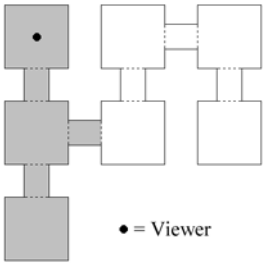

Figure 1

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

The method we use in practice involves the portals we generated last time. We observe that each leaf can potentially see the union of what its outgoing portals can potentially see: in other words, if I'm standing somewhere arbitrary in a room, I can potentially see exactly what can potentially be seen from all the openings (doorways, windows, skylights, etc.) leading out of that room. The leaf-to-leaf visibility problem can thereby be reduced to a portal-to-portal visibility problem (see Figure 2: note that only the portals relevant to the example have been shown), and it turns out that there is a direct method for generating a portal-to-portal visibility relation.

Antipenumbrae

The basic idea is shown in Figure 3. We imagine the source leaf as a volume light source, and note that its *antipenumbra* is the volume in which it can be partially seen (see Figure 3a). For our purposes, however, we will define an antipenumbra as a set of clip planes separating two portals (see Figure 3b): the analogy is obvious, but the latter definition is a more practically useful one.

To calculate the (portal) PVS for a given *source* portal (see Figure 4a), we start by considering all the *target* portals leading out of its neighbour leaf, i.e. the leaf into which it points (for example, the *neighbour leaf* of the source portal is the target leaf here). We add any target portal that can be seen from the source portal to the source portal's PVS, and use it to build an antipenumbra which represents the volume that can be seen from the source leaf through the source and target portals. Next, we clip the (outgoing) portals of the target portal's neighbour leaf (the *generator* leaf) to this volume (see Figure 4b). If a generator portal is not entirely clipped out of existence by this process, then we build a *reverse antipenumbra* from the generator portal to the target portal (see Figure 4c), and try and clip the source portal to that (this is an optimization that may allow us to clip further bits off our portals and speed up the algorithm). If the source

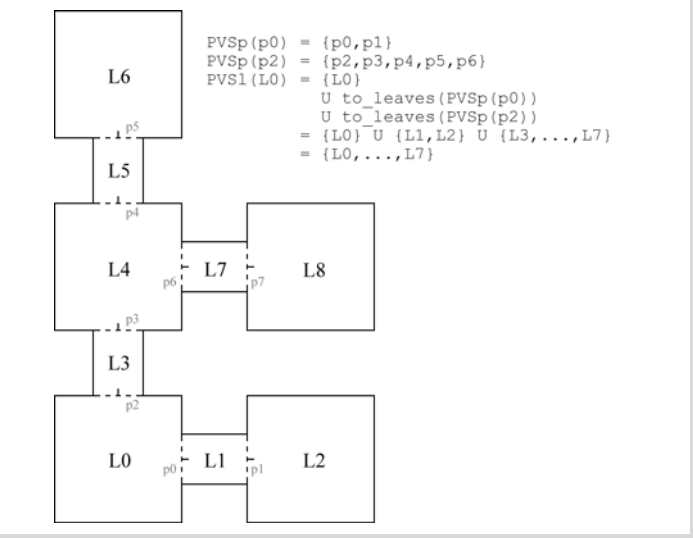

Figure 2

This greatly reduces the number of polygons that need to be rendered

Figure 3

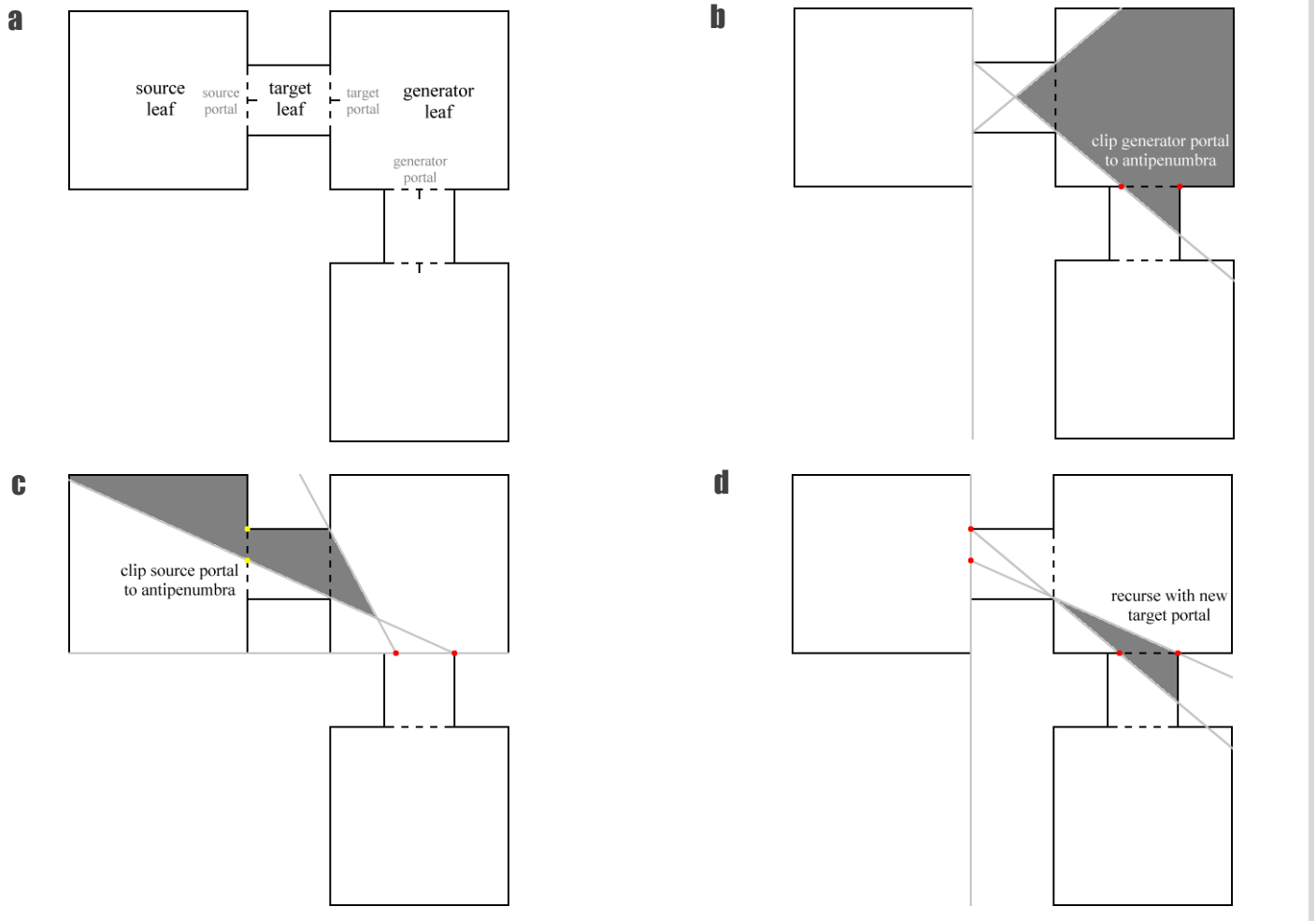

Figure 4

```
/**
Calculates the set of portals that are potentially
visible from the specified portal, and updates the
portal visibility table accordingly.

@param originalSource
The portal for which to calculate the PVS
*/
void VisCalculator::calculate_portal_pvs(
   const Portal_Ptr& originalSource)
{
  int originalSourceIndex =
     portal_index(originalSource);
  Plane originalSourcePlane =
     make_plane(*originalSource);
  std::stack<PortalTriple> st;
  // Initialise the stack with triples targeting
  // all the portals that can be seen from the
  // original source. If only part of a target
  // portal can be seen, we simply split it.
  const std::vector<int>& originalCandidates =
    m_portalsFromLeaf[neighbour_leaf(
    originalSource)];
  for(size_t i=0, size=originalCandidates.size();
     i<size; ++i)
  {
    Portal_Ptr target =
       m_portals[originalCandidates[i]];
    int targetIndex = originalCandidates[i];
    if((*m_portalVis)(originalSourceIndex,
       targetIndex) != PV_NO)
    {
      if((*m_classifiers)(originalSourceIndex,
         targetIndex) == CP_STRADDLE)
      {
        target = split_polygon(*target,
           originalSourcePlane).front;
      }
      st.push(PortalTriple(originalSource,
         Portal_Ptr(), target));
      (*m_portalVis)(originalSourceIndex,
         targetIndex) = PV_YES;
    }
  }
  // Run the actual visibility calculation process.
  while(!st.empty())
  {
    PortalTriple triple = st.top();
    Portal_Ptr source = triple.source, inter =
       triple.inter, target = triple.target;
    int targetIndex = portal_index(target);
    st.pop();
    Antipenumbra ap(source, target);
    const std::vector<int>& candidates =
       m_portalsFromLeaf[neighbour_leaf(target)];
    for(size_t i=0, size=candidates.size();
       i<size; ++i)
    {
      Portal_Ptr generator =
         m_portals[candidates[i]];
      int generatorIndex = candidates[i];
      // If this generator portal might be visible
      // from both the intermediate portal (if it
      // exists) and the target portal, then we
      // need to clip it to find out.
      if((!inter || (*m_portalVis)
         (portal_index(inter),
         generatorIndex) != PV_NO) &&
         ((*m_portalVis)(targetIndex,
```

Listing 1

```
          generatorIndex) != PV_NO))
      {
        Portal_Ptr clippedGen =
           ap.clip(generator);
        if(clippedGen)
        {
          Antipenumbra reverseAp(
             clippedGen->flipped_winding(),
             target);
          Portal_Ptr clippedSrc =
             reverseAp.clip(source);
          if(clippedSrc)
          {
             st.push(PortalTriple(clippedSrc,
                target, clippedGen));
             (*m_portalVis)(originalSourceIndex,
                generatorIndex) = PV_YES;
          }
        }
      }
    }
  }
  // Any portals which haven't been definitely
  // marked as potentially visible at this point
  // can't be seen.
  int portalCount =
     static_cast<int>(m_portals.size());
  for(int i=0; i<portalCount; ++i)
  {
    if((*m_portalVis)(originalSourceIndex,i) !=
       PV_YES)
    (*m_portalVis)(originalSourceIndex,i) = PV_NO;
  }
}
```

Listing 1 (cont'd)

portal is not entirely clipped out of existence either, then we add the generator portal to the source portal's PVS and recurse, using the generator portal as the new target portal (see Figure 4d).

Despite its recursive nature, the implementation of this algorithm is actually easier if done iteratively (see Listing 1). We store a stack of *portal triples*, (source, inter, target), where the *inter* portal is the portal that was the target when the *target* portal was the *generator*. When initialising the stack with the visible portals from the original target leaf, of course, these intermediate portals do not exist, so we just use null (the intermediate portals are actually only an optimization, in any case).

```
/**
Constructs an antipenumbra from a source portal to
a target portal.
For each antipenumbral plane p (except the source
plane), it is guaranteed to be the case that:
- classify_polygon_against_plane(*source, p)
     == CP_BACK
- classify_polygon_against_plane(*target, p)
     == CP_FRONT

@param source    The source portal
@param target    The target portal
*/
Antipenumbra::Antipenumbra(
   const Portal_Ptr& source,
   const Portal_Ptr& target)
{
  m_planes.push_back(make_plane(*source));
  // Note: In both cases here, source lies behind
  // the generated planes and target lies in front
  // of them.
```

Listing 2

```
  add_clip_planes(source, target, CP_BACK);
  // add planes from source to target, with source
  // behind them
  add_clip_planes(target, source, CP_FRONT);
  // add planes from target to source, with target
  // in front of them
}
/**
Adds clip planes which separate portal 'from' and
portal 'to'. The classifier specifies on which side
of the generated planes portal 'from' should lie.
@param from                    The from portal
@param to                      The to portal
@param desiredFromClassifier   The side of the
            planes on which portal from should lie
*/
void Antipenumbra::add_clip_planes(
   const Portal_Ptr& from, const Portal_Ptr& to,
   PlaneClassifier desiredFromClassifier)
{
  int fromCount = from->vertex_count();
  int toCount = to->vertex_count();
  for(int i=0; i<fromCount; ++i)
  {
    const Vector3d& a = from->vertex(i);
    const Vector3d& b = from->vertex(
       (i+1)%fromCount);
    for(int j=0; j<toCount; ++j)
    {
      const Vector3d& c = to->vertex(j);
      Plane_Ptr plane = construct_clip_plane(
         a, b, c);
      if(!plane) continue;
      PlaneClassifier cpFrom =
         classify_polygon_against_plane(*from,
         *plane);
      if(cpFrom == CP_BACK || cpFrom == CP_FRONT)
      {
        PlaneClassifier cpTo =
           classify_polygon_against_plane(*to,
           *plane);
        if((cpTo == CP_BACK || cpTo == CP_FRONT)
           && cpTo != cpFrom)
        {
        // If we get here, either cpFrom ==
        // CP_BACK && cpTo == CP_FRONT, or
        // vice-versa.
          if(cpFrom != desiredFromClassifier)
             m_planes.push_back(plane->flip());
          else m_planes.push_back(*plane);
            break;
        }
      }
    }
  }
}
/**
Returns the plane through a, b and c.
@param a     The first vector in the plane
@param b     The second vector in the plane
@param c     The third vector in the plane
@return      As stated
*/
Plane_Ptr Antipenumbra::construct_clip_plane(
   const Vector3d& a, const Vector3d& b,
   const Vector3d& c)
{
    Vector3d v1 = b - a;
    Vector3d v2 = c - a;
```

