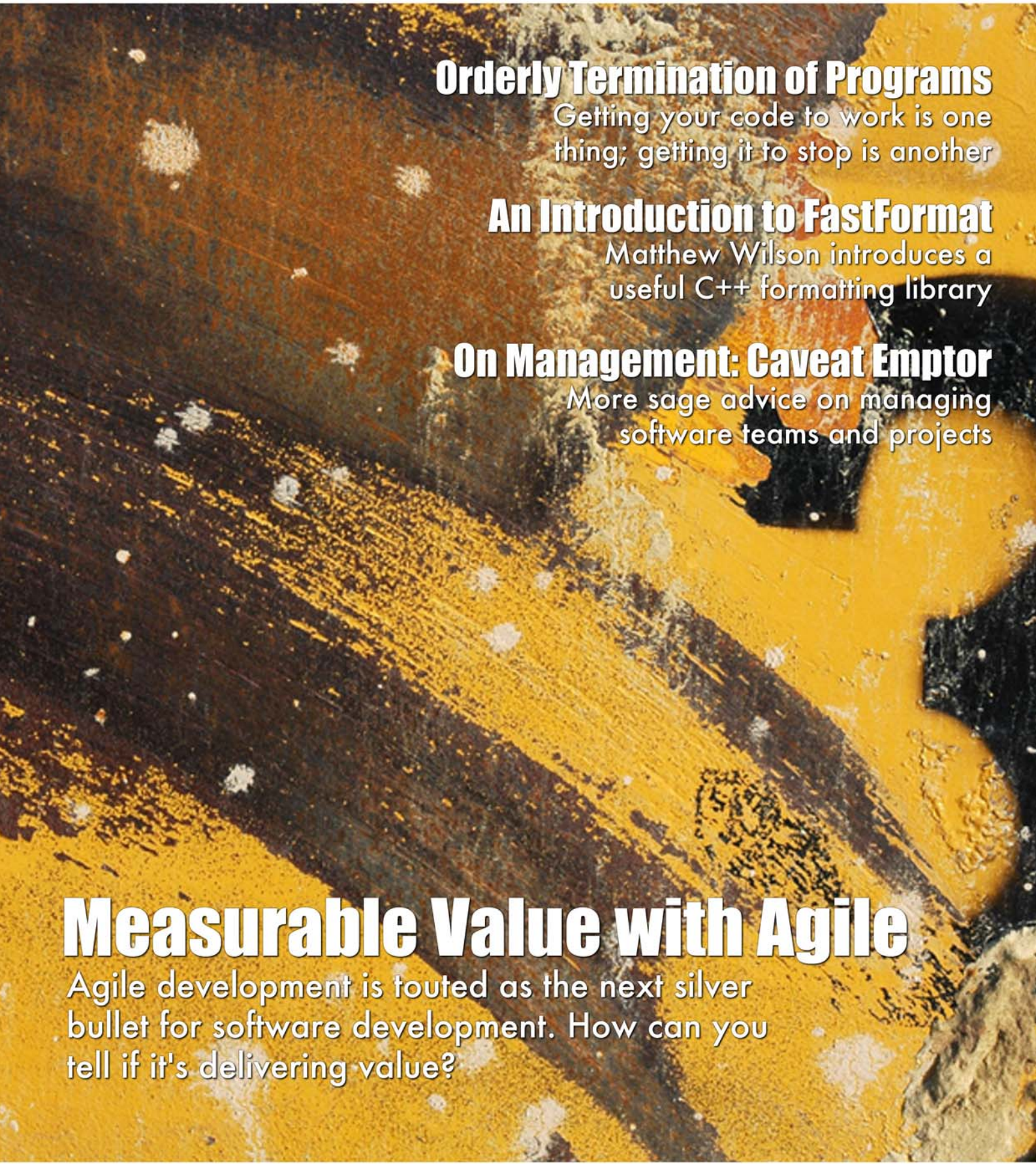
overload 89

FEBRUARY 2008 £3



Orderly Termination of Programs

Getting your code to work is one thing; getting it to stop is another

An Introduction to FastFormat

Matthew Wilson introduces a useful C++ formatting library

On Management: Caveat Emptor

More sage advice on managing software teams and projects

Measurable Value with Agile

Agile development is touted as the next silver bullet for software development. How can you tell if it's delivering value?

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 89

February 2009

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 90 should be submitted by 1st March 2009 and for Overload 91 by 1st May 2009.

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

ACCU

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

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

Watt's going on?

Just how much power are you using...

Over the summer, I'd bought a few new electronic devices and so reorganised all the power cables to be neater, and allow me simple control over which were on. To do this properly I got a simple power meter which allowed me to make decisions informed by the actual power usage, and a few power blocks with individual switches. Didn't take very long and I now have an easy and accessible ability to choose which things are on and drawing power.

I was reminded of this during a recent thread on accu-general which touched on whether it was pointless or not to switch off a TV at the wall. Time to revisit my assumptions, and do some extra measuring at work – after all in an IT business, there are an awful lot of electronic devices on all day (and many people leave them on all night and weekend too.) If we can save significant amounts of energy, we can save money, and perhaps reduce CO_2 emissions too.

So my power meter got dusted off and I went around getting real figures, shown in the following two tables. The measurements are how much a device draws when you turn it 'off' but it is still plugged into the power supply, and how much it uses when being used (which is often a range for devices such as PCs that do sophisticated power management.)

Device	Off (W)	In Use (W)
Amp	0	12
CD	0	9
Radio	0	8.5
TV (also used as PC monitor)	0.5	85
DVD	0	20
Power block	N/A	0.3
Broadband Cable Modem	N/A	8.5
Wireless Router	N/A	4.5
PC	4.5/8.8*	85...160
PS3	1.6	100...113
Wii	1.7	16.5

Table 1: Home

* for some reason my PC uses a smaller amount when it first gets power than after it has been shut down.

Device	Off (W)	In Use (W)
PC	5/75**	130...150
Monitor	0.4-1.5	30...45
Docking station	2.4	5.4
Phone Charger	0.7	3.2
Laptop	0.7	30...50
Laptop Transformer	N/A	0.3
USB Hub	N/A	3.8
Total system (4 monitors)	~10	250...300

Table 2: Work

** first figure is for shutting off the PC, the second is for putting it into Standby.

Some things jump out here immediately. Many devices have very good 'off' power consumption – even zero from old audio equipment and the DVD player – but most still draw a small amount, I suspect from the power transformers – eg a laptop power transformer draws 0.3W even without the laptop plugged in. The PC was an exception, which I suspect is a combination of the PSU and the network card (there's a function to wake up a PC remotely via the network which would need this to be available), but even that is only about the same as a low-power light bulb. I don't have a TV set-top box, but I understand that at some models were really bad – in 'standby' most of the electronics stay on to download updates.

With such figures you can compare various scenarios, such as for my work rig: leave PC on permanently (~250-300W); leave PC on but switch the monitors off, eg via screen savers (~150W); switch everything off but leave everything plugged in (~10W); and switch off at the wall (0W).

This shows that automatic power management can really help reduce power consumption – it will power off monitors and spin down hard drives when you go home or are in a meeting, saving a lot for no effort beyond the initial setup. Getting into the habit of manually switching off devices that are not in use can save even more, but takes some ongoing effort (although there are some power-blocks that use a drop in power use from one socket to switch off everything automatically which makes this even easier); and the final switch off at the wall saves a little bit more, roughly equivalent of a low energy light bulb, but can also be made easier by an accessible power block with a master switch – I've one that's designed to sit on a desk and act as a power block and network cabling.