Listing 2 (cont'd)

```
    Vector3d n = v1.cross(v2);
    if(n.length_squared() < EPSILON)
       return Plane_Ptr();
    return Plane_Ptr(new Plane(n,a));
}
```

Listing 2 (cont'd)

Clipping

Having seen the core visibility calculation process, we need to look at how we construct an antipenumbra in the first place, and then clip to it. Constructing an antipenumbra is a relative simple process (see Listing 2). The idea is to add planes which separate the source portal from the target portal. To do this, we consider each edge on the source portal in turn, and iterate through the target vertices until we find a vertex such that the plane

```
/**
Clips the specified polygon to the antipenumbra.

@param poly     The polygon
@return         The clipped version of the polygon
*/
template <typename Vert, typename AuxData>
shared_ptr<Polygon<Vert,AuxData> >
Antipenumbra::clip(
   const shared_ptr<Polygon<Vert,AuxData> >& poly)
{
  typedef Polygon<Vert,AuxData> Poly;
  typedef shared_ptr<Poly> Poly_Ptr;
  Poly_Ptr ret = poly;
  for(std::vector<Plane>::const_iterator it=
     m_planes.begin(), iend=m_planes.end();
     it!=iend; ++it)
  {
     switch(classify_polygon_against_plane(
        *ret, *it))
     {
       case CP_BACK:
       {
         // The polygon is completely outside the
         // antipenumbra.
         return Poly_Ptr();
       }
       case CP_COPLANAR:
       {
         // The polygon lies on the antipenumbra
         // boundary and can't be seen properly
         // from the source.
         return Poly_Ptr();
       }
       case CP_FRONT:
       {
         // Nothing to clip: move onto the next
         // clip plane.
         break;
       }
       case CP_STRADDLE:
       {
         // Split the polygon across the clip plane
         // and keep the bit inside the
         // antipenumbra.
         ret = split_polygon(*ret, *it).front;
         break;
       }
     }
  }
  return ret;
}
```

Listing 3

```
/**
Performs the first phase of the visibility
calculation process. In this phase, portals which
obviously can't see each other (e.g. one portal is
fully behind another) are marked as such in the
portal visibility
table. This helps avoid a lot of unnecessary
clipping later on.
*/

void VisCalculator::initial_portal_vis()
{
  int portalCount =
     static_cast<int>(m_portals.size());
  m_portalVis.reset(
     new PortalVisTable(portalCount,
     PV_INITIALMAYBE));

  // Calculate the classification relation between
  // the portals. Specifically, classifiers(i,j)
  // will contain the classification of polygon j
  // relative to the plane of i.
  // Note:    This bit could potentially be
  // optimized if we required that portal pairs
  // occupied consecutive indices in the list (e.g.
  // if 1 were necessarily the reverse portal of 0,
  // etc.).
  m_classifiers.reset(
     new ClassifierTable(portalCount));
  for(int i=0; i<portalCount; ++i)
  {
    const Plane plane = make_plane(*m_portals[i]);
    for(int j=0; j<portalCount; ++j)
    {
      if(j == i) (*m_classifiers)(i,j) =
         CP_COPLANAR;
      else (*m_classifiers)(i,j) =
         classify_polygon_against_plane(
         *m_portals[j], plane);
    }
  }

  // Run through the portal visibility table and
  // mark (*m_portalVis)(i,j) as PV_NO if portal
  // i definitely can't see through portal j.
  for(int i=0; i<portalCount; ++i)
  {
    for(int j=0; j<portalCount; ++j)
    {
      if(j == i)
      {
         (*m_portalVis)(i,j) = PV_NO;
         continue;
      }
      // Note: Portals can only see through the
      // back of other portals.
      // If portal j is behind or on the plane of
      // portal i, then i can't see it.
      if((*m_classifiers)(i,j) ==
         CP_BACK || (*m_classifiers)(i,j) ==
         CP_COPLANAR) (*m_portalVis)(i,j) = PV_NO;
      // If portal i is completely in front of
      // portal j, then it's facing i and i can't
      // see through it.
      if((*m_classifiers)(j,i) == CP_FRONT)
         (*m_portalVis)(i,j) = PV_NO;
    }
  }
}
```

Listing 4

of the triangle it makes with the source edge separates the two portals. We then make sure the plane in question is facing away from the source portal and towards the target portal (for consistency: we could have done it the other way round as well), and add it to the list.

Clipping a portal to an antipenumbra is also relatively straightforward (see Listing 3). All we have to do is classify the portal against each clip plane in turn: if it's behind the plane, it's completely outside the antipenumbra, so we dump it; if it's in front of the plane, we need to carry on clipping it against the other planes; if it straddles the plane, we split it and keep the

```
/**
Performs the second phase of the visibility
calculation process, namely flood filling. This is
used to refine the initial portal visibility table
before calculating the final version, the aim being
to speed up the final calculation process.
*/
void VisCalculator::flood_fill()
{
  int portalCount =
     static_cast<int>(m_portals.size());
  for(int i=0; i<portalCount; ++i)
  {
    flood_from(i);
    // If any portals previously thought possible
    // didn't get marked by the flood fill,
    // then they're not actually possible and
    // need to be marked as such.
    for(int j=0; j<portalCount; ++j)
    {
      if((*m_portalVis)(i,j) == PV_INITIALMAYBE)
         (*m_portalVis)(i,j) = PV_NO;
    }
  }
}

/**
Peforms a flood fill from a given portal to refine
its approximate PVS before it is calculated for
real.

@param originalSource    The portal from which to
flood fill
*/
void VisCalculator::flood_from(int originalSource)
{
  std::stack<int> st;
  st.push(originalSource);
  while(!st.empty())
  {
    int curPortal = st.top();
    st.pop();
    if(curPortal != originalSource)
       (*m_portalVis)(originalSource, curPortal) =
       PV_FLOODFILLMAYBE;
    int leaf = m_portals[curPortal]
       ->auxiliary_data().toLeaf;
    const std::vector<int>& candidates =
       m_portalsFromLeaf[leaf];
    for(size_t i=0, size=candidates.size();
       i<size; ++i)
    {
      if((*m_portalVis)(originalSource,
         candidates[i]) == PV_INITIALMAYBE)
         st.push(candidates[i]);
    }
  }
}
```

Listing 5

front half (the bit potentially inside the antipenumbra); if it's on the plane, we treat it as if it were outside the antipenumbra, since it can't be seen properly from the source.

Pre-processing

The visibility calculation process is the main part of the algorithm, but before it takes place, we should first construct an initial portal-to-portal vis table to avoid doing lots of redundant work later. At a minimum (see Listing 4), we should fill in entries where:

- Portal *i* can't see portal *j* because *j* is behind or on the plane of *i*
- Portal *i* can't see portal *j* because the two portals are facing each other (portals can only see through the back of other portals)

We can do better than this initial portal vis, however, if we next perform a flood fill from each portal (see Listing 5): this allows us to eliminate even more possibilities before we get to the more expensive part of the vis process. These optimizations are essential to make the vis process run in a decent amount of time: without them, it can take ages. Once the flood-filling has taken place, we're ready to run the actual visibility calculation process described initially.

Post-processing: portal vis to leaf vis

Once the visibility calculations are complete, we have a portal-to-portal visibility table, but what we really need is a leaf-to-leaf table which will tell us which rooms can be seen from which other rooms. To convert the portal vis table to a leaf vis table, we use our earlier observation that a leaf can see precisely the union of whatever its (outgoing) portals can see. The process is slightly complicated by the fact that for implementation reasons, no portal is contained within its own PVS in the code (i.e. portal *i* can't see itself for coding purposes), but by and large the process is quite simple (see Listing 6). The result is a leaf-to-leaf visibility table we can use to speed up not only in-game rendering, but also lightmapping calculations for static level lighting.

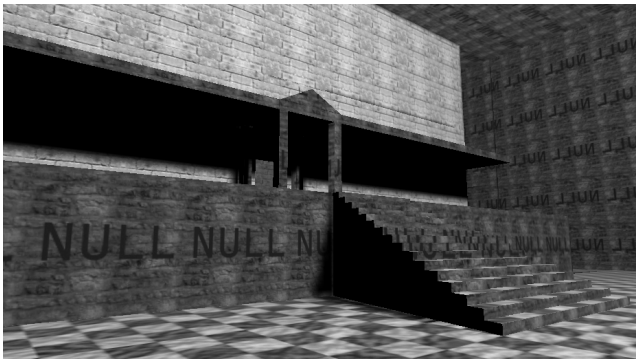

Figure 5

Conclusion

In this article, we've seen how to generate a leaf-to-leaf visibility table for a BSP-based level from a set of portals. This allows us to render larger levels at a reasonable frame-rate. As can be seen from Figure 5, using the PVS approach allows us to render larger levels at a reasonable frame-rate – because we can't possibly see most of the inside of this building, we don't need to render it. ■

References

[Abrash] *Graphics Programming Black Book (Special Edition)*, Mike Abrash, Coriolis Group Books, July 1997.

[Teller] *Visibility Computations in Densely Occluded Polyhedral Environments*, Seth Teller, PhD Thesis, October 1992.

```
/**
Constructs a leaf visibility table from the portal one. The result of the visibility calculation process
is actually this leaf visibility table, not the portal one, but the latter is more convenient during the
calculation process itself: we thus convert the one to the other once we've finished calculating.
*/
void VisCalculator::portal_to_leaf_vis()
{
  const int portalCount = static_cast<int>(m_portals.size());
  m_leafVis.reset( new LeafVisTable(m_emptyLeafCount, LEAFVIS_NO));
  for(int i=0; i<m_emptyLeafCount; ++i)
  {
  // Leaf i can see itself, plus the union of whatever leaves its portals can see.
    (*m_leafVis)(i, i) = LEAFVIS_YES;
    const std::vector<int>& ps = m_portalsFromLeaf[i];
    for(std::vector<int>::const_iterator jt=ps.begin(), jend=ps.end(); jt!=jend; ++jt)
    {
      const int j = *jt;
      // Leaf i can see the leaf pointed to by portal j (even though portal j can't see itself).
      (*m_leafVis)(i, m_portals[j] ->auxiliary_data().toLeaf) = LEAFVIS_YES;
      // Leaf i can see all the leaves pointed to by portals portal j can see.
      for(int k=0; k<portalCount; ++k)
      {
        if((*m_portalVis)(j,k) == PV_YES)
        {
          (*m_leafVis)(i, m_portals[k] ->auxiliary_data().toLeaf) = LEAFVIS_YES;
        }
      }
    }
  }
}
```

Listing 6

The Model Student: A Rube-ish Square (Part 2)

A rube-ish square embodies some simple group theory. Richard Harris explores its properties.

In the previous article in this series, presumably to counter a lifetime of the humiliation of being the only person in my socially maladjusted peer group unable to solve Rubik's Cube, I introduced the Rube-ish Square; a two dimensional version of that fiendish puzzle that even I might be able to solve. The Rube-ish Square is manipulated by rotating its rows and columns, with the element pushed out being returned on the opposite side, as illustrated in figure 1.

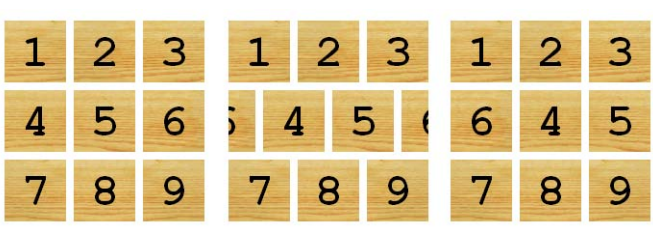

Figure 1

As you will no doubt recall, in an unforgivably maths heavy discourse, we exploited group theory to describe the properties of this simplified version of the cube. Remember that a group is defined as a set of elements together with an operator (usually denoted by $\circ$) which satisfies the following rules:

Closed: $\forall a, b \in G \Rightarrow a \circ b \in G$

Associative: $\forall a, b, c \in G \Rightarrow a \circ (b \circ c) = (a \circ b) \circ c$

Identity: $\exists i \in G$ such that $\forall a \in G\, a \circ i = i \circ a = a$

Inverse: $\forall a \in G \Rightarrow \exists a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = i$

Since they are not common programming notation, I feel that I should probably remind you that the upside down A means *for all*, the backwards E means *there exists* and the rounded E means *within*.

So translating this formal definition into English again, these rules mean that for a group G:

Closed: For all a and b within G, $a \circ b$ is within G

Associative: For all a, b and c within G, $a \circ (b \circ c) = (a \circ b) \circ c$

Identity: There exists a unique element i within G such that for all a within G, $a \circ i = i \circ a = a$

Inverse: For all a within G, there exists a unique element a^{-1} within G such that $a \circ a^{-1} = a^{-1} \circ a = i$

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.

Specifically we showed that the Rube-ish Square is fully described by the group representing the permutations of a vector of nine elements that involve swapping an even number of pairs of elements; namely the alternating group of degree nine.

Hence we were able to show that the Rube-ish Square has

$$\frac{9!}{2} = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{2}$$
$$= 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \approx 180,000$$

possible states.

This time we shall address a question that still remains unanswered for Rubik's Cube; what is the largest number of moves ever required to return the square to its initial state? And for this, as promised, we shall abandon the harsh mistress of mathematics and return to the warm safe embrace of C++.

Make with the C++ already

The first things we're going to need are some classes to represent the board and our interactions with it.

Listing 1 illustrates the **`move_type`** structure that we shall use to indicate how we wish to manipulate the square. The decision to use a structure rather than a class reflects this type's intended use to simply bind together the three pieces of data describing a move; the direction, the id of the row or column and the count of how many squares to the left or right, n. The constructors are straightforward, as shown in Listing 2.

The default constructor is required since we will eventually want to store objects of this type in a standard sequence container and it leaves the member data uninitialised since they have no meaningful default values.

Listing 3 illustrates the **`board`** class that we will use to represent the state of the Rube-ish square and manipulate it.

```
enum direction
{
  horizontal,
  vertical,
};

struct move_type
{
  direction dir;
  size_t    id;
  long      n;

  move_type();
  move_type(direction dir, size_t id, long n);
};
```

Listing 1

what is the largest number of moves ever required to return the square to its initial state?

```
move_type::move_type()
{
}

move_type::move_type(
   direction dir, size_t id, long n) : dir(dir),
                                       id(id),
                                       n(n)
{
}
```

Listing 2

The first thing to note is that by enabling construction with different lengths of side n this class represents a more general puzzle than the one we are currently investigating. The **move** and **undo** member functions provide the means to manipulate the state of the square, and the remaining public member functions the means to observe it.

The private member functions provide the state manipulation mechanism that will be used by both **move** and **undo**.

Finally, the member data **board_** and **buffer_** represent the board itself and a temporary area to assist in the manipulation of its rows and columns.

Listing 4 illustrates the implementation of the constructors.

```
class board
{
public:
  typedef size_t                    value_type;
  typedef std::vector<value_type>   board_type;
  typedef board_type::const_iterator
     const_iterator;
  board();
  explicit board(size_t n);
  size_t             size() const;
  void               move(move_type m);
  void               undo(move_type m);
  const board_type & data() const;
  const_iterator     begin() const;
  const_iterator     end() const;
  value_type         at(size_t row,
     size_t col) const;

private:
  void              move_row(size_t row, long n);
  void              move_col(size_t col, long n);
  value_type &       at(size_t row, size_t col);
  board_type board_;
  board_type buffer_;
};
```

Listing 3

```
board::board()
{
}
board::board(size_t n) : board_(n*n, 0),
                         buffer_(n, 0)
{
  static const size_t max_n = 1UL<<(
     std::numeric_limits<size_t>::digits/2);
  if(n>=max_n)  throw std::invalid_argument("");
  size_t i = 1;
  board_type::iterator first = board_.begin();
  board_type::iterator last  = board_.end();
  while(first!=last)*first++ = i++;
}
```

Listing 4

The default constructor need do nothing since both the member data have default constructors that do what we require; namely create a container of length 0.

The initialising constructor is of slightly more interest. This initialises the board with n^2 elements that will hold the row-wise elements of the board and initialises the working buffer with n elements that will hold the elements of a single row or column whilst they are being manipulated. It then proceeds to fill the board with integers counting up from 1 to create the initial, solved, state.

The **size** member function returns the length of side of the square and exploits the fact that the number of elements in the working buffer is equal to the number of rows and columns, as illustrated below:

```
size_t
board::size() const
{
  return buffer_.size();
}
```

The state access member functions are fairly straightforward, generally simply forwarding on to the underlying data type, as illustrated in Listing 5. The p1most complex of these, the **at** member function, simply maps the row and column onto a row-wise element of the **board_** member, throwing an exception if an attempt is made to access an element out of the board's limits.

The **move** and **undo** member functions simply forward on the requests to manipulate the state of the board to the private member functions **move_row** and **move_col** as shown in Listing 6.

As you can see, **undo** is simply the opposite operation to a **move**. Listing 7 illustrates how the private member functions perform the required state manipulation.

So we can see that the **move_row** and **move_col** member functions simply copy the row or column to be manipulated into the working buffer and then copy the elements back into the board starting from the required

```
const board::board_type &
board::data() const
{
  return board_;
}
board::const_iterator
board::begin() const
{
  return board_.begin();
}
board::const_iterator
board::end() const
{
  return board_.end();
}
board::value_type
board::at(size_t row, size_t col) const
{
  if(row>=size() || col>=size()) throw
     std::out_of_range("");
  assert(row*size()+col<board_.size());
    return board_[row*size()+col];
}
```

Listing 5

```
void
board::move(move_type m)
{
  if(m.dir==horizontal) move_row(m.id, m.n);
  else                  move_col(m.id, m.n);
}

void
board::undo(move_type m)
{
  if(m.dir==horizontal) move_row(m.id, -m.n);
  else                  move_col(m.id, -m.n);
}
```

Listing 6

```
void
board::move_row(size_t row, long n)
{
  if(row>=size()) throw std::out_of_range("");
  if(n<0)n = size() - (-n)%size();
  for(size_t i=0;i!=size();++i) buffer_[i] =
     at(row, i);
    for(size_t j=0;j!=size();++j) at(row,
       (j+n)%size()) = buffer_[j];
}
void
board::move_col(size_t col, long n)
{
  if(col>=size()) throw std::out_of_range("");
  if(n<0)n = size() - (-n)%size();
  for(size_t i=0;i!=size();++i) buffer_[i] =
     at(i, col);
  for(size_t j=0;j!=size();++j) at((j+n)%size(),
     col) = buffer_[j];
}
board::value_type &
board::at(size_t row, size_t col)
{
  assert(row<size() && col<size() &&
     row*size()+col<board_.size());
  return board_[row*size()+col];
}
```

Listing 7

offset, using modular arithmetic to ensure that we correctly wrap them around the left and right or top and bottom.

We've got a Rube-ish square and we're not afraid to use it

So, now we have the implementation of the board we are ready to use it to investigate the question of what is the largest number of moves required to return the square to its initial state from any other state.

To answer this, we shall represent the 180,000 or so states of the square as nodes, each of which is connected to 12 others by lines representing the 12 distinct rotations of rows and columns with which we transform the square from one state to another.

The technical name for this kind of structure is a graph, although the term network is also occasionally used. Figure 2 provides an illustration of a small graph of just 9 nodes.

Graphs played a big role in the early days of artificial intelligence [Russell95], where they were used to represent the states of an artificial world, often a game such as chess, and the actions that transform it between states. A number of algorithms were developed to efficiently search these graphs for a series of transformations that would lead from an initial to a desired state.

Unfortunately, the real world rarely provides us with such well-defined states and transformations and so these algorithms never led to the goal of machines that demonstrate intelligent behaviour. Within the field of computer gaming, however, we often do have rigidly defined rules and goals and consequently these types of algorithms still form the basis for many game playing machines. A particularly striking example is IBM's chess playing machine, Deep Blue, which defeated world champion Gary Kasparov in 1997 by searching through some 200 million states per second [Hsu02].

If we disallow returning to a node already visited, the paths through a graph may be represented by a simpler structure; namely a tree. Numbering the nodes in our example graph, the first three levels of the tree of all paths starting at the uppermost node are illustrated in Figure 3.

Strictly speaking, we could also represent paths that allowed nodes to be revisited with a tree, although if we wished to examine all possible paths it would need an infinite number of levels.

Structuring the problem in this form is highly suggestive of a way to search the paths through the graph. Starting at the first node we can recursively descend the leftmost branch until we reach the bottom, then reverse up a level and descend the next leftmost, and so on until we have traversed every non-returning path starting at the first node. Figure 4 illustrates the full 9 steps of the search.

This approach is known as depth-first search, and is one of the simpler graph traversal algorithms. Its principal drawback is that, when searching for a path to a node with a particular property, it often follows a large number of fruitless paths before discovering a relatively short one that has the desired result. Specifically, if the rightmost branch leads to a node with

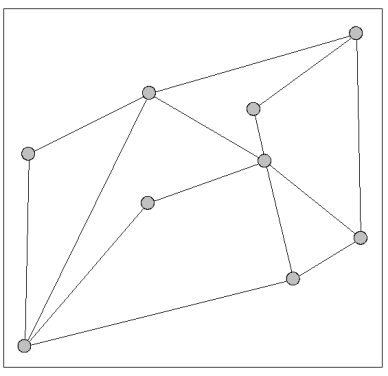

Figure 2

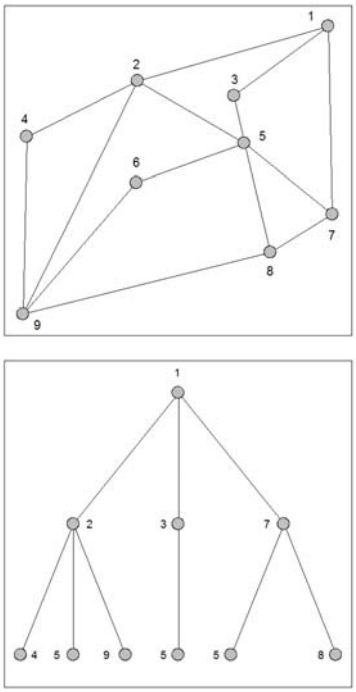

Figure 3

the desired property after one step, depth-first search is going to waste a lot of time on the leftmost branches.

As a result, it is generally not the algorithm of choice for many problems. That said, the alternatives require maintaining a great deal more state, trading computational expense for memory usage.

The first thing we should note is that $9^9 \approx 2^{28.53}$, so we can generate a unique identifier for any given state of the 3 by 3 square that fits inside a 32 bit integer. The scheme we shall use is to represent the state as a 9 digit base 9 number:

$$\begin{aligned} id = {} & (n_{11}-1)\times 9^8 + (n_{12}-1)\times 9^7 + (n_{13}-1)\times 9^6 \\ & + (n_{21}-1)\times 9^5 + (n_{22}-1)\times 9^4 + (n_{23}-1)\times 9^3 \\ & + (n_{31}-1)\times 9^2 + (n_{32}-1)\times 9^1 + (n_{33}-1)\times 9^0 \end{aligned}$$

where n_{ij} is the number in row i and column j of the square in the given state. Note that we must subtract 1 from each element since they range from 1 to 9, rather than from 0 to 8.

Listing 8 illustrates the addition of such an `id` member function to the board class.

This scheme is fairly wasteful of bits, since for the 3 by 3 square can only exhibit even permutations of the integers from 1 to 9; approximately $2^{18.47}$ states. We could certainly do better by identifying each permutation with no gaps, which would allow us to use a sequential rather than an associative container to record the set of already visited states. Unfortunately, this would be rather more complicated to implement.

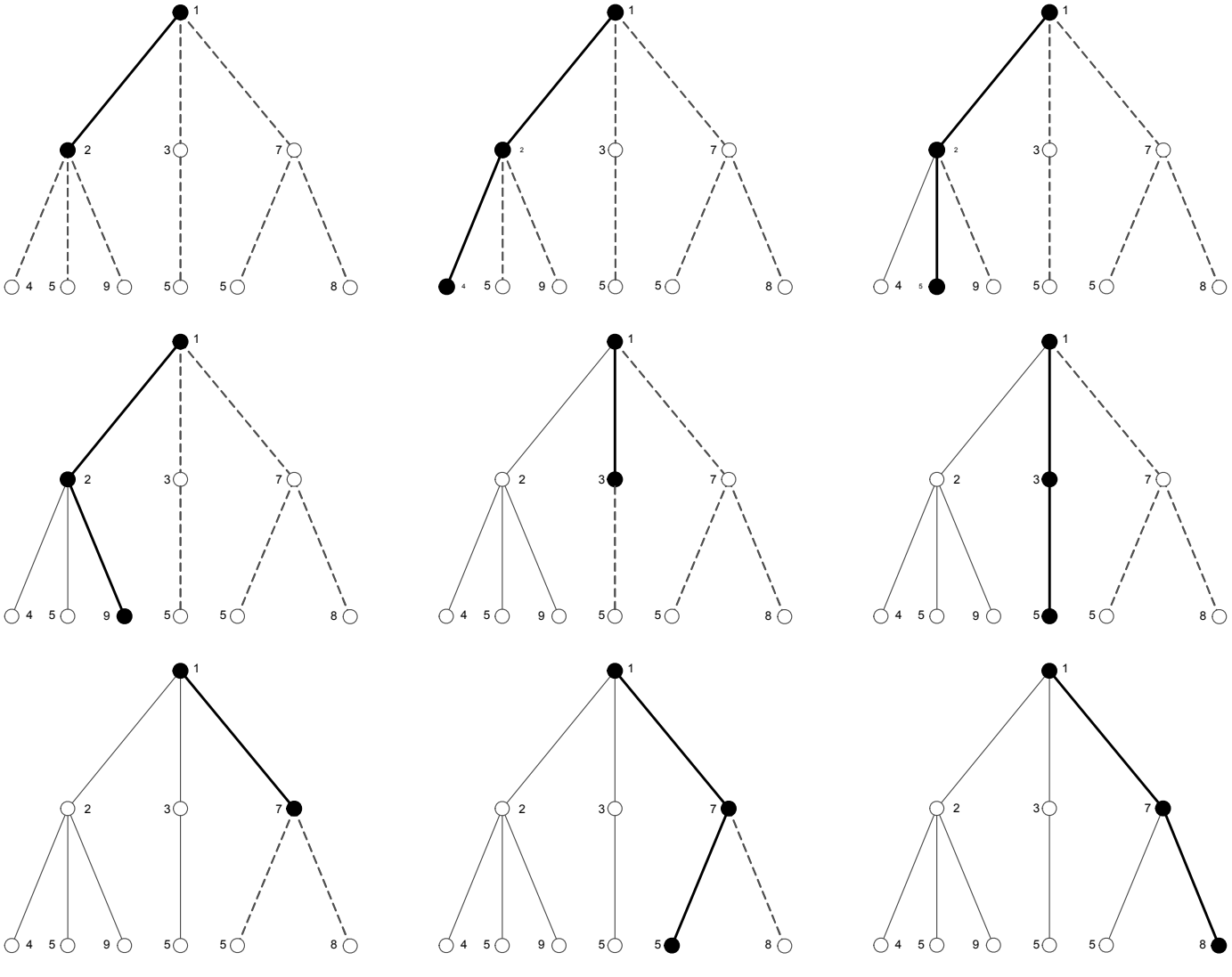

Figure 4

```
class board
{
public:
  ...
  unsigned long id() const;
  ...
};
unsigned long
board::id() const
{
  if(size()>3)  throw std::out_of_range("");
  unsigned long i = 0;
  const_iterator first = begin();
  const_iterator last  = end();
  while(first!=last)
  {
    i *= size()*size();
    i += *first++ - 1;
  }
  return i;
}
```

Listing 8

Furthermore, as you may have guessed from the first line of the function, it won't work for board sizes greater than 3. This is because the result of the formula won't fit into a 32 bit integer if the board size is 4 or greater.