This last explains why so many people don't bother switching off at the wall – it's more effort to do so, and doesn't save much. Some argue that it's not worth it, especially when compared to the fluctuations in normal usage. Of course those fluctuations happen anyway, and this saves an

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.

extra, independent, amount. And even if it looks small for the individual, when you add it up around a whole office, town and country, this small saving accumulates and becomes significant.

But this is just looking at the small, personal stuff. Are there more systematic changes to our IT systems to make a big difference?

The first is for us to not buy or use a device we don't really need. An interesting blog recently on how start-ups can save money [Startup] suggests not buying everyone a phone, and not to bother with an in-house email server.

The next is for manufacturers to reduce the energy needed to create and run devices. This takes time and we can only indirectly influence this as consumers, but sometimes we can make a difference as developers – I've spent time making sure software correctly puts hardware into a low power state. It might only save a small amount per device, but with a large customer base such actions have a significant total effect.

Suitably setting up a computer's power management settings can have a significant ongoing effect – powering off screens, hard disks, and eventually putting the PC into hibernate will trim unnecessary costs, and can be significant when they are added up over the whole office. A good IT department can influence things here.

Another possibility is to not use complex, power hungry, computers directly – with fast networks, virtualisation, and server farms we can do many tasks using only small network appliances sharing processing power. This still uses power-hungry computers, but they are much less likely to be idle, and they are now centrally managed and so power usage can be controlled more easily. And as power is a major cost, there is a strong incentive for the management to actively control it. A neat example of this: one company is going to build a server farm in Inverness, citing the lower ambient temperature as reducing the amount of cooling needed [Climate] (and it hopes to use the waste heat to provide heating and hot water to local buildings). You can take this further, and site them in places like Iceland – not only is it even colder, but a lot of the electricity is from geothermal sources, and so is cheap and practically carbon-free [Iceland].

Of course some of these savings are minor for the individuals involved, and if you only have limited time and resources you can often get a better return by avoiding waste from the real energy hogs: heat. So don't overfill the kettle, turn the central heating down a degree, keep the jumper on, and top up the insulation...

Core blimey

A few years ago, Herb Sutter published an article 'The Free Lunch is Over' [FreeLunch], and gave a talk version of it at the ACCU conference. This discussed how the increase in processor speeds had levelled off, and the extra transistors predicted by Moore's law were instead being used to provide more on-chip memory, and extra cores. The message was, to take advantage of this we will have to change our programming models from single-threaded to multi-threaded and multi-processor, and our programs can continue to get faster as before.

Or not. Recently some studies were published about how multicores will scale [Memory]. The answer was: not very. Well, almost – if you looked a bit more carefully, what it actually found was that sharing memory across cores didn't scale. This is because accessing main memory is a major bottleneck – memory speed hasn't improved by much compared to processors, and it takes time to get the data to and from the cores. Local caches help, but it adds the problem of keeping their view of memory consistent. And this was in fact mentioned in Herb's article (and has been known for years before that) but here was some more evidence that we really do have to rethink how our programs are designed – things won't improve by themselves. In many ways programming on multi-cores (and multi-processors) is just distributed computing on a smaller scale. There are (relatively) high call latencies, slow data access, and data consistency issues. There is a large literature on how to program with these constraints, but a lot of the ideas and techniques are very different from how many people currently program and it will take time to convert people's thinking – from asynchronous message passing, changing data processing to use algorithms that can be processed in parallel chunks, perhaps using to functional programming ideas and languages.

And a random thought: could OS writers and chip designers start organising things to reflect such a model? For example a chip could be designed to have several groups of cores with their own dedicated local memory. Then the OS could run on a dedicated core-group, and launches processes running on their own core-group. Thus each process' threads would be able to use its own memory without contention from other processes, and the only syncing would be to send messages to other processes which would be coordinated by the OS. Original? Unlikely – it wouldn't surprise me in the least if this hadn't already been done years ago, and the mainstream is only now catching up.

ACCU Conference

And finally a quick reminder – this year's conference lineup has been announced and booking has been open for just over a month. Early bird rates are still available until 28th Februrary, so if you haven't already booked, now is the time. I look forward to seeing you there!

References

[Climate] http://www.itpro.co.uk/608874/data-centre-heads-north-for-cooler-climate)

[FreeLunch] Free Lunch is Over: http://www.gotw.ca/publications/concurrency-ddj.htm

[Iceland] http://www.theregister.co.uk/2007/04/10/iceland_to_power_server_farms/

[Memory] Limits of multicore sharing memory http://www.sandia.gov/news/resources/releases/2009/multicore.html

[Startup] http://calacanis.com/2008/03/07/how-to-save-money-running-a-startup-17-really-good-tips/

Measurable Value with Agile

Are you solving the right problem or simply solving the problem right? Ryan Shriver shows us that both are needed for value delivery.

Agile is one of the hottest trends in IT. It's the shiny new toy that's gone from underground movement to mass marketed panacea. It's done all this within the last ten years or so. Now agile is maturing and being marketed as 'delivering business value', but there's little agreement in the agile community on what this is and how to measure it.

This past summer I spoke at the Agile 2008 conference in Toronto where there were over 1,500 attendees and 400 sessions on everything you could possibly want to know about agile. There was an entire series of presentations on 'Customers & Business Value', yet amongst the presentations, none that I saw covered:

- What is value? What do we mean when we say agile delivers business value?
- How do you measure value?
- What do you measure it with (and when)?

After the conference I thought, 'For a community where nearly everyone talks about delivering business value and prioritizing by business value, I've seen very few specifics on how to implement this in practice'. Yes, organizations see features getting implemented and they track velocity, but they've got no real way to measure the value delivered by these features.

The challenge – are we delivering the right things, now?

In my experience, most IT project teams, even agile ones, rarely grasp the business objectives of their stakeholders investing in the project. Project leaders either don't understand, can't articulate or don't care what drives business value or how it aligns with business strategy. It's sad to witness the flurry of new project activities while nearly everyone fails to distinguish between:

Delivering the right things and delivering things right

This is especially acute in the agile community and it's setting a dangerous precedent. As methods like Scrum increase in popularity [VersionOne09], an overall focus on real value and delivering the right things becomes even more critical. Today, in practice, teams can be performing Scrum flawlessly (delivering things right) only to find out they were doing the wrong project all along (delivering the wrong things) because they didn't understand the real business objectives. The result is an investment that may result in running software that delivers no business value despite the (apparent) success of the agile process. Whoops!

But it doesn't have to be this way, as this article will demonstrate.

Ryan Shriver is a Managing Consultant with Dominion Digital, a Virginia-based process and technology-consulting firm, where he leads the IT Performance Improvement Solution. With a background in systems architecture and large-scale Agile development, he currently focuses on measurable business value and systems engineering. He writes and speaks on these topics in the US and Europe, posting his current thoughts at theagileengineer.com.

Determining the right thing doesn't have to be costly and complex. In fact, it can be done without changing Scrum and without slowing teams down.

In this article the reader will learn that measurable value using quantified business objectives and Scrum can work together to ensure teams are both focused on the right things and delivering the things right. Used together, teams can move beyond feature-builders to value-delivers, measuring progress not in features built but value delivered using business-defined metrics.

Now is the time for agile teams and the agile community to seek out and embrace practical ways to demonstrate measurable business value. By engaging the business and quantifying their objectives, agile teams can ensure investments are aligned with strategies. The agile community, including you, can help IT transform from feature builders to value deliverers.