In fact, even the total number of possible states won't fit into a 32 bit integer for a 4 by 4 board, so the more efficient scheme wouldn't do us any good either. For a 5 by 5 board, we wouldn't even have enough space if we had 64 bits to work with.

Figure 5 illustrates just how quickly the number of states and the upper bound of our encoding scheme grow with the size of the board, *n*..

Clearly we're going to run out of bits pretty quickly no matter how large our integers are. Giving up the encoding scheme and using vectors of integers to represent the board state doesn't help much either since instead of running out of bits, we'll run out of memory.

Given the enormous difficulty we will clearly have representing the full set of states of large boards, I'm reasonably happy to accept the weaknesses of this encoding scheme. In any event, the board we are studying is just 3 by 3, so it's something of a moot point.

So, all that said, exactly how inefficient is a depth first search going to be?

Unfortunately, it's going to be pretty damn well inefficient since we may very well visit all 180,000 states during traversal of the leftmost sub-tree of the root node, only to replace them with shorter paths later in the search.

Thankfully, we can dramatically improve our algorithmic performance by setting an upper bound on the depth of the tree.

To do this, we need only describe a naïve scheme for solving the puzzle; namely find the worst case for returning each number to its initial position in turn, whilst leaving previously returned numbers in place. I shall simply assert what these worst cases are and how many moves they take, so you may wish to take a moment to confirm that I've got them all right.

n	States	Encoding
2	2^5	2^8
3	2^{17}	2^{29}
4	2^{44}	2^{64}
5	2^{83}	2^{116}
6	2^{138}	2^{186}

Figure 5

- For 1, the worst case is when it is in neither the first row nor the first column, requiring 2 moves.
- For 2, whilst leaving the 1 in the correct position, the worst case is if it is in the first row and third column, requiring 3 moves to ensure that 1 remains in the correct position.
- For 3, whilst leaving the 1 and the 2 in the correct positions, the worst case is when it is in neither the first row nor the third column, requiring 2 moves.
- For 4, whilst leaving the previous elements in their correct positions, the worst case is if it is in the third row and the first column, requiring 4 moves.
- For 5, whilst leaving the previous elements in their correct positions, the worst case is if it is in the third row and the second column, requiring 4 moves.
- For 6, whilst leaving the previous elements in their correct positions, the worst case is if it is in the third row and the third column, requiring 4 moves.
- For 7, whilst leaving the previous elements in their correct positions, the worst case is if it is in the second or third column, requiring 1 move.
- At this point, 8 and 9 *must* be in their correct positions since if they weren't we'd have an odd permutation of the square and, as we so tediously demonstrated last time, that just isn't possible.

So, that gives us an upper bound of just 20 moves, significantly reducing the worst case number of branches we must traverse.

Adding a typedef for a `std::map` from the board state id to the number of moves required to get there from the initial state to the board, we are ready to begin searching for the state that requires the most moves to return the square to its initial state.

The final piece of the puzzle

Listing 9 illustrates the definition of the record of moves and the declaration of the `max_moves` static member function that we'll use to find the state requiring the most moves to solve.

Note that the `hint` argument will be used to inform the algorithm of any upper bound on the depth of the search tree that we have been able to deduce. For our analysis this will of course be 20 moves. We'll use the default value of 0 as an indication that we haven't done our homework and are unaware of any such upper bound.

The `max_moves` function simply initialises a board and a move count and forwards on to another function, examining its result for the worst case, as illustrated in Listing 10.

So, the real work is being done by the as yet undefined `all_moves` function, given in Listing 11.

This simply checks whether the current state of the board has been visited before and if not, or if so but after a greater number of moves, adds it to the record of states and the number of moves required to reach them. It then recursively takes a further step by calling the `next_moves` member

```
class board
{
public:
  ...
  static size_t max_moves(size_t n,
     size_t hint = 0);
  ...
private:
  typedef std::map<unsigned long, size_t>
     move_counts_type;
  ...
};
```

Listing 9

```
size_t
board::max_moves(size_t n, size_t hint)
{
  move_counts_type move_counts;
  board b(n);
  all_moves(b, move_counts, 0, hint);
  move_counts_type::const_iterator first =
     move_counts.begin();
  move_counts_type::const_iterator last  =
     move_counts.end();
  size_t result = 0;
  while(first!=last)
  {
    if(first->second>result) result =
       first->second;
    ++first;
  }
  return result;
}
```

Listing 10

function, which in turn relies upon a **next_move** function, both of which are illustrated in Listing 12.

So **next_moves** simply iterates over the rows and columns of the board and calls **next_move** to continue the search with each move that can be applied to them. Note that the number of squares through which we rotate the rows and columns by starts at 1 rather than 0, since the latter would trivially have no effect upon the square. We don't have to worry about the inner loop running into trouble if the board has a size of 0, since in that case the outer loop will terminate immediately.

Now we are finally ready to calculate the worst case number of moves required to solve our Rube-ish Square, and upon doing so we discover that it is, rather anti-climactically, 8.

Before I leave you again, I thought it might be rather fun to give you a couple of examples of 8-move states of the square in Figure 6, so that you can have a crack at solving them.

As a final thought, I'd like you to imagine that the elements of the square are also labelled on the underside and that, in addition to rotating its rows and columns, we allow the whole square to be turned over, swapping the

```
class board
{
  ...
private:
  static void all_moves(board &b,
     move_counts_type &move_counts,
     size_t depth, size_t hint);
  ...
};
void
board::all_moves(board &b,
   move_counts_type &move_counts,
   size_t depth, size_t hint)
{
  size_t id = b.id();
  move_counts_type::value_type current_count(id,
     depth);
  std::pair<move_counts_type::iterator, bool>
     current = move_counts.insert(current_count);
  if((current.second || depth<current.first
     ->second) && (hint==0 || depth<=hint))
  {
    if(!current.second) current.first->second =
       depth;
    next_moves(b, move_counts, depth, hint);
  }
}
```

Listing 11

```
class board
{
  ...
private:
  static void next_move(board &b, move_type m,
     move_counts_type &move_counts,
     size_t depth, size_t hint);
  static void next_moves(board &b,
     move_counts_type &move_counts,
     size_t depth, size_t hint);
  ...
};
void
board::next_move(board &b, move_type m,
   move_counts_type &move_counts,
   size_t depth, size_t hint)
{
  b.move(m);
  all_moves(b, move_counts, depth+1, hint);
  b.undo(m);
}
void
board::next_moves(board &b,
   move_counts_type &move_counts,
   size_t depth, size_t hint)
{
  for(size_t i=0;i!=b.size();++i)
  {
    for(size_t n=1;n!=b.size();++n)
    {
      move_type h(horizontal, i, n);
      move_type v(vertical,   i, n);
      next_move(b, h, move_counts, depth, hint);
      next_move(b, v, move_counts, depth, hint);
    }
  }
}
```

Listing 12

1st and 3rd, the 4th and 6th and the 7th and 9th pieces. This is an odd permutation, so trivially doubles the number of possible states. The question is, what effect does it have on the worst case number of moves?

Until next time, dear reader, happy puzzling. ■

Acknowledgements

With thanks to Astrid Byro, Keith Garbutt and John Paul Barjaktarevic for proof reading this article.

References and further reading

[Hsu02] Hsu, Feng-hsiung, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*, Princeton University Press, 2002

[Russell95] Russell, S. & Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995

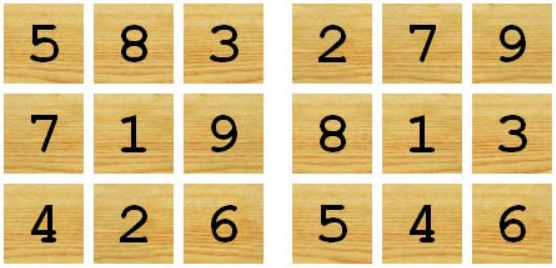

Figure 6

On Management: Product Managers

Product Management is a poorly understood activity. Allan Kelly sheds some light on its mysteries.

Of all the roles that play a part when software is created only one is essential: Coders. You can get by without Testers, either because the Coders are so good you don't need to test or through sheer bloody mindedness. And if you are very lucky the Coders will be able to work out what is required directly from the people who want the software. But generally speaking it helps to have someone decide what is needed from software before coding begins.

This is not to make a case for big requirements document – or indeed any documentation. Documentation may help, but it is only a medium for communication. The spoken word is another, and requirements are often better communicated by a conversation. The advantage of conversation is that it is a two way process. The listener can ask questions during a conversation and they can resume the conversation at a later date if they need clarification – both difficult to do with a document.

However the requirements are communicated, somebody needs to decide what they are. Scrum calls this person the Product Owner. It is the person (or persons) who decides what is needed, what is to be created, and what the priorities are. However Scrum has nothing to say about how the Product Owner finds or decides what is required.

Similarly XP has a role called the Customer. However, again XP has nothing to say about how this role is performed. On the original XP team the customers' role was to translate what the current system did so the C3 team could make the new system do something similar.

For any system of size, knowing what needs to be created and what the priorities are is a process of discovery. Unless a team is building a replacement system with no added functionality requirements, then requirements discovery is complicated and involved process.

Fortunately there are people with these skills and experience. In most organizations these people are either known as Business Analysts or Product Managers. Their role is to decide what needs to be built and communicate it to the developers.

Unfortunately the roles of Business Analyst and Product Manager are not always clear. More unfortunately still the role Project Manager is often mistaken for that of a Business Analyst or Product Manager.

Lesson 1: The term Product Owner is an alias for Product Manager or Business Analysts. In some organizations the Product Owner will be a Product Manager and in others they will be a Business Analyst. For the development team the net result is the same: a Product Owner with the authority and legitimacy in the organization to decide what needs to be done.

Allan Kelly realised after years at the code-face 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, helping teams adopt Agile methods and improve development practices and processes. He can be contacted at allan@allankelly.net and http://allankelly.blogspot.net.

Product management in the UK

I spent the first ten years of my professional career working in and around London. I worked for a variety of companies but I never met a Product Manager.

Then I went to work in Silicon Valley for a bit. Here it seemed I met Product Managers all the time. Not just in the office but socially. There are a lot of Product Managers in Silicon Valley.

Since then I've made a point of understanding what Product Managers do, I've worked as Product Manager myself, I've been on Product Manager training and I've read a lot about what they do and how they do it. And I've become convinced that good Product Management is one of the key differentiators between successful software product companies and the unsuccessful ones.

Without someone who knows what customers value in a product and why they use it then all attempts to improve and enhance the product are just a shot in the dark. Success and failure come down to luck.

This person, whether they actually have the title Product Manager or not, needs the authority and legitimacy inside the organization to direct the product and guide its development.

This is vital for software products – software which people are expect to pay for directly or indirectly. Companies that create software for their own use, or develop bespoke software for customers who pay for the software to be written have a very different user base. When this software is delivered customers have little choice but to use it. The software has already been paid for.

There are lots of successful Silicon Valley software companies using Product Managers and acting as role models for new software companies. In the UK there are fewer successful software companies and fewer role models. While things have improved in the UK the Product Manager role is still not as widely known as it needs to be.

In this article and the next article I would like to try and unravel these roles and examine them in a little more depth.

Product Managers and Business Analysts are different

Basically, Product Managers work at companies that create software that sells in a market. They are outward facing. They know they need to seek out customers and find out what they need to make their lives better. Product Managers are concerned with competitor products and change outside the company.

In contrast, Business Analysts work at companies that develop software for their own internal use, or on specific developments for other companies, which will be used internally. Thus they are normally found in corporate IT departments and external service provider (ESP) companies.

Business Analysts look inwards, they look at the operations and needs inside a company. They know exactly who their users are, indeed, in some cases there may only be one user. When Business Analysts look outside the company they are looking at suppliers as alternatives to development not as competitors in the market.

it is the hundreds, thousands, of small **decisions made every day** that make the difference

Lesson 2: The Product Manager and the Business Analyst roles are different. Not enough people appreciate the difference.

What does the Product Manager do?

The Product Manager role is summarised in Figure 1. The most obvious activity for the Product Manager is talking to the development team. This is a two way conversation during which the Product Manager tells the team what is needed.

Or rather, the Product Manager specifies what the goal is: they should not propose solutions, and neither should they get involved in technical design. However, they may review several proposed solutions and express a preference for one when the solution makes a difference to how customers relate to the product. For example, a Product Manager would not review alternative database schema designs, but they would review and comment on different user interface designs. One is visible to the final customer and the other invisible. As shown in Figure 1, product managers sit in the centre of many conversations.

Lesson 3: Product Managers work with the development team, customers, senior managers, sales and keep watch on the wider market, including competitors.

Product Managers continue the dialogue about what is needed as the product is built. Developers come and ask for clarification, 'Do you mean this? or that?' Whenever there is a decision that makes a difference to how the product functions to the customer, Product Managers need to be consulted.

On products with a non-trivial user interface, development teams should include a user interface designer. In matters of UI design and operation, they will deputise for the Product Manager on UI decisions. However in the absence of a UI designer these kind of decisions should be made by a Product Manager rather than a developer.

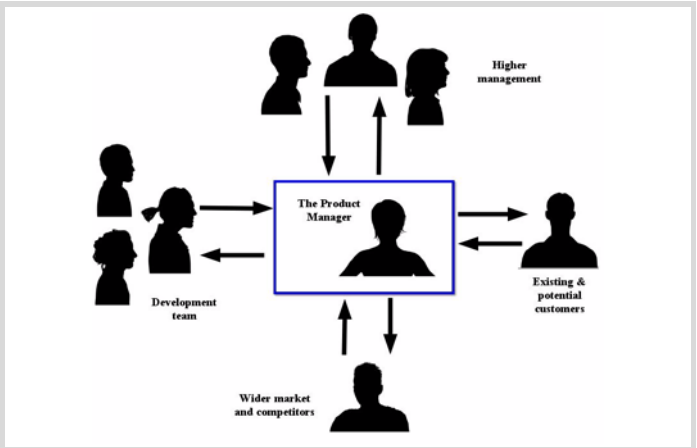

Figure 1

In addition, Product Managers talk to the development team about what is technologically possible. Both about the development in hand at the moment, and about for future work. Product Managers need to reconcile what the technology can do with what customers need. To do this they need to stay abreast of technology developments.

It also means that Product Managers need to be in an ongoing dialogue with customers. Both customers who have already bought the product and are using it and potential customers, the kind of customers the company wants to have.

Product Managers need to visit, observe and research the customer market as a whole. They need to identify the problems customers have for which the software product is a solution. They need to understand how this relates to the customers' tasks, problems and daily routines. Importantly they need to understand what will make the customer part with money.

There are various ways a Product Manager can do this. When customers are not known, the first task is to find them. Once the customers are known there needs to be a dialogue. The Product Manager may ring them up and talk to them directly. Better still is a face-to-face visit.

Lesson 4: Product Managers need to be in regular contact with customers.

Traditional market research methods like surveys and focus groups are part of the Product Manager's toolkit. So, too, is win–loss analysis. Product Managers visit (without a salesman) customers who have bought the product, and potential customers who have not bought the product. The objective is not to make a sale, or turn around a failing one but to understand why one customer bought and another one didn't.

Product Managers need to look at the wider market and at competitors. They need to attend trade shows and read trade journals, watch competitors' websites and talk to customers about competitors.

Once information is gathered, Product Managers combine it all into product roadmaps and strategies. These they present to senior managers, but before they can do so they need to understand what senior managers are trying to achieve with the company. What is the company strategy? And how does this product play a part?

Lesson 5: Product Managers both follow corporate strategy and influence it.

There are even more tasks that naturally fall to Product Managers but are not so core. They may be asked to visit customers with sales staff to talk about the product, present product roadmaps. and listen to customer issues.

Product Managers are never very far from Product Marketing and are often the public face of the product. (Product Marketing is described below.)

They may be asked to speak at conferences or to the press, they may need to advise on how the product is presented in marketing literature.

Time

What should be obvious by now is that there is a lot for a Product Manager to do. When the work is done well it really can make the difference between a big success and an also-ran product. What isn't so clear is that it's too easy for the role to be squeezed and become ineffective.

Squeezing happens for two reasons. Firstly, everyone in the company, from CEO to Receptionist, has an opinion on what the product could do, should do and what will make money. The Product Manager will have these feelings too, but they need to find the facts to support their decisions. Only when armed with these facts can they make rational decisions and deny others their requests.

But getting facts brings us to the second reason for the squeeze: time. With so much to do, Product Manager time is at a premium. Visiting a customer may involve two days of travel for a two hour meeting. This might not seem like an effective use of time but without these meetings the Product Manager will be blind to customer needs.

> **Lesson 6:** It costs to acquire information; but without this information and facts money will be wasted elsewhere chasing guesses and opinions.

One solution is simply to have more Product Managers working on a Product. How many you need depends on the nature of your product, the number and type of customers, how new the product is and many other factors.

As a rough guide, I recommend one Product Manager for every three to seven developers. If a product has been around for a while, the market is stable, and no big new innovation is planned then one Product Manager can probably keep seven developers busy.

If however the product is new, the product is innovative, the market is developing rapidly, and there are many needs to be addressed then one Product Manager for three developers is more realistic.

> **Lesson 7:** Appoint one Product Manager for every 3 to 7 developers. Without a Product Manager to guide them, a team will be guessing, success will be based on luck.

When a development team is larger and multiple Product Managers are needed then things become more complicated because different Product Managers must co-ordinate their work. One answer is to have a Tactical and a Strategic Product Manager – TPM and SPM respectively.

The SPM does most of the customer visits, most of the conversations with management and does the long term roadmaps. The TPM spends more of their time working with the developers, helping sales and the near term roadmaps. Importantly the SPM and TPM talk regularly, they should sit next to each other. This arrangement also makes it easier to arrange visits to customers and debrief afterwards. (By now you may have realised that Product Managers need to travel a lot.)

In-bound versus out-bound marketing

Another way to refactor the Product Manager role is to ensure it does not involve any outbound marketing. Strictly speaking this is the role of Product Marketing.

In the purest form, marketing is about both discovering customer needs and communicating solutions to the customer. Discovering needs is in-bound marketing, it's about finding out what is needed. Once there is a solution available the focus is then on communicating about the product. This second form of marketing is what most people think of as marketing, but strictly speaking this is out-bound marketing and is known as Product Marketing.

MRD–PRD model

Within product management circles there is an accepted way of creating requirements documents. Of course, organizations and even individuals differ in exactly what they expect from each document, and what they say in each document but as a general rule it goes like this.

First it starts with a Business Case, or Business Requirements Document – a BRD for short. This outlines the business opportunity and how it might be exploited, and what the return might be. For example, it might say: 'Telesales staff (who have the problem) waste a lot of time calling to customers who do not wish to receive telesales calls (the problem to be solved); a product which automatically prevents numbers on the national "do not call" list being called (solution) would increase productivity (benefit).'

The BRD might also give some indication of market size, number of potential customers and so on. Sometimes there is no need to write a BRD, perhaps because the product is a development of an earlier one, or because the business case is implicit in the company's purpose.

Next comes the Market Requirements Document, the MRD. This is the document that examines what the market needs. The MRD will take any BRD as a starting point and develop the ideas further. It discusses the problems a product would need to address, who would buy the product, makes some suggestions on the functionality needed, what the performance characteristics would need to be and examine the potential competition.

In some cases the BRD may be merged into the MRD, forming the first few pages of the MRD. When it exists as a stand-alone document, it should be a short document.

Neither the BRD or MRD addresses the features of the product in depth. The MRD might say 'User access needs to be controlled' and might discuss current market standards but it would not go into detail. The MRD would also lay out the constraints on the product: 'Needs to sell for less than $100 to be competitive.'

Next comes the Product Requirements Document. The PRD translates the requirements in functionality into features with essential details. The PRD may also refine statements on the performance and constraints.

Again, sometimes the MRD and the PRD get merged together. In my view this is a mistake because the MRD should focus on the need, and the PRD is the start of the solution. While the BRD (if it exists) and MRD should be created by a the business side (Product Manager or higher executive), the PRD is the start of the engineering process. As such the engineering group should contribute to, or even write the PRD, with the Product Manager.

What happens after the PRD is less well defined. Sometimes the PRD is enough to get started on. Other times the PRD may be further refined as an functional specification – sticking with the convention this would be called a Functional Requirements Document or FRD. Software developers might like to respond to the PRD with a design document.

If this all sounds very waterfall-ish that's because it is. One document leads to another and eventually some code gets written. However the idea of separating the market need and business case from the product requirements and the functionality to meet those needs holds in iterative and evolutionary models. These documents need to become living documents. Market needs change and so the MRD can be expected to change over time.

Document creation needs to overlap. The BRD kick-starts the work with a small team, Product Managers continue to develop the MRD and engineers respond with PRD changes or directly in code. As the scale of the work becomes apparent the team may increase in size.

Product Management is, at heart, an in-bound marketing role. However the role is often caught up with out-bound marketing, communicating about the product. This is particularly true at small companies who might not be able to afford an additional person. As a company grows these responsibilities should be passed to a dedicated Product Marketing Manager.

The Product Marketing function, often filled by Product Marketing Managers, is concerned with communicating to customers that the product exists, the benefits of the product, and changes to the product. This is done

through advertising, public relations, press releases, online websites and other media.

Lesson 8: Outbound Product Marketing is a different and distinct activity from inbound Product Management.

(Just to complicate things, Product Manager as described in this article and as practiced by successful software companies is different to the Product Manager found in many non-technology companies. The role of Product Manager at a company like Proctor & Gamble is an out-bound marketing role, one usually involved with brand management.)

A Product Manager cannot be a Developer

A Product Manager needs to be technically knowledgeable, they need to understand what technology can and cannot do and they need to understand their products. Thus it is not uncommon to find Developers moving into a Product Manager role. However anyone taking this route must accept that their coding days are behind them for several reasons.

Firstly, perhaps unsurprisingly: time. As already outlined the Product Manager role is a full time role. It is wrong to think a Product Manager will have time to talk to customers, decide what is needed, survey the competition and so on, then change hats and write the code.

That said I have come across developers who, in the absence of a Product Manager, take on many of the duties. Often this happens unofficially, and often the developer doesn't do the complete role. While this is understandable, it is a sign that there is a role to be filled.

Lesson 9: In the absence of an official Product Manager, others are likely to fill the role.

Secondly the priorities of the two roles conflict. Good developers have an empathy for the code base and the product architecture. The code speaks to them. It says things like 'refactor me' and 'add a database abstraction layer'. Good developers hear these messages and do their best to give the code what it wants.

Product Managers also have a relationship with the product but their empathy needs to be with the customers. The messages they hear are 'simplify the UI' and 'Give me the product on Oracle'.

Asking one person to keep all this in their mind, and empathise equally with customers and code is too much. It would be like asking one person to present two personalities.

Anyone who tries to fill both roles will inevitably tend towards one side or the other. For a Developer moving into product management they continue to listen to the code when they should be listening to the customer.

Conclusion

Although Product Managers have the word 'Manager' in their title, they are not managers in some the senses of the word. They manage a thing, not people. Their power rests on their legitimacy and knowledge rather than their direct authority.

Without a Product Manager directing the direction of development the success of a commercial product is down to luck and chance. Product Managers are the people who take the luck out of developing software products.

Product Management is not a misunderstood role, it is simply an overlooked role. Too many software companies are either ignorant of what good product management can do for them or they simply believe that developers know best.

Fungible – a correction

Back in the first article of this series I used the word fungible, I described it like this: 'Money is, economists like to tell us, fungible. Which is another way of saying it can be exchanged for other things very easily. Money can be exchanged for resources such as a new developer, thereby increasing our resources.'

Actually, I got it wrong. My Oxford English Dictionary says: 'fungible ... replaceable by another identical item'.

So while a £20 note is fungible – one £20 is the same as another – the exchanging of the note for 15 minutes of developer services is not. Exchanging money for services or goods is an example of liquidity.

I was trying to convey the idea that, because money can substitute (via liquidity) for many things, there is no need to consider different types of resources.

Apologies to all, and thanks Edmund Stephen-Smith for point out my mistake.

Lesson 10: If you work at a software company which sells its products – either shrink wrapped or online via the web – to multiple customers then you need Product Managers working with the development team.

Further reading

Unfortunately, there is still a lack of good books on product management. Any aspiring Product Manager should certainly read *The Inmates are Running the Asylum* [Cooper04] and *Crossing the Chasm* [Moore99] – and probably the sequel, *Inside the Tornado* [Moore05] is also worth reading. Clayton Christensen's *Innovator's Dilemma* and *Innovator's Solution* are also to be recommended [Christensen97] [Christensen03].

Although I've not read *Tuned In* [Stull08] and *Beyond Software Architecture* [Hohmann03], I've heard good reports about both. Some background in marketing and business strategy – especially in technology – is also a good idea.

Over the last five years I have been writing a set of patterns about the software business. Many of these patterns relate to the product development process and product management function. For example, the patterns include 'Single Product Company, Product Roadmap' [Kelly08] and 'Same Customer, Different Product' [Kelly07]. These patterns and some more can be found at http://www.allankelly.net/patterns/business.html. ■

References

[Christensen97] Christensen, Clayton M. 1997. *The Innovator's Dilemma*. Boston, Mass.: Harvard Business School Press.

[Christensen03] Christensen, Clayton M. and Michael E. Raynor. 2003. *The Innovator's Solution*: Harvard Business School Press.

[Cooper04] Cooper, A. 2004. *The Inmates Are Running the Asylum*: Que.

[Hohmann03] Hohmann, Luke. 2003. *Beyond software architecture : creating and sustaining winning solutions*. Boston: Addison-Wesley.

[Kelly07] Kelly, A. 2007. 'More patterns for Technology Companies.' In *EuroPLoP*, eds. L. Hvatum and T. Schümmer. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

[Kelly08] Kelly, A. 2008. 'Business Strategy Patterns for Product Development.' In *EuroPLoP* (European conference on Pattern Languages of Program design). Iresee, Germany.