Value delivery approach

This article presents value delivery, a practical approach for measuring the stakeholder value delivered by teams. This approach directly aligns business strategies and stakeholder objectives using simple quantitative methods for clarity.

To understand this approach, we must first get to the root of the issue with defining value. It is, not surprisingly, communication. In most organizations a communication gap exists between the lofty prose used by leadership to describe strategic initiatives and the planning prose required by teams to deliver business value. Value delivery bridges this communication gap by transforming vague language into clear objectives that can be planned and measured. Teams that help stakeholders get closer to achieving these objectives are delivering tangible value to the business.

Value delivery advocates measuring value using quantified business objectives in alignment with business strategies. It does not advocate measuring value using features, functions, function points, epics, user stories or tasks. These are practically all too low-level for measuring value to be worth the investment.

Rather, to deliver value, people, process and technology must be properly blended so that stakeholders' objectives are met. Value delivery advocates a systems-thinking approach that encourages teams to think holistically about the problem space using numbers to assess the impact of designs on objectives.

Value delivery is a combination of existing principles and practices from the Evo [Gilb05] method that can be used in conjunction with the Scrum method. It is not the only method to measure value, but I believe it is a method that works well with the Scrum method. I believe value delivery can help agile teams show measurable value delivered in alignment with organizational strategies quickly and effectively.

Today's engagement

To show value delivery in action, we'll use a slightly altered real-world case study. You and I are going to consult with a non-profit client and we're

Now that we've identified our stakeholders and resources (time and money), that just leaves defining the objectives

going to help them adapt the value delivery approach to their existing Scrum process.

Our client, a leading non-profit research organization, recently completed its 2009 strategic planning sessions. Senior management's business strategy is:

> In 2009, we are embarking on a strategy to increase charitable giving through improvements to our web site. We believe this strategy will help increase our market share for online giving while positively impacting our key customers: non-profit organizations.

The organization currently uses Scrum for developing their web-based application. The Vice President of Marketing and business sponsor, Nancy, has personally asked us if we could help her and the organization:

1. Establish a set of strategic objectives so value can be measured and managed
2. Make smart funding decisions with web site improvements so budget and risk can be managed
3. Identify the improvements with the best 'bang for the buck' for doing first so quick progress can be demonstrated to everyone.

We need to ensure our work integrates nicely with the existing Scrum process used for web site development.

At our initial meeting, Nancy asks if we can take this job on. 'No problem' I tell her. Then she shrugs a bit, 'And being a non-profit, we can't pay much at all.' I hesitate for a second, then respond, 'If you can ensure we get access to the right people, and provide us with someone to organize the meetings, we'll do the project pro-bono!'

'Wonderful!' Nancy exclaims. 'If it's ok by you, let's get started the week after next. That'll give me some time to lineup the right stakeholders.'

So with that, you and I are off to do some valuable work for a worthy non-profit. Ready to get started?

Step 1: Identify stakeholders, objectives and resources

When faced with problems like these, I like to ask myself three questions:

- Who are my stakeholders?
- What are their objectives?
- What resources are available?

Since we're looking to make improvements at the organizational level, we're certain board of directors, CEO and executive management are all stakeholders. Their customers: non-profit organizations and for-profit organizations that donate to non-profits, are also stakeholders. Other internal stakeholders include operations, development, marketing and management.

Our first day on-site we conduct interviews with key stakeholders (up to the CEO) and spend quite some time with Nancy. We meet a diverse set of individuals in marketing, sales and IT who provide us background on their roles and how the strategy will likely impact them. We cast a wide net to ensure that we don't leave anyone out.

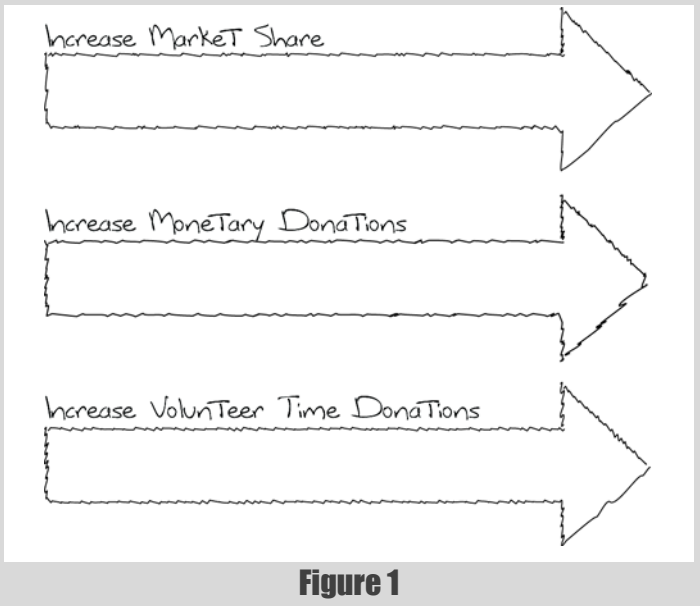


Figure 1

During our interview with Nancy, she says, 'The CEO recently agreed to provide $1 million and 10 months for implementing the business strategy, but wants to see results quickly. My responsibility as business sponsor is to ensure this succeeds, but where do I start?'

After a bit more conversation, Nancy and I sit down together at the table and I continue, 'Now that we've identified our stakeholders and resources (time and money), that just leaves defining the objectives. This is by far the hardest question to answer, so let's take an iterative approach to creating our objectives. The first step is to identify each with a simple name.'

I turn my attention to a copy of the business strategy on the table, pull out my pen and underline the key themes I see:

> In 2009, we are embarking on a strategy to increase charitable giving through improvements to our web site. We believe this strategy will help increase our market share for online giving while positively impacting our key customers: non-profit organizations.

After some further discussion with Nancy, we quickly identify the following objectives and write them on the whiteboard in the room (Figure 1).

The first two come straight from the strategy. Nancy tells us the last is a request from our non-profit customers. In addition to money, non-profits also value the time donated by volunteers to help them fulfill their missions. We decide these three objectives are enough to get started and provide the right focus for the team, so we capture these and move on.

Step 2: Quantify our objectives

With these objectives identified quickly, we're feeling pretty good about our progress. I say to Nancy, 'The next step is making them quantifiable.' Nancy, looking a bit puzzled, says,'Why should our objectives be

It is through the process of trying to quantify objectives that we probe more deeply into what's really important