[Moore99] Moore, G.A. 1999. *Crossing the Chasm*. Capstone publishing.

[Moore05] Moore, G.A. 2005. *Inside the Tornado*: Collins.

[Stull08] Stull, Craig, Phil Myers and David Meerman Scott. 2008. *Tuned in : uncover the extraordinary opportunities that lead to business breakthroughs*. Hoboken, N.J.: J. Wiley & Sons.

An Introduction to FastFormat (Part 2): Custom Argument and Sink Types

A library should be customisable and have good performance. Matthew Wilson shows how to achieve both.

This article, the second in the series on the new FastFormat formatting library, discusses various ways in which the library can be extended. In doing so, it reveals some important aspects of the FastFormat design, and discusses some of the mechanisms by which it achieves its high performance characteristics.

Introduction

The two main subjects of the article are custom argument types – usually in the form of user-defined classes – and custom sink types. It would be natural to discuss the use of custom argument types first, because that is likely to be the most common way in which the library is extended. Both subjects involve performance considerations, but examining how custom sinks are defined will give you a better appreciation for the internals of FastFormat, and make clearer some of the design decisions that come into play when defining custom argument adaptors.

Before I start on either of those, though, I'm going to have a bit of a soapbox moment, to get you in the mood.

Performance? Really??

One of my fun-but-in-a-different-way day jobs is writing and/or conducting technical interviews for senior engineers, architects and development managers on behalf of my clients. One of my favourite, most revealing, questions is a seemingly simple parsing scenario, with loose similarities to the examples we've already seen. It seems to matter little what programming language candidates wish to employ, the devil is in the detail of the algorithms they choose to use, and the adjustments they make in answer to my (ceaselessly) changing requirements.

Let's look again at the main Professor Yaffle example discussed in part 1, wearing a language-independent programming hat.

```
std::string  forename  = "Professor";
char         surname[] = "Yaffle";
int          age       = 134;
std::string  result;

AcmeFormat(result, "My name is %0 %1; I am %2
years old; call me %0", forename, surname, age)
```

Picture a whiteboard, some coloured pens, a penetrating and relentless interviewer, and just a few minutes to come up with an efficient replacement strategy. If I were to ask you in what form, rather than how, you would effect the replacements, you may well come up with the following:

1. Take the **`"My name is "`** bit of the format
2. Take the forename
3. Take the **`" "`** bit of the format
4. Take the surname
5. Take the **`"; I am "`** bit of the format
6. Take the age and turn it into a string
7. Take the **`" years old; call me "`** bit of the format
8. Take the forename again
9. Concatenate them all together

Suppose I then gave you a large-square-grided sheet of paper (memory), a pair of scissors (**`malloc()`**) and some pens (**`memcpy()`**) and asked you to produce a 1xN rectangle piece containing exactly the result described above. In that case, I hope that your algorithm would be:

1. Calculate the sum of all part lengths to determine the total length of the result, which in this case is 11 + 9 + 1 + 6 + 7 + 3 + 20 + 5 = 58
2. Cut out a 1x58 square sheet of paper.
3. Copy the exact number of characters for each part of the resulting statement, starting each at the square after the last one in the preceding part.

I hope that you would not use an algorithm such as:

1. Cut out a 1x12 piece and write **`"My name is "`** in it.
2. Repeat Step 1 for the remaining 7 individual parts to give a total of eight pieces.
3. Take the first two pieces from this pile, and determine their combined length.
4. Cut out a piece of this size.
5. Copy in the contents of the first piece.
6. Copy in the contents of the second piece, directly after the last square occupied by the first piece's contents, e.g. **`"My name is Professor"`**
7. Discard the first two pieces in the pile – **`"My name is "`** and **`"Professor"`** – and place the new piece on top of the pile.
8. Repeat steps 3–6 (a further 6 times) until only one piece, the result, remains.

Sound like fun? Definitely not! Not to mention the amount of wasted paper. The second algorithm would not help you in the interview. So why is it that we're prepared to tolerate such things operating in many (if not most) of the world's largest and most important software systems?

The main reason that FastFormat is so much faster than its peers is that they follow the second algorithm, and it follows the first. Neither the FastFormat core nor the application layer do any intermediate memory allocation, copying or concatenation. Naturally, the question is how. That will be discussed as we go through the remaining parts of this series.

Custom sink types

Before we look at the sink mechanism, we have to discuss string slices. A slice is a view onto a contiguous area of memory. In the case of a

Matthew Wilson is a development consultant specialising in performance and robustness, and author of numerous articles, books, and open-source software libraries. He prides himself on writing faster software than anyone else, yet is abashed that his books are slower to write (and to sell) than everyone else's. He can be contacted at stlsoft@gmail.com.

(character) string, a slice is a read-only view onto an array of character elements forming part, or the whole, of a string.

In FastFormat, a string slice is represented by the type **ff_string_slice_t**, which is defined as a length + a pointer. (**ff_char_t** is **char** or **wchar_t**, for multibyte or wide string builds, respectively.)

```
struct ff_string_slice_t
{
  size_t      len; // # of chars
  ff_char_t*  ptr; // ptr to 1st char
};
```

This is the only type understood by the FastFormat core. Furthermore, pointers to arrays of slices are the type that sinks receive to represent the replacement/concatenation results. Let's look at how this works. Say we want to write a sink for the Windows **OutputDebugString()** API function:

```
void OutputDebugString(TCHAR const* s);
```

There are three important things to note about this function. First, it takes a single C-style string, so whatever we pass to it must be nul-terminated. It should also be non-NULL, by the way.

Second, the function outputs to a debug stream potentially shared by all threads on the host system. Although the function itself operates atomically, it means that if you try to do things such as the following it is possible, indeed likely, that the three parts of your output will be interleaved with output from other threads/processes on the system.

```
void fn(char const* str);
{
  OutputDebugString("fn(");
  OutputDebugString(str);
  OutputDebugString(")\n");
  … // rest of fn()
```

This means that we must combine all parts of the statement, including new-line, if required, before sending it to **OutputDebugString()**.

Finally, what appears as a single function is actually a **#define** to one of the following two actual functions, depending on whether the UNICODE pre-processor symbol is defined.

```
void OutputDebugStringA(char const* s);
void OutputDebugStringW(wchar_t const* s);
#ifdef UNICODE
# define OutputDebugString  OutputDebugStringW
#else /* ? UNICODE */
# define OutputDebugString  OutputDebugStringA
#endif /* UNICODE */
```

Standard string sinks

Before we get into the implementation of the sink for **OutputDebugString()**, I'd like to walk you through the stock sink support, which works with any type, particularly **std::basic_string**,

```
// File: fastformat/shims/action/fmt_slices/
// generic_string.hpp in namespace
// fastformat::sinks

template <typename S>
S& fmt_slices(
  S&       sink
, int      flags
, size_t   total
, size_t   numResults
, ff_string_slice_t const*  results
)
{
  sink.reserve(sink.size() + total + 2);
  { for(size_t i = 0; i != numResults; ++i)
  {
    ff_string_slice_t const& slice = results[i];
    if(0 != slice.len)
    {
      sink.append(slice.ptr, slice.len);
    }
  }}
  if(flags::ff_newLine & flags)
  {
    const ff_string_slice_t newLine =
       fastformat_getNewlineForPlatform();
    sink.append(newLine.ptr, newLine.len);
  }
  return sink;
}
```

Listing 1

which provides the **reserve()** and **append()** methods defined in Listing 1. The function is an action shim [XSTLv1] – a composite shim type that both controls and may modify its primary parameter – called **fastformat::sinks::fmt_slices**, meaning that it is an overload of a function named **fmt_slices()** defined in the namespace **fastformat::sinks**.

First consider the signature. It's a function template – to support either **std::string** or **std::wstring**, depending on the ambient character encoding of the build – with five parameters. The first parameter, **sink**, is a mutating reference to the sink, which allows for it to be changed. The second parameter is a bit-mask of flags that moderate the formatting operation: currently two stock flags are defined:

- **fastformat::flags::ff_newLi**
- **fastformat::flags::ff_flush**

The final two parameters, **numResults** and **results**, define an array of string slices representing all the constituent parts of the resulting statement.

The third parameter, **total**, is the total length of all the string slices. It is an advisory, to enable optimisation in any allocation that may have to be

```
// file: fastformat/sinks/OutputDebugString.hpp
// in namespace fastformat::sinks
class OutputDebugString_sink
{
  … // T.B.D.
};
inline OutputDebugString_sink& fmt_slices(
  OutputDebugString_sink&   sink
, int     flags
, size_t  total
, size_t  numResults
, ff_string_slice_t const*  results
)
{
  … // T.B.D.
}
```

Listing 2

performed to assemble the results. In this case, it facilitates the call to **reserve()**, which means that there will only be, at most, one memory allocation associated with preparing the result. This prescience regarding required memory is one of the secondary reasons why FastFormat is fast.

The loop is pretty straightforward: each slice is appended to the sink via the **append()** method, specifying the pointer and length. It is **very important** that length is always used along with pointer, because (i) the pointer may not point to a nul-terminated string, and (ii) the pointer may actually be **NULL** when the length is 0. In the case of **std::basic_string**, the requirement of 21.3.5.2/6 and 21.3.1/6 necessitate the conditional test against length.

The only remaining task of the function body is to handle the request – via **fmtln()** or **writeln()** – for a new-line to be written. Once again, this is achieved using the sink's **append()** method. It writes from a special instance of **ff_string_slice_t** returned from the helper function **fastformat_getNewlineForPlatform()**. This function returns a slice of one or two characters in length, depending on whether the platform's newline is "\r", "**\r\n**" or "**\n**". (Since specifying the wrong value to **reserve()** does not result in a functional error, the use of the magic number 2 is valid because I 'know' that it cannot be more than that. If the newline slice ever changed, then I would change the implementation of the string sink accordingly.)

Finally, the sink reference is returned, allowing for concatenation of format statements, if required (which is seldom, by the way).

```
std::string sink;
ff::fmt(ff::fmt(sink, "{0}", 1), "{0}", 2);
// sink => "12"
ff::write(ff::write(sink2, 1), 2);
```

OutputDebugString sink

Armed with this knowledge, let's look at the **OutputDebugString()** sink. One striking difference to the **string** sink is that **OutputDebugString()** is a function: there are no instances of a class to use as sinks. So we must make one (Listing 2).

This would be used as follows:

```
void fn(char const* str);
{
  ff::sinks::OutputDebugString_sink sink;
  ff::fmtln("fn({0})", str);
  … // rest of fn()
```

The first question to ask when implementing the class and the associated action shim function is whether the logic should go in the class, or in the function. Some sink classes, such as **speech_sink**, are stateful, remembering options that moderate their output behaviour. Therefore, for consistency, I always place the logic in the class, and implement the action shim function in terms of the class's **write()** method, as shown in Listing 3. If you're sure your sink won't need to be stateful, feel free to do it all in the function and just have a simple empty struct for the sink type.

```
class OutputDebugString_sink
{
public: /// Member Types
  typedef OutputDebugString_sink   class_type;
public: /// Construction
  OutputDebugString_sink()
  {}
public: /// Operations
  class_type& write(int flags, size_t total,
     size_t numResults,
     ff_string_slice_t const* results);
};
inline OutputDebugString_sink& fmt_slices( … )
{
  return sink.write(flags, total, numResults,
     results);
}
```

Listing 3

Now that we know the structure of the code, all that remains is to implement the **write()** method. Remembering our first two design constraints – the need to supply a non-NULL nul-terminated C-style string, and the shared final output destination – it's clear that we cannot follow the example of the standard string sink and write out a slice at a time. Rather, we must write into an intermediate buffer, appending a nul-terminator, and a new-line if required.

The STLSoft libraries [STLSOFT] have a class template called **auto_buffer** [EVAB] [IC++], which provides a middle ground between the speed of stack allocation and the flexibility of heap allocation. Simply, it has a fixed internal buffer from which it attempts to fulfil requests for memory. If the request is too large, it is satisfied from the heap. In many circumstances, this can lead to dramatic performance improvements [EVAB] [IC++]. The more you learn about FastFormat, the more you'll see **auto_buffer** lending a high-performing hand in even the most unexpected places. Use of **auto_buffer** is the third reason why FastFormat is fast. (One point to note: even though it shares much with the interface of **std::vector**, it is important to realise that it is not a container, and you must not attempt to use it assuming any more intelligence than it is documented to have. See section 16.2 of [XSTLv1] for more discussion on this point.)

So what does this have to do with our new sink? Well, one of the utility functions that comes with the library, **concat_slices()**, takes an **auto_buffer** instance, along with the array of slices, and concatenates them all together, resizing the buffer as necessary. We can use this to simplify the implementation of **write()** (Listing 4).

In this case, we need to know exact lengths, so we get hold of the platform newline at the start. We then calculate the exact length required for the **auto_buffer**, which will throw **std::bad_alloc** if the request cannot be satisfied. If all goes well, **concat_slices()** is invoked, and the slice contents are written into the buffer.

The last parts of the preparation are to write in the newline, if requested, and to nul-terminate the string. Then we just invoke **OutputDebugString()**. Q.E.D.

Except … as some eagle-eyed readers may already have pondered, this is assuming consistency between the presence/absence of **UNICODE** and **FASTFORMAT_USE_WIDE_STRINGS**, the pre-processor symbol whose definition dictates whether the FastFormat library is built for wide strings or left as multibyte strings. It's possible that a user may demand that FastFormat be wide string while not correspondingly defining **UNICODE**. Putting aside whether you (or I) think this is meaningful/desirable, we can easily side step the whole issue, by simply using overloading.

Along with the sink and the action shim, the fastformat/sinks/OutputDebugString.hpp header also defines the helper structure **OutputDebugString_helper**, to which we can defer the decision-making (Listing 5).

```
#include <fastformat/util/sinks/helpers.hpp>

class_type& OutputDebugString_sink::write(
  int flags
, size_t total
, size_t numResults
, ff_string_slice_t const* results
)
{
  const ff_string_slice_t newLine =
fastformat_getNewlineForPlatform();
  stlsoft::auto_buffer<ff_char_t> buff(
     1 + total + ((flags::ff_newLine & flags) ?
     newLine.len : 0));
  fastformat::util::concat_slices(buff,
     numResults, results);
  if(flags::ff_newLine & flags)
  {
    ::memcpy(&buff[total], newLine.ptr,
       sizeof(ff_char_t) * newLine.len);
    total += newLine.len;
  }
  buff[total] = '\0';
  OutputDebugString(buff.data());
  return *this;
}
```

Listing 4

And that's the final version. It writes atomically, provides nul-termination and, if requested, appends a new line, and works regardless of the character encodings of the library and/or the application.

Atomicity

Just a last word on atomicity: In Part 1 [FF1] I made just criticism of the other libraries that do not support atomic output, and observed that it is essential that it is the library, and not the user, that handles it. You can see from the two action shim implementations we've considered that it is possible to avoid paying the cost of copying and concatenating in a context where atomicity is a moot point, while being able to easily apply it otherwise. In this respect, FastFormat supports the best of both worlds, with the simple caveat that the writer of a sink must do the right thing.

Custom argument types

Probably the most common way in which a user would wish to extend a formatting library is in adding support for custom types. The remainder of this article will illustrate how that is done.

```
 struct OutputDebugString_helper
{
  static void fn(char const* s)
  {
    ::OutputDebugStringA(s);
  }
  static void fn(wchar_t const* s)
  {
    ::OutputDebugStringW(s);
  }
};
. . .
class_type& OutputDebugString_sink::write( . . . )
{
  . . .
  buff[total] = '\0';
  OutputDebugString_helper::fn(buff.data());
  return *this;
}
```

Listing 5

```
class superhero
{
public: /// Member Types
  typedef std::string     string_type;
  typedef superhero       class_type;
public: /// Construction
  superhero(string_type const& name, int weight,
     int strength, int goodness)
    : name(name)
    , weight(weight)
    , strength(strength)
    , goodness(goodness)
  {}
private:
  class_type& operator =(class_type const&);
public: /// Member Variables
  const string_type   name;
  const int           weight;
  const int           strength;
  const int           goodness;
};
```

Listing 6

First, we need a user-defined type to be passed as an argument to the format statements. Listing 6 shows the definition of a simple superhero type.

Now let's try and insert one into some format statements, in Listing 7.

If you compile this, you'll get a number of errors along the lines of:

```
. . ./fastformat/internal/generated/
helper_functions.hpp(160) : error:
'stlsoft::c_str_data_a' : none of the 4
overloads could convert all the argument types .
. . while trying to match the argument list
'(const superhero)' . . ./fastformat/internal/
generated/helper_functions.hpp(160) : error:
'stlsoft::c_str_len_a' : none of the 4 overloads
could convert all the argument types . . . while
trying to match the argument list '(const
superhero)'
```

The compiler has failed to find matching string access shim overloads – of `stlsoft::c_str_data_a()` and `stlsoft::c_str_len_a()` – for the superhero type. This is to be expected, since we haven't yet defined any.

In point of fact, it's not actually necessary to define string access shims for our type. Indeed, there are several options for working with a user-defined type:

- Inserters (functions or classes)
- Type filters
- String access shims [IC++] [XSTLv1]

With the first two approaches, what you define is instead an intermediary type for which string access shims are already defined. An obvious type would be `std::string` (or `std::wstring`, for wide string builds), although we'll see later that there are better options.

```
#include <fastformat/ff.hpp>
#include <fastformat/sinks/ostream.hpp>
…
superhero thing("The Thing", 200, 99, 100);
superhero batman("Batman", 100, 80, 95);
ff::writeln(std::cout, "Ben Grimm is ", thing);
ff::fmtln(
   std::cout, "Bruce Wayne is {0}", batman);
```

Listing 7

```
std::string edna(superhero const& hero)
{
  std::string result;
  char        num[21];
  result += hero.name;
  result += " {weight=";
  result.append(num, sprintf(num, "%d",
     hero.weight));
  result += ", strength=";
  result.append(num, sprintf(num, "%d",
     hero.strength));
  result += ", goodness=";
  result.append(num, sprintf(num, "%d",
     hero.goodness));
  result += '}';

  return result;
};
```

Listing 8.1

The hero format

Let's stipulate that the format for a super-hero is as follows:

```
<name> {weight=<weight>, strength=<strength>,
   goodness=<goodness>}
```

Inserter function

Let's start by building the simplest option, an inserter function. When defining stock inserters (for Pantheios [PAN], anyway, since I've not done the FastFormat ones yet), it's easy to think of names, such as `pantheios::integer`, `fastformat::real`, and so forth. When it comes to your own types, it can be a little trickier, since you want to be succinct, and you can't give the inserter the same name unless you put it into a different namespace (which will hinder succinctness). For this example, when dressing up a bunch of superheros I think the name is obvious. Listing 8.1 shows a first attempt.

Well, that will work, but it's ugly, not terribly maintainable, and not in the slightest bit localised. Furthermore, it's not efficient, and not strictly robust (although no `sprintf()` should ever return a negative result in this case).

We can handle most of the performance issue just by adding in a call to `reserve()` before the first concatenation, taking into account the length of the name, the literal fragments, and the maximum sizes of the three integer attributes (Listing 8.2)..

But that still leaves us with the other problems. What we really need here is a good formatting library . . .

I hope you're ahead of me here. We can rewrite this in terms of one of the FastFormat APIs. If we want to maximise performance, and we are able to forego localisation, then we'd use `FastFormat.Write`, as in Listing 8.3.

Note that we don't return the result of `ff::write()`, because the compiler doesn't know that the returned value is actually result, and we don't want to stymie its ability to apply the named return value optimisation [IC++].

If it must be localisable, then we'd use `FastFormat.Format`. Note the double `{{` to produce the literal `{` in the result; see Listing 8.4.

```
std::string edna(superhero const& hero)
{
  std::string result;
  char        num[21];
  result.reserve(hero.name.size() + 32 + (
     3 * 20));
  result += hero.name;
  . . .
```

Listing 8.2

```
std::string edna(superhero const& hero)
{
  std::string result;
  ff::write(result, hero.name, " {weight=",
hero.weight
          , ", strength=", hero.strength, ",
goodness=", hero.goodness
          , "}");
  return result;
};
```

Listing 8.3

```
std::string edna(superhero const& hero)
{
  std::string result;
  ff::fmt(result, "{0} {{weight={1}, strength={2},
goodness={3}}"
        , hero.name, hero.weight
        , hero.strength, hero.goodness);
  return result;
};
```

Listing 8.4

And to actually localise, we could use a resource bundle, as in Listing 8.5.

I hope you'll see how convenient is the statelessness of FastFormat, allowing us to implement an inserter function using the library itself.

With any of these inserter functions, we can now successfully format a superhero:

```
ff::writeln(std::cout, "Ben Grimm is ",
   edna(thing));
ff::fmtln(std::cout, "Bruce Wayne is {0}",
   edna(batman));
```

The obvious little fly in the ointment is that `edna()` has to be called explicitly, and this intrudes slightly on the expressiveness of our application code.

Inserter class

If/when I write a future article on Pantheios, I'll explain the reason why inserter classes are preferred, since they can employ lazy evaluation to forego paying costs if logging is not enabled. With FastFormat, arguments to format statements are always used, so the use of classes is unnecessary, and functions suffice. (This is good, because they're a fair bit simpler.)

Type-filter

If we want to be able to have the original formatting statements work (without `edna()`), we have two options. The more specific of these, the *filter-type mechanism*, provides compatibility that only works with FastFormat. It involves overloading a conversion shim [IC++] [XSTLv1]

```
#include
   <fastformat/bundles/properties_bundle.hpp>
ff::properties_file_bundle const& getAppBundle();
std::string edna(superhero const& hero)
{
  std::string result;
  ff::properties_file_bundle const& bundle =
     getAppBundle();
  ff::fmt(result, bundle["superhero.format"]
        , hero.name, hero.weight
        , hero.strength, hero.goodness);
  return result;
};
```

Listing 8.5

```
// file: fastformat/internal/generated/
api_functions.hpp
// in namespace fastformat

template<typename S
, typename A0, typename A1
>
inline S& writeln(S& sink
, A0 const& arg0, A1 const& arg1)
{
  return
fastformat::internal::helpers::write_outer_helper_
2(
      sink
    , flags::ff_newLine
    , fastformat::filters::filter_type(arg0,
&arg0, static_cast<ff_char_t const volatile*>(0))
    , fastformat::filters::filter_type(arg1,
&arg1, static_cast<ff_char_t const volatile*>(0))
    );
}
```

Listing 10

– a primary shim type that involves conversion of instances of heterogeneous types to a single type – called **fastformat::filters::filter_type**, meaning that it is an overload of a function named **filter_type()** defined in the namespace **fastformat::filters**.

Let's look at how this can be implemented for our superhero type in Listing 9.

The body of this should be immediately recognisable, as it's a straight lift from **edna()**. (I hope she's not aggressively litigious!) What is probably not so recognisable is the strange function signature of the shim overload. What are the purposes of the second and third arguments, both of which are unused?

To understand these parameters we must peek a little inside the FastFormat application layer templates, which are responsible for translating your nice, heterogeneous application layer statements into arrays of string slices. Consider the two parameter overload of the **fastformat::writeln()** API function shown in Listing 10.

The third parameter simply informs the conversion shim overload which character encoding it's being asked to work with. The purpose of the parameter is to allow different implementations for multibyte and wide string forms. In our example, we only defined the char-form, and it will only work in a multibyte build. We could instead have actually specified the third parameter as **ff_char_t const volatile***, which would have allowed us to be encoding-agnostic.

The purpose of the second parameter is considerably less obvious. To understand this, we need to have a review of C++ law (and lore).

```
// in namespace fastformat:: filters
inline std::string filter_type(
  superhero const& hero
, superhero const*
, char const volatile*
)
{
  std::string result;
  ff::fmt(result, "{0} {{weight={1}, strength={2},
      goodness={3}}"
      ,  hero.name, hero.weight
      ,  hero.strength, hero.goodness);
  return result;
}
```

Listing 9

The pedantic pointer idiom

In C++, matching functions takes into account implicit conversions. Consider the following class hierarchy, and three functions that dump out information on instances of these classes.

```
class superhero
{};
class extrasuperhero
  : public superhero
{};

void dump(extrasuperhero const* xhero);
void dump(superhero const* hero);
void dump(void const* pv);
```

If we declare instances of the two hero types, and pass their addresses to **dump()**, all will be well.

```
superhero hero;
extrasuperhero xhero;

dump(&hero);
dump(&xhero);
```

If we now remove the **dump(extrasuperhero const*)** overload, the code still compiles, but **xhero** will be dumped in the form of a **superhero**. Since an **extrasuperhero** *isa* **superhero**, this is probably ok, although that may not be so. If we now also remove the **dump(superhero const*)** overload, the code still compiles, but both heros will just be dumped like raw pointers. An ignominious end for such great men (or women)!

Readers who read part 1 [FF1] will recognise this as the source of the design flaws that prevent IOStreams and Boost.Format from being adequately robust. The way around this is to define a single function template, **dump()**, and to pass off the work to appropriately defined two-parameter worker functions, as follows:

```
template <typename T>
void dump(T const* t)
{
  dump(t, &t);
}
void dump(extrasuperhero const* xhero,
   extrasuperhero const**);
void dump(superhero const* hero,
   superhero const**);
void dump(void const* xhero, void const**);
```

We add a second pointer parameter that is the address of the first parameter. By doing so, we sidestep any implicit conversions in the primary parameter, because the implicit conversions do not apply at an extra level of indirection. Just because **superhero const*** may happily convert to **void const***, **superhero const**** will not implicitly convert to **void const****.

So, if we now remove the **dump(extrasuperhero const* xhero, extrasuperhero const**)** overload, we will find that the request to **dump &xhero** will not compile.

I call this technique the **pedantic pointer idiom**. (For alliterative purposes, I ache to call it the **pedantic pointer pattern**, but it can't really claim to be a pattern.) It is used in several STLSoft components, and in my commercial work, to enforce 100% type-safety. Clearly it finds good use in the FastFormat application layer, facilitating infinite extensibility while enforcing total robustness.

The one issue is that each time you derive from **superhero** you need to define a new two-parameter overload of **dump()**. You may see this as a cost; I see it as a huge benefit: implicit conversion being far less worth than

it is effort. Naturally, this also applies to the **filter_type** conversion shim overloads. It's hardly onerous though, since if you don't need any new formatting you can just use a forwarding function, as in Listing 11.

One last point I'd like to make: the type-filter mechanism takes effect *before* any application of string access shims, so you can use a type-filter to override an existing conversion of a type (implemented using string access shims) that you don't happen to care for.