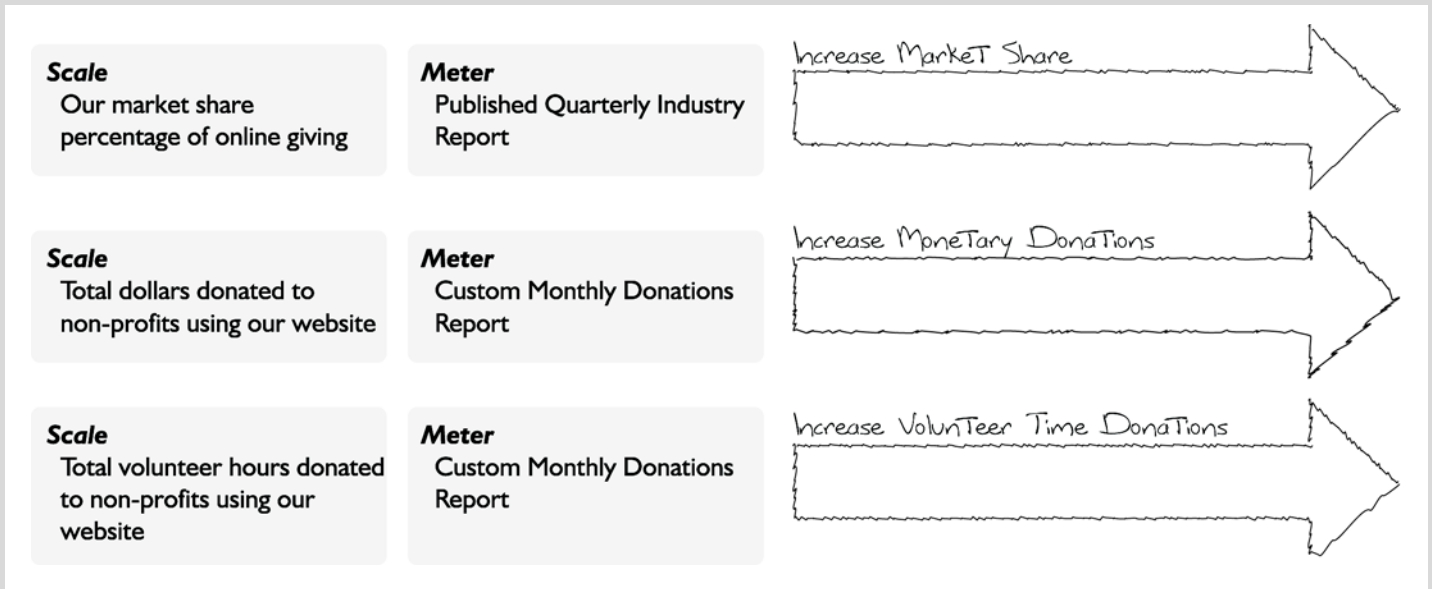


Figure 2

quantified?' I respond that contrary to what she may think, 'The main purpose of quantification isn't to measure and track. The main purpose is to provide clarity in requirements. Tom Gilb [Gilb] says it best:'

> The fact that we can set numeric objectives, and track them, is powerful; but in fact is not the main point. The main purpose of quantification is to force us to think deeply, and debate exactly, what we mean; so that others, later, cannot fail to understand us.

I continue, 'It is through the process of trying to quantify objectives that we probe more deeply into what's really important.'

Nancy responds, 'But how do we do this? Remember, we don't have much time to start showing progress. I'm not looking for an academic exercise, I need results!'

'It's OK', I say. 'There's actually a way we can quantify these objectives pretty quickly using Planguage [Gilb05]. With our objectives identified, we'll next add a Scale (*what to measure*) and then a Meter (*how to measure*).'

Nancy and I return to the board and update our objectives (Figure 2).

Step 3: Identify targets, constraints and benchmarks

Nancy is again happy with the progress but asks, 'Now I see quantifiable objectives, but without knowing where we are today or going in the future, what use is this?' She's right.

'Time to tell you about Targets, Constraints and Benchmarks!' I respond.

Targets, as the name suggests, are the performance levels the team is striving to achieve. It's the level of performance that, when reached, everyone agrees is success. Stakeholders agree to provide the necessary resources to achieve these levels and technologists agree to design systems to meet these levels. Target levels are not simply edicts laid down by stakeholders absolutely. Rather, setting them requires collaboration and agreement from the implementation team to ensure the levels are achievable (with an estimate of what resources it may take to get there).

Constraints are the levels that must be avoided. In practice, these could be contracted Service Level Agreements (SLAs) identifying the minimal performance levels before penalties are assessed. They may also be the minimum levels needed to ship the product.

In our case, these are the levels below which senior management recognizes things didn't go well (and perhaps bonuses would be impacted!). Just like targets, setting constraint levels requires collaboration from all parties.

Finally, benchmarks are the levels achieved today or what's been achieved in the past. Benchmarks enable an understanding of the current state and assessing how close (or far) we are from achieving the target levels of performance. In practice, it's often easiest to start with identifying current benchmark(s) and using this to set appropriate target and constraint levels.

After listening Nancy responds, 'Why do we need constraints? Can't we just set targets?'

I remind her, 'As important as setting levels for success is, it's often more important to set levels for failure. Everyone in the project needs to understand clearly what's success, what's failure, and where the organization is today. With that understanding, we can begin honest discussions about what to do next.'

Nancy takes a guess at target and constraint levels and we do follow-up interviews with developers and testers to gather benchmark data. Pulling

A design idea is a potential solution that moves a team closer to achieving the stakeholder's objectives

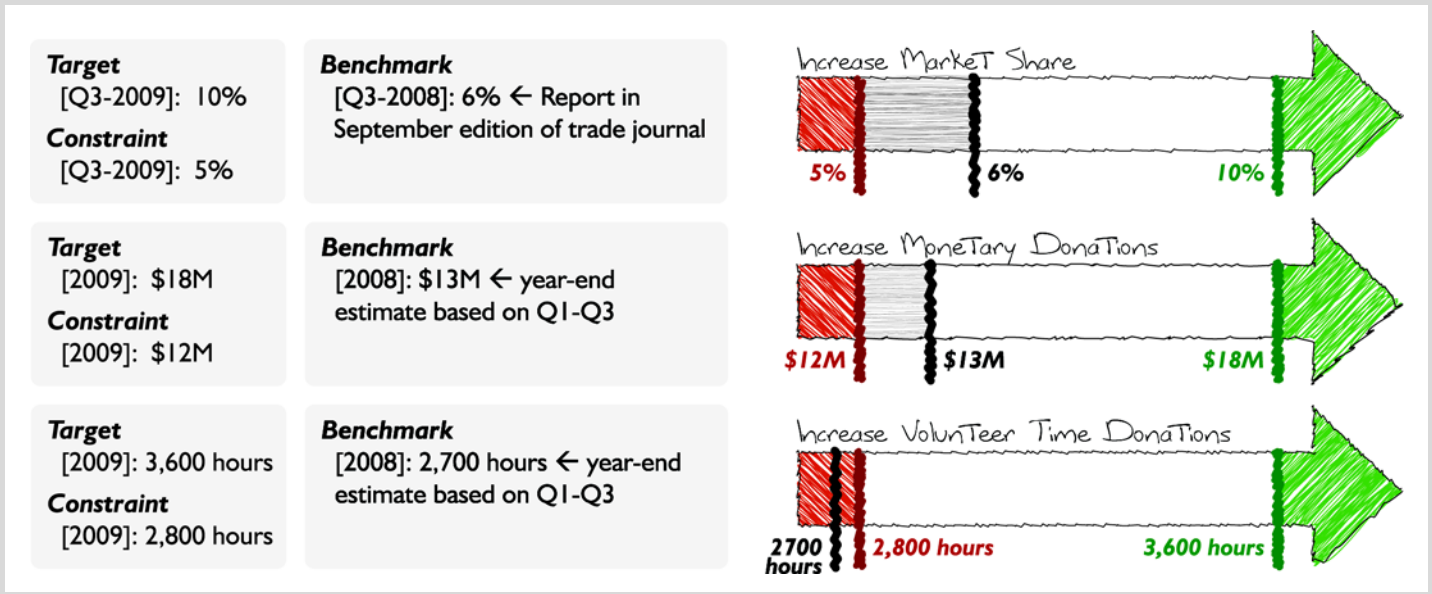


Figure 3

this all together, we update our whiteboard and show Nancy our results the following morning (Figure 3).

As we walk Nancy through this, we also explain the last two Planguage concepts: Qualifiers and Sources.

Qualifiers are the variables in brackets [] and they help specify under what conditions the levels apply. Qualifiers are typically dates, places or events. Examples could include: 2008, Q1-2009, UK only or Release 3. These are user-defined and can be anything that makes sense for a particular situation.

Sources are the text to the right of the arrows ← and they help convey where information originated. Sources can be applied to any piece of information to add transparency, credibility and traceability.

We've introduced Nancy and the organization to a lot of new concepts, so let's briefly recap before moving forward:

- **Scale** – What's measured (units)
- **Meter** – How it's measured (method)
- **Targets** – Levels aiming to achieve
- **Constraints** – Levels trying to avoid
- **Benchmark** – Current or past performance levels
- **Qualifiers** – Dates, places or events useful for clarification
- **Sources** – Origin of information for transparency and credibility

I tell Nancy, 'Think about our progress right now. We have quantifiable objectives for our business strategy that will fit on a single PowerPoint slide and can be communicated and understood by all project team members and stakeholders, including the CEO!'

She responds, 'Wow, *now that's powerful*! I think we're ready to share this with the other senior managers and stakeholders, I'll set up a working session for this Friday so we can get validation before moving forward.'

Step 4: Brainstorm design ideas

During Friday's working session the business sponsor, product owners, leaders, analysts, developers, testers and ScrumMaster are all in attendance. Agreement is reached on the target and constraint levels we previously established with Nancy. With the team itching to start designing solutions, I explain 'Next comes the really fun part: creative brainstorming of design ideas.'

'What's a design idea?', Steve, one of the architects asks. I respond, 'A design idea is a potential solution that moves a team closer to achieving the stakeholder's objectives.'

In order to ensure the brainstorming is productive, I tell the team, 'The objectives have been established and validated, so now let's find the solutions. But I'd request that each of your focus the brainstorming session on finding design ideas that will:

1. Increase Market Share
2. Increase Monetary Donations
3. Increase Volunteer Time Donations

Good design ideas will positively impact one objective, but **great design ideas** will positively impact all three objectives with a single design idea!'

The team knows their current web site is pretty basic with no fancy Web 2.0 stuff. There's a basic search and the ability to view reports of non-profits. Users can also make donations directly to non-profits on the web site by clicking a 'Donate Now' button.

The process of identifying design ideas is a creative one that encourages out-of-the-box thinking engaging the entire team

Nancy offers up to the team, 'The CEO is looking for ideas that would kick off an implementation project and are achievable within a $1 million budget and 10 months. Those are the constraints of our brainstorming today.'

The process of identifying design ideas is a creative one that encourages out-of-the-box thinking engaging the entire team, not just executives and product managers. In practice, good ideas often build off one another. Teams that do this before starting their projects achieve a greater shared vision of what they're being asked to deliver and understand the real measures of success that translate to value to the organization.

During the session lots of ideas are generated and captured. Some of the more interesting ones include:

1. Setup recurring payments for members so each month a donation is automatically made to the non-profits of their choice.
2. Create a Facebook application that integrates the web site to the 100+ million Facebook users so they can get connected to non-profits organizations of their interest.
3. Create ability for Non-Profits to upload images and videos of their charitable work to the site to help solicit donations.

Nancy says, 'Based upon the team's intuition, these design ideas sound like the best candidates. Let's move forward.'

She turns to us and says, 'Now, how do we get down to just one that we can do?'

Step 5: Select the next best design

'Arm wrestling works well' I quip.

'Get serious', responds Nancy.

'OK, we could vote on each option using a secret ballot. Or we could just let the CEO pick. There are more options, but I think they would be irresponsible, given the time and energy we just spent creating measurable objectives.'

I continue to explain to Nancy and the team, 'We should use Impact Estimation (IE) to help us with this problem. IE is a very simple tool for calculating cost/benefit of design ideas and identifying the one with the best return on investment or *bang for the buck*'.

To help Nancy and the team understand the concept, I draw Figure 4 on the board to show the structure of an IE table with objectives and resources as the rows and design ideas as the columns. The last row is the benefit to cost ratio (value delivered) which will help determine the best design idea. This is a simple ratio of the sum impact of objectives divided by the sum impact on resources (i.e. sum objectives impact / sum resources impact).

After a little explanation, it appears the team gets the gist of IE enough so we can get started.

It's been a very productive session but I feel the energy slipping on Friday afternoon. 'Let's break for the week and next week we'll start to fill in the impact estimation table with the help of the team', I say. Everyone agrees and heads back to their desks and home for the weekend.

	Design Idea #1	Design Idea #2	Design Idea #3	Total Impacts
Objectives	Impact on Objective	Impact on Objective	Impact on Objective	Total Impact on Objective
Resources	Impact on Budget	Impact on Budget	Impact on Budget	Total Impact on Budget
Benefits to Cost Ratio	Ratio	Ratio	Ratio	

Figure 4

Nancy approaches us afterwards, 'I really want to thank you, I have never seen our team identify so many good ideas so quickly. They were amazing! Because we had focus on our objectives, they really embraced the process and the result was great collaborative energy in the room. I'm anxious to see which design idea we end up doing!'

The following week the team meets again and puts the top three design ideas into the IE table and estimates each ones impact on objectives and budget. I encourage the team not to try to get too precise; instead simply try to make a first pass using the best information at their disposal. To show uncertainty, with each value best and worse case is provided using ± notation. At the end of the session the team has identified the Facebook Integration as the best design idea and circles it in the IE table (Figure 5).

In doing this exercise, I point out the following things to the team members, so they see how we arrived at our decision:

- **Recurring Payments** have the biggest impact on the Increase Monetary Donations objective, but relatively low impacts on the others. Its estimated cost is $200K–$400K (30% ± 10% of monetary budget) and its expected to take 2–6 months to complete (40% ± 20% of time budget).
- **Facebook Integration** has the lowest sum impact on objectives (110%) of any of the ideas, but its low cost ($100K–$300K and 1–4 months) means it has the best benefit / cost ratio of all design ideas. So it's the idea we'll go with first.
- **Image & Video Uploads** has the biggest sum impact on objectives of any of the design ideas, but it also costs the most, resulting in the lowest benefit/cost ratio.

These are the obvious observations, but there are other more subtle things that are also interesting. I present the following to the group for consideration:

If we're wrong, at least **we'll know quickly** and can change direction

	Recurring Payments	Facebook Integration	Image & Video Uploads	Total Impacts
Objectives				
Increase Market Share (6% → 10%)	**30%** ±20%	**30%** ±20%	**20%** ±10%	**80% ± 50%**
Increase Monetary Donations ($13m→$18m)	**80%** ±30%	**30%** ±30%	**50%** ±20%	**160% ± 80%**
Increase Volunteer time Donations (2,700→3,600)	**10%** ±10%	**50%** ±20%	**80%** ±20%	**140% ± 70%**
Total Objective Impact	**120% ± 60%**	**110% ± 70%**	**150% ±50%**	
Resources				
Money ($1.0M)	**30%** ±10%	**20%** ±10%	**50%** ±20%	**100% ± 40%**
Time (10 months)	**40%** ±20%	**20%** ±10%	**50%** ±20%	**110% ± 60%**
Total Budget Impact	**70% ±30%**	**40% ± 20%**	**100% ±30%**	
Benefit to Cost Ratio	**120/70 = 1.7 Worst: 0.6 Best: 4.5**	**110/40 = 2.8 Worst: 0.7 Best: 9**	**150/100 = 1.5 Worst: 0.9 Best: 2.9**	

Figure 5

- Assuming the design ideas are mutually exclusive, implementing all three would not likely help the organization meet their Increase Market Share objective (Total Impact is 80%). If meeting this objective is the most critical, then more design ideas must be identified.
- Similarly, if meeting the Increase Monetary Donations objective is the highest priority, then Recurring Payments must be one of the design ideas implemented because it alone gets us 80% there.
- The Facebook Integration design idea has the most uncertainty associated with it (widest range of benefit/cost ratios: 0.7 to 9). Although this looks like the best design idea now based on its benefit/cost ratio (2.8), more research may be required to bring this uncertainty down or get the stakeholders to consciously accept this level of risk before moving forward.
- Implementing all three design ideas would use 100% of the money budget and 110% of the time budget, meaning the organization would likely not have time to implement all three design ideas in the next 10 months.