String access shims

The type-filter mechanism defines conversions that are usable only with FastFormat. Now, in most cases this is a positive thing. However, you may also be using other STLSoft-related libraries – I'm mainly thinking of a superlative logging API library here ☺ – and wish to share your types' to-string conversions between them all. If so, you may instead define string access shims for your types. Let's do that now for our **superhero** type.

To do so, you must understand the rules for access shims. Unfortunately, there's not the space here to explain all the rules for shims; for that you'll have to consult *Imperfect C++* [IC++] and/or *Extended STL, volume 1* [XSTLv1]. (The most comprehensive and definitive explanation will be found in my next book, *Breaking Up The Monolith*, but since it's not yet finished, it's not much good to you.) Instead, I will show you how to do it, and point out the major issues as we go.

The *string access shims* are actually three sets of four shims. For simplicity we will consider only the ones that are used with multibyte strings: **stlsoft::c_str_ptr_a**, **stlsoft::c_str_ptr_null_a**, **stlsoft::c_str_data_a**, **stlsoft::c_str_len_a**. Further simplifying, we need only consider the pair of shims **stlsoft::c_str_data_a** and **stlsoft::c_str_len_a** for our extension of FastFormat. Analogous versions exist of all four exist for wide strings, with the **_w** suffix, and you'll need to define **stlsoft::c_str_data_w** and **stlsoft::c_str_len_w** for your extension if you wish to use FastFormat in wide string guise. (The use of the **_a** suffix for multibyte, rather than **_m**, is just historical, but unfortunately we're stuck with it.)

All shims have *name*, *intent*, *category* and *ostensible return type*. A shim is allowed to return any type that is implicitly convertible to the ostensible return type. For our two shims these are (**stlsoft::c_str_data_a**; obtain a pointer to the string representation of the given type; *Access*; **char const***) and (**stlsoft::c_str_len_a**; obtain the length of the string representation of the given type; *Access*; **size_t**). The degenerate forms of each are given in Listing 12.

It is a requirement that, for any matched pair, **c_str_len_a()** always yields exactly the number of characters available at the pointer returned by **c_str_data_a()**. Definitions for **std::string** are equally simple, as shown in Listing 13.

```
// in namespace fastformat:: filters

inline std::string filter_type(
  superhero const& hero
, superhero const*
, char const volatile*
);

inline std::string filter_type(
  extrasuperhero const& hero
, extrasuperhero const*
, char const volatile* p)
{
  superhero const& regular_hero = hero;
  return filter_type(regular_hero,
     &regular_hero, p);
)
```

Listing 11

The complexity comes when dealing with types that are not strings, and do not already contain a viable string form representing their state. Our superhero type is one such. In this case, we must synthesise the string on the fly, as shown in Listing 14.

Once again, for convenience we've used FastFormat to implement the conversion to string. If we were planning to use this string access shim in a context without FastFormat we'd have to resort to **sprintf()** or plain string concatenation. (But not IOStreams or Boost.Format, eh?!). Note that this would not detract from FastFormat's robustness claims, because the correctness of a custom conversion component such as these string access shim overloads can be assessed and verified independently of FastFormat

```
// in namespace stlsoft
inline char const* c_str_data_a(
  char const* s
)
{
  return s;
}
inline size_t c_str_len_a(
  char const* s
)
{
  return (NULL != s) ? ::strlen(s) : 0;
}
```

Listing 12

```
// in namespace stlsoft
inline char const* c_str_data_a(
  std::string const& s
)
{
  return s.data();
}
inline size_t c_str_len_a(
  std::string const& s
)
{
  return s.size();
}
```

Listing 13

```
inline stlsoft::basic_shim_string<char>
c_str_data_a(superhero const& hero)
{
  stlsoft::basic_shim_string<char>result;
  ff::fmt(result, "{0} {{weight={1}, strength={2}
        , goodness={3}}"
        , hero.name, hero.weight
        , hero.strength, hero.goodness);
  return result;
}
inline size_t c_str_len_a(
  superhero const& hero
)
{
  size_t n = hero.name.size() + 32;
  char buff[21];
  // NOTE: not checking -ve return value!
  n += sprintf(buff, "%d", hero.weight);
  n += sprintf(buff, "%d", hero.strength);
  n += sprintf(buff, "%d", hero.goodness);
  return n;
}
```

Listing 14

Erratum

The feature comparison table from part 1 [FF1] had a few defects in it. (This probably resulted from a manual preparation of its strings, rather than concatenating them robustly. Ho hum!) It was entirely my fault, and no reflection on the superb skills and dedication of the Overload staff. The correct version is available at http://www.fastformat.org/errata/overload/introduction-to-fastformat-part-1/table4.html.

(i.e. in a (practically) exhaustive test harness). For the length, we've added the length of the superhero format (minus the sizes of the insertions and the `{` escape character) plus the length of the name, plus the lengths of the string forms of the three integers. This is the common methodology of the string access shim pairs when synthesising string forms: the **`c_str_data[_a|_w]()`** overload creates the string form, and the corresponding **`c_str_len[_a|_w]()`** overload calculates its exact length.

You're probably looking at the definitions with three questions:

- What is a shim string?
- What happens when it goes out of scope?
- Does that **`sprintf()`** stuff in the second function look a bit dodgy?

A shim string is a specialisation of **`stlsoft::basic_shim_string`**, which is a component specifically designed to act as the intermediary return value for string access shims. It has two important characteristics:

- It uses an **`stlsoft::auto_buffer`** internally, such that many cases of conversion can be performed without a heap allocation
- It provides an implicit conversion operator to **`char const*`** (or **`wchar_t const*`**, when specialised with **`wchar_t`**), which means that it fulfils the requirements of the shim's ostensible return type.

The second question touches on an important part of shim lore. Because conversion shims return instances of types by value, it is important that they are either copied or are used within the expression in which the shim is invoked. If not, crashes will ensue.

Access shims are a composite of attribute and conversion shims. Where you would use an attribute shim to access the string form of a **`std::string`**, because it is already in string form, you must use a conversion shim to access the string form of, say, **`struct tm`**. This composite nature means that the most restrictive rules from each of the primary shim categories must apply. In the case of access shims, you must observe rule on the use of return values of conversion shims [XSTLv1]:

> The return value from an access shim must not be used outside the lifetime of the expression in which the shim is invoked.

Thankfully, FastFormat, Pantheios and the other libraries and programs (I know of) that make use of strings observe this rule, and all is well. A temporary is returned, its value used, and then it is destroyed, all in the right order. If you're feeling adventurous, check out the `fastformat/internal/generated/helper_functions.hpp` file in the distribution to see how this is done.

The answer to the third question is: it depends. If I were writing for a non-localised context, I would use another STLSoft component, the **`integer_to_string()`** function suite [I2S] [IC++], to effect the length calculation, as they're quicker than **`sprintf()`**, and do not have a (potential) failure return to worry us. However, all that would be moot if I were writing for a localised context, since I would not consider the extra cycles saved in making manual calculations worth the risk of getting them wrong. Instead, I would take advantage of the fact that **`stlsoft::basic_shim_string`** also has an implicit conversion operator to **`size_t`** and implement **`c_str_len_a()`** as:

```
inline size_t c_str_len_a(
  superhero const& hero
)
{
  return c_str_data_a(hero);
}
```

This looks strange, and indeed it is. It's very rarely good design for a class to have one implicit conversion operator [EC++] [GC++] [IC++], never mind two! But for this special-purpose class, it is not only proper, it is also very useful: the function pair is guaranteed to be correct (in terms of the number of characters available), and because there's likely to be no memory allocation anyway, the performance impact of doing the conversion twice is not likely to be that big. Nonetheless, it is not zero, and so this length-safe double conversion is the exceptional way of doing shims, not the norm.

As is always the case, you can have increased performance as long as you're prepared to wear the attendant increase in effort and/or risk. FastFormat allows you to be master of your own domain.

Summary

This article has discussed customising FastFormat in terms of adding new sinks, and of adding explicit and implicit support for user-defined types. In doing so, it has shone light on several aspects of the design and implementation that support the library's superior robustness, flexibility and performance, including introducing the pedantic pointer idiom.

The next and final part of the series will look at advanced functional usages and performance customisations, how FastFormat co-exists and cooperates with other libraries (both open-source and commercial), and sees some examples from its use in real-world projects.

As before, requests, comments, abuse, and offers of help are all welcome, via the project website on SourceForge: http://sourceforge.net/projects/fastformat. ■

References

[EC++] *Effective C++*, 3rd Edition, Scott Meyers, Addison-Wesley, 2005

[EVAB] 'Efficient Variable Automatic Buffers', Matthew Wilson, *C/C++ User's Journal*, December 2003

[FF1] 'An Introduction to FastFormat, part 1: The State of the Art', Matthew Wilson, *Overload* #89, February 2009; http://accu.org/index.php/journals/1539

[GC++] *C++ Gotchas*, Steve Dewhurst, Addison-Wesley, 2002

[IC++] *Imperfect C++*, Matthew Wilson, Addison-Wesley 2004; http://www.imperfectcplusplus.com/

[I2S] 'Efficient Integer To String Conversions', Matthew Wilson, *C/C++ User's Journal*, December 2002; http://www.ddj.com/cpp/184401596

[PAN] 'The Pantheios Logging API Library', http://www.pantheios.org/; to see why it's the best choice in C++ logging APIs, check out http://www.pantheios.org/performance.html#sweet-spot, which shows graphically how Pantheios can be up to two-orders of magnitude faster than the rest.

[STLSOFT] http://www.stlsoft.org/

[XSTLv1] *Extended STL*, volume 1, Matthew Wilson, Addison-Wesley 2007; http://www.extendedstl.com/

WRESTLE: Aggressive and Unprincipled Agile Development in the Small

Agile development is all the rage. Teedy Deigh introduces a popular variant.

The world of agile development can be an exciting and varied place, especially for a consultant. It can, however, also be a dull place prone to stasis, groupthink and no small amount of tree hugging. It is time to rediscover some of the hidden core values.

To assist in this brief journey of rediscovery, we can also break away from the hegemony of Scrum and XP by examining a new process for very small teams: WRESTLE. Like the common miswriting of Scrum as SCRUM, WRESTLE is not an acronym. However, it looks that little bit more impressive and slightly more technical for having the suggestion that it might actually stand for something. As we shall see, WRESTLE doesn't stand for very much, whether in terms of spelling, principles or patience. Indeed, it has an impressively low tolerance for anything.

WRESTLE fills an important and, of late, neglected niche in the world of software development process: the very small team. In the past, the soft spot for agile development processes was considered to be small to medium-sized teams, ideally collocated, and the question used to be whether it could scale beyond this. These days it seems that agilistas spend much of their time focusing on large-scale distributed development – so much so that some have been prompted to ask whether or not agile development is relevant to systems that are of modest size and teams that are not geographically scattered! The overlooked scenario in all these cases is the very small team of two developers. What agile processes are relevant to these teams? This is the question that WRESTLE struggles to answer.

WRESTLE's default team size is based on the fundamental unit of development, the pair, although it can be scaled up to four by using two pairs. A full house and beyond is seen to be the preserve of other less resource-constrained agile processes. But rather than treating a pair as a co-operative unit, WRESTLE believes that you get the best from developers by establishing rivalry and heightened tension. To this end, essential techniques such as mocking find a new and alternative expression in WRESTLE. Confrontation is normally carried out in a daily stand-off, but can also occur at other times, such as by the coffee machine, at the water cooler or in front of the taken-to-task board.

Embedded within WRESTLE is a deep respect for values of simplicity in design. For example, WRESTLE borrows and distils the central theme of 'patterns are an aggressive disregard of originality', simplifying it to just 'aggressive disregard'. The notion of merciless refactoring is generalised, so that broader decisions beyond basic refactoring are also taken without mercy or consideration. To get things started, developers are encouraged to do the simplest thing that could possibly irk.

Where some development processes are said to focus on exposing organisational dysfunction, WRESTLE is a little more resilient and seeks to distract from organisational dysfunction by providing a focus for the natural pent up dissatisfaction often felt in such environments (as well catering as for the general cynicism common among development types, regardless of the surrounding organisational mood and temperament). The traditional model of a developer as a solitary individual is also respected, with plenty of time given over to introspectives, email and code tweaking. Testing is most definitely left to other people and testers have no place on a WRESTLE team.

The heavy but dull focus on business value and the customer that seems to weigh down many other so-called agile processes is not present in WRESTLE. Instead, WRESTLE is truly agile because it liberates the developer from such constraints. Developers are free to tell the customer stories about what they may or may not have developed, and customers' schedules are pleasantly unburdened from frequent demos and discussions of requirements, so they are free to pursue other activities and don't have to hang around on-site.

WRESTLE is a young process, so some of the terminology has yet to settle. For example, timeboxed developed is said to be carried out in rounds, bouts or matches, depending on personal disposition and organisational culture. There is also a question of whether it would help to have some kind of dedicated organisation, such as a federation, to standardise terminology and to certify RingMasters, the official designation for project managers in WRESTLE. That said, the question has also been raised as to how much standardisation is needed of a process, especially given Parnas and Clement's insight: 'A rational design process: How and why to fake it'. It is indeed likely that most applications of WRESTLE will be staged for the benefit of others. If you know how it's fixed, it's a process you can bet on with confidence. ■

Teedy Deigh developed WRESTLE in response to what she felt was an obvious omission in the rainbow of agile development processes, and has found it to be a useful vehicle for justifying and sounding off about strongly held opinions on software practice to a new and more receptive audience, ranging from newbie developers to unreconstructed hackers, from to agilist enthusiasts to old-school management.