Nancy steps into our team session at the end of the day and I explain to her that the team now has:

1. Quantified the impact of the three best design ideas against the objectives and budget
2. Determined the design idea that has the best benefit/cost ratio
3. Identified the risk and uncertainty associated with each design idea, prompting more research to reduce the uncertainty or acknowledging the risk comfort level and moving forward

The team's choice of which improvement to pursue first is now less guesswork and more fact-based using quantified data. The IE table easily explains to all stakeholders why the Facebook Integration is the best project to pursue and can back it up with real numbers including expected benefit and cost compared to the other design ideas.

Nancy agrees, 'This is perfect. I can now present the Facebook Integration project to the stakeholders and the CEO for approval. When they question why this project was chosen, and I know they will, I can show them how we measured the impact of the best design ideas against our objectives and budget.'

Nancy continues, 'I'll set up another working session for this Friday with the stakeholders and the team so we can get feedback and make a decision on the project choose so we can move forward.'

On Friday, Nancy presents the Facebook Integration idea as the first initiative the team will pursue. After some initial questions, the stakeholder's agree and give Nancy and the team the green light to forward.

The CEO explains to Nancy and the team, 'Although I realize we could spend more time reducing uncertainty and getting more refined estimates, I accept the level of risk in order to move forward and start seeing results!'

He continues, 'If this were a bigger project I'd expect more details, but what I've seen here today makes me comfortable the team has assessed our options well. If we're wrong, at least we'll know quickly and can change direction. Best should not get in the way of better. Let's go with the Facebook option and see how quickly we can impact those objectives!'

Later that afternoon the CEO drops by Nancy's office. He asks how things are going and congratulates her on today's working session. He's clearly

we now have a process for quickly identifying design ideas and assessing their impacts

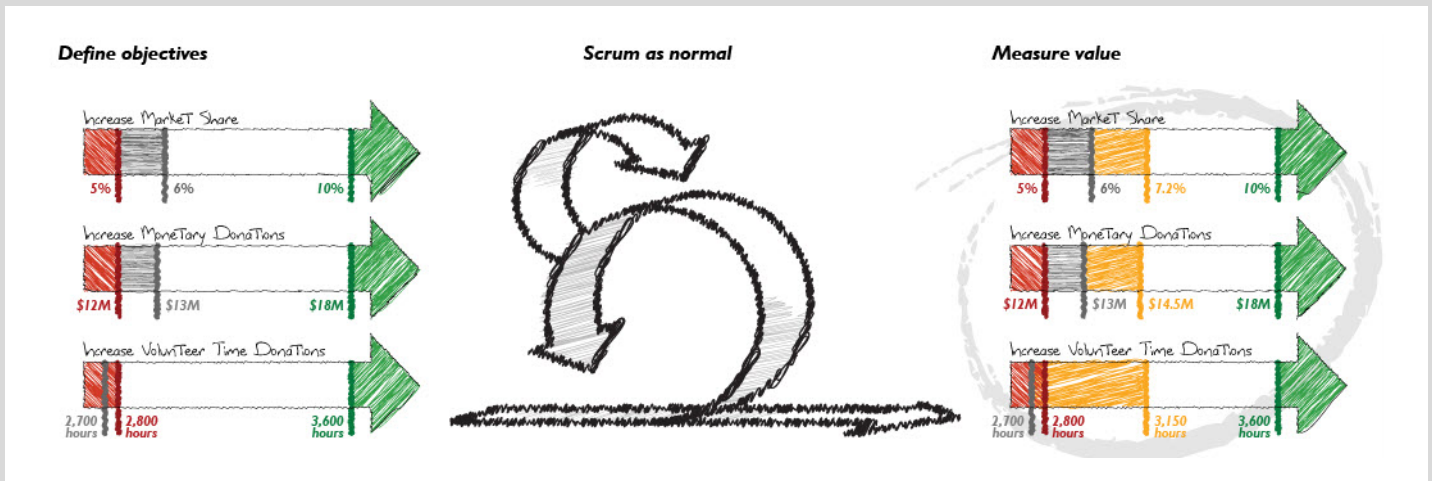


Figure 6

happy with how quickly the strategy and team are progressing and quips, 'you've got a real team of movers and shakers working for you!'

The normally reserved Nancy can't help but beam with pride. She, too, is proud of what her team has accomplished in two short weeks. She recaps for her boss that because of their efforts, the organization has:

1. Established a set of strategic objectives so each sprint the value delivered by teams can be measured and the project properly managed. Not just that, but the entire team is aligned with the objectives, something that's never happened before.
2. Leveraged the Impact Estimation process to ensure smart funding decisions are based on quantified information. This allows budget to be managed and risk to be mitigated.
3. Identified the Facebook Integration as the best 'bang for the buck' and therefore the one that'll be done next. Other design ideas will be evaluated and scheduled for future releases using the Impact Estimation process during planning.
4. Started to update the web site product backlog so the initial Facebook Integration features are rolled out in the next release of the web site in six weeks.

'I tell you what. Take the team out after work today for a celebration. I think they deserve it.' responds the CEO.

Over drinks at the local pub after work, the team celebrates their success and their growing camaraderie. Nancy comes up to me and says, 'After going through this process I'm very confident we're focused on the right problem. Like I told my boss, even if the Facebook option isn't the best choice, we now have a process for quickly identifying design ideas and assessing their impacts. We're focused on the results and if this one doesn't work, we can quickly find the right ideas and make them happen.'

But then she pauses, 'Before we celebrate too much, we still need to ensure we can do this project using our Scrum method.'

'Don't worry', I say. 'Scrum will fit right into this process. In fact, we're not going to change Scrum at all. We're simply going to do a few steps before and after each sprint. It'll have a minimum disruption on the team, you have my assurances.'

'OK, I'll trust you on this.' Nancy responds. Then, after a few seconds, she says, 'But I'm not too worried. Our team is accustomed to delivering projects using Scrum, so long as you can items prioritized and into their backlog, we should be good.'

Step 6: Agile integration

Now that Nancy and her team are focused on right problem, it's time for them to solve the problem right. This is where an agile method like Scrum comes in. Some of the benefits of using agile to implement design ideas include:

1. Short iterations and inspect-and-adapt philosophy get us working solutions quicker and reduce our delivery risk
2. Collaborative nature of cross-functional teams with feedback from stakeholders.
3. Team communication and collaboration creates a positive, supportive work environment where teams feel ownership of the process.

The ways in which agile feels different with the value delivery approach include:

1. Incremental funding is based on real value delivered. If, after a few iterations, teams are not making progress on the prioritized objectives, stakeholder's can call 'stop' and pursue a different design idea with their remaining budget.
2. Entire teams are aligned to the business value of the project and know how success is measured.
3. In each sprint, progress towards objectives is measured to show the value delivered in the sprint, as shown in Figure 6.

Teams can still develop user stories for functional requirements and prioritize them in the backlog as normal

The value delivery approach rewards teams for delivering value, not just building features. It raises the level of measurement up from feature-centric velocity to value-centric objectives. And because these objectives are aligned to key business strategies, we're ensuring we're not just solving the problem right, we're solving the right problem.

The value delivery approach integrates with Scrum's product backlog in three ways:

1. Teams can still develop user stories for functional requirements and prioritize them in the backlog as normal. User stories are useful for documenting 'what' the system will do. User stories should be prioritized by business value, meaning how they help the organization make progress towards their prioritized objectives. User stories use the following format:

   As a <user>, I would like to <do some function>, so that I can <achieve some goal>

2. Teams should also prioritize quality requirements (aka non-functional requirements) alongside functional requirements. These are useful for ensuring critical system qualities such as availability, scalability, usability, response time, etc. can be met. Quality requirements should be defined using the Planguage concepts introduced earlier including Scale, Meter, Target, Constraints and Benchmarks so expectations are clearly stated (with numbers) about how 'well' the system must perform. Because meeting target levels for quality requirements often requires multiple rounds of improvement, quality stories can help ensure:

   As a <stakeholder>, I would like to improve <some quality dimension> from <current level> to <desired level>, so that I can <achieve some goal>

3. Both user stories and qualities stories are specified such that teams can use story point estimates to assess their relative implementation effort. This helps communicate to the product owner what can be done each sprint and how long it will take to get meet specific objectives.

Value delivery uses traditional Scrum activities such as release planning, sprint planning, story point estimates, burn-down charts and stand-up meetings all without modification. In practice this minimizes the change curve and ensures existing processes that are working aren't disrupted.

The difference is that in addition to the normal end-of-sprint activities such as demos and retrospectives, value delivery teams report the value delivered (progress achieved that sprint or release towards stakeholder objectives). While every sprint may not result in measurable business value delivered, making this reporting part of the process ensures everyone stays focused on what's really important and not focused on simply building features. It also helps focus the release planning process on delivering measurable value if product owners and teams know they will be asked to show the value they are delivering with each release.

Value delivery adoption

Just like Nancy and her team, organizations need some guidance with adopting value delivery. Adoption is ideally done from the start of new projects. However, because value delivery doesn't change the mechanics of functioning agile teams, organizations can transition to value delivery on existing projects. The value measuring steps before and after each sprint will hopefully change the team's focus, but shouldn't interrupt their natural team dynamics.

Although this approach is fairly new, I hope it will slowly start to gain practice and will evolve with feedback from the community. I have noticed it does follow an emerging pattern of combining practices from multiple methods to achieve better results. Scrum co-inventor, Jeff Sutherland, in 2008 published the positive effects of combining CMMI + Scrum over implementing pure Scrum alone [Sutherland08]. I believe value delivery is on this same trend, seeking to get better results by complimenting Scrum with established practices from the Evo method.

If you would like to try value delivery on your next project, in addition to this article there are free tools to assist you available on theagileengineer.com. I am actively recruiting individuals who would like support using value delivery on their next project. Please contact me for guidance and coaching on applying these concepts in your organization.

Case studies

The value delivery method described here is essentially selected practices from the Evo method combined with the Scrum method. Both of these methods are well tested and have large numbers of case studies behind them (Evo starting in the 1970s and Scrum starting in the 1990s).

In 2007, I started applying the value delivery approach on projects ranging from custom software development to Microsoft SharePoint implementations. As of 2008, three clients in the United States have used the method and all reported they enjoyed the focus they achieved on business value and reported their teams felt more aligned to stakeholder objectives. However, publishable case studies have yet to be done with the value delivery method, although that remains a goal.

In putting this approach into practice, I observed the following hurdles to adoption:

1. Introducing the concepts of measurable value may be very challenging if an organization is struggling to get the basic Scrum process working and think their issues are with Scrum adoption.
2. When Scrum adoption is a 'bottom-up' approach, developers will often not appreciate the concepts of measurable value delivered (at least not initially).
3. Progressive business leaders or experienced agile coaches who recognize the value this brings will be required to help drive adoption and maintain focus.
4. If you don't have support from business stakeholders to continually measure value, teams won't think it's important to measure their results.

Are we delivering the right things, now?

5. If business stakeholders want to 'place an order with IT' and not be engaged in defining objectives and measuring value, adoption will be practically impossible.
6. Making value delivery stick requires discipline in not only creating the initial objectives, but also following through with actual results and using these in project management. Organizations can start with value delivery, but unless management is committed to asking for the value delivered from their investments, interest can quickly fade.

Others are starting to integrate Evo and Scrum, but as of yet have no published results. Recently, Jens Egil Evensen of Norway has been reporting success using Evo with Scrum in a method he calls 'Avegility' for his customer's projects. Evo is used for project management and to prioritize the backlog, Scrum is used to develop the software and Planguage is leveraged to write the requirements. [Evensen09]

Kai Gilb is teaching and coaching management and Scrum product owners at an international organization how to define value-results, and how to link the business, stakeholder, product and solution value-results to product backlog Items. He reports, 'This process enables them to create a product backlog that is optimized and justified all the way through from product values to stakeholder values all the way up to business values. Everything the Scrum development team does is justified all the way up the value chain, it gives early, frequent, measurable, highly leverage benefits at all levels. Management can manage the value creation, and Scrum developers can deliver it.'[Gilb08]

Summary

To return to our challenge, 'Are we delivering the right things, now?' hopefully you can see there is an approach for answering 'yes' with confidence that's lightweight and agile! Nancy has learned it and so has our team. For our non-profit client, the value delivery approach:

1. Defined what 'delivering value' means within their teams
2. Measures value as progress towards stakeholder objectives
3. Prioritizes design ideas according to ROI
4. Integrates with their existing Scrum method

In summary, value delivery's philosophy is that teams and organizations should use whatever agile method they prefer for delivering things right. But they must also focus on delivering the right things with a laser focus on business value. Only by covering both of these perspectives will organizations ensure their outcomes are results driven and not feature driven. ■

Thanks to Chris Allport, Kai Gilb and Jimmy Chou for early feedback.

References

[Evensen09] Evensen, J. E. (2009) Based on private email conversation

[Gilb] Gilb, Tom, www.gilb.com

[Gilb05] Gilb, T. (2005) *Competitive Engineering*, Oxford: Elsevier Butterworth-Heinemann

[Gilb08] Gilb, K. (2008) Based on private email conversation.

[Sutherland08] Sutherland, J., Jakobsen, C., Johnson, K. (2008) Scrum and CMMI Level 5: The Magic Potion for Code Warriors. Retrieved 14 Jan 2009 from http://jeffsutherland.com/scrum/Sutherland-ScrumCMM16pages.pdf

[VersionOne09] VersionOne, '3rd Annual State of Agile Development Survey', retrieved 11 Jan 2009 from http://www.versinone.com/agilesurvey

EXCEPTIONAL REWARDS, CENTRAL OXFORD

Why does a leading Quant Fund want to recruit leading software engineers?

You're developing software that's highly efficient and never fails. We're using software to generate positive returns in chaotic financial markets. We think we should talk.

Who Are We?

OxFORD ASSET MANAGEMENT is an investment management company situated in the centre of Oxford. Founded in 1996, we're proud of having generated positive returns for our investors each year, including 2008, especially as many assets are managed for pensions, charities and endowments. We blend the intellectual rigour of a leading research group with advanced technical implementation. We like to maintain a low profile and avoid publicity. Nobody comes to work in a suit and we are a sociable company, enjoying an annual ski-trip away together.

What Do We Do?

We use quantitative computer-based models to predict price changes in liquid financial instruments. Our models are based on analyzing as much data as we can gather and we actively trade in markets around the world. As these markets become more efficient, partly because of organizations like ours, we must have new insights and develop improved models in order to remain competitive. Working to understand and profit from these markets provides many interesting mathematical, computational and technical challenges, especially as markets become increasingly electronic and automated. We enjoy tackling difficult problems, and strive to find better solutions.

Who Do We Want?

Although most of us have advanced degrees in mathematics, computer science, physics or econometrics from the world's leading universities and departments, we are just as interested in raw talent and will consider all outstanding graduate applicants. We expect all prospective candidates to work efficiently both in a team environment and individually. We value mental flexibility, innovative thinking and the ability to work in a collaborative atmosphere. No prior experience in the financial industry is necessary. We want to hear from you if you are ambitious and would relish the challenge, opportunity and excellent compensation offered.

Software Engineers

We are seeking outstanding software engineers to develop and maintain system critical software running in a 24-hour production environment. You will be responsible for all aspects of software development on a diverse range of projects, such as automating trading strategies, integrating third party data into our system and the development of data analysis tools. You will have the following:

- A high quality degree in computer science or related discipline
- C++ experience that demonstrates your ability to work efficiently in a fast paced environment
- Extensive knowledge of the development life cycle, from prototyping and coding through to testing, documentation, live deployment and maintenance

Desirable experience includes Linux, scripting, working with large numerical data sets, and large scale systems.

Systems Administrator

We are seeking a creative, hands-on system administrator who loves dealing with computer hardware to build upon our existing facilities and help us grow into a brand new facility. As our operations grow, we need to manage and process increasingly large volumes (terabytes) of data.

You will be responsible for the day to day monitoring and maintenance of the data centre, as well as determining short and long term resource requirements. You will also design and build a hot standby site, setup and monitor third party connectivity, and scale the entire operation to remote presences in the major world financial centres. You will have experience in the following:

- Project planning
- Linux system administration using Kerberos, LDAP, and Apache
- Setup and maintenance of NAS appliances and backup systems
- High availability systems
- Grid and clustered computing

How to apply

Please email or post CV with covering letter to:

Address
Dr Steven Kurlander,
Oxford Asset Management,
13-14 Broad Street,
Oxford OX1 3BP
United Kingdom

Email
accu2009@applytooxam.com

Telephone +44 1865 258 138

Closing Date
Ongoing

Start Date
Spring 09 – ongoing

Benefits
Health Insurance (including family),
Pension Scheme,
Life Insurance.

OxFORD
ASSET MANAGEMENT

Through The Looking Glass

What's happening in the next room?
Stuart Golodetz has to find the door first!

In a previous article [Golodetz08], I talked about BSP trees and some of their uses for 3D games. One such use is in rendering a level: given an arbitrary position for the player camera, a BSP tree can be used to render the polygons in a scene in back-to-front order relative to the camera, with no need for a z-buffer. In practice, however, rendering a scene in this way will engender an unacceptably low frame-rate for all but the smallest levels (it's not a method that scales well).

One solution to this, as I mentioned briefly at the time, is to precalculate which empty leaves of a BSP tree can potentially see each other (recall that an empty leaf is called that because it represents an empty convex subspace of the world). For example, suppose there are five empty leaves (*A*-*E*): see Figure 1. If the player is standing in leaf *A*, and we know that leaf *A* cannot possibly see leaves *B* and *E*, then we need only render the polygons in leaves *A*, *C* and *D*. Needless to say, this makes rendering substantially faster.

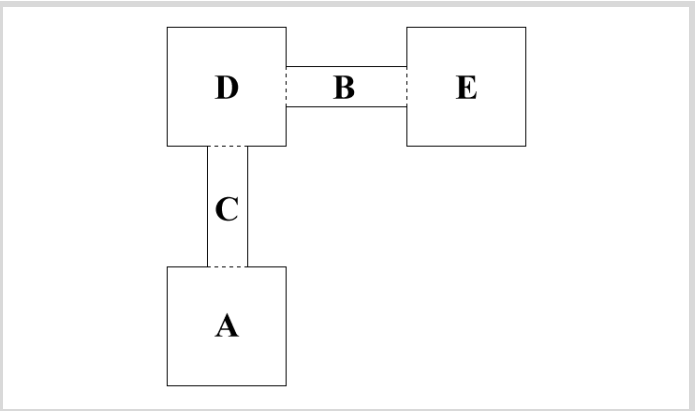


Figure 1

This precalculation process comes in two parts: first, we must determine the portals (or doorways) between adjacent leaves in the level (see the dotted lines in Figure 1). Having obtained these portals, we then calculate the visibility relation between them (i.e. which portals can see each other) in a manner which will be explained later. This portal visibility relation induces a related visibility relation between the leaves, which is what we're ultimately after.

Since the whole process is quite long and involved, I'll explain portal generation in this article, and save calculating the visibility relation for next time.

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

Portal generation

For our purposes, a portal is a polygon which forms a directed link between adjacent (empty) leaves. Its normal will point in the direction of the link, i.e. away from the portal's *from* leaf and towards its *to* leaf. This means that portals are *one-way*, so we actually need two portals for each doorway, one pointing in each direction. For example, we might have a triangular portal from leaf *A* to leaf *B* (whose normal points towards *B*), and another identical triangular portal (with reversed vertex winding order and opposite-facing normal) from leaf *B* to leaf *A*. (I will use counter-clockwise winding order for the purposes of this article, but it doesn't matter which you use as long as you're consistent.)

Generating portals is a three-step process:

1. Determine the set of undirected planes in which the portals could lie.
2. For each undirected plane:
   a) Build a huge *initial portal* (level-spanning polygon) on each plane.
   b) Clip the initial portal to the BSP tree and add surviving portal fragments to the list of level portals.
3. Run through the list of level portals and make a reverse-facing copy of each generated portal.

Determining the undirected plane set

Before we can generate any portals, we need to know the set of (undirected) planes in which they can potentially lie. (By 'undirected', I mean that we consider two otherwise identical planes which face in opposite directions to be the same, e.g. $(1,0,0) \,.\, x - d = 0$ is the same (in undirected plane terms) as $(-1,0,0) \,.\, x + d = 0$.) Intuitively, this seems quite simple: we just add the planes of all the polygons in our level to a set and we're done. In practice, though, we need to be careful to ensure that we don't get duplicate copies of the same plane. This is non-trivial because we're using floating-point values for our plane coefficients: for instance, as far as we're concerned, $1x + 0y - 2z = 5$ and $1.000001x + 0y - 1.99999z = 4.99999$ are basically the same plane, but to the computer they're different. What we need, then, is some way of clustering these very similar planes together.

A solution to this problem is to use a special ordering predicate for our plane set (see Listing 1). This compares two planes, `lhs` and `rhs`, and returns false if they are sufficiently similar. If not, it compares them lexicographically. The idea is that when we're inserting a new plane into a tree (using the normal procedure for set insertion), it can be 'captured'and rejected if it's too near one of the existing planes.

Note that this is similar to the approach taken in [Tampieri92] for grouping nearly coplanar polygons together. I've taken a simpler approach because only one plane from each nearly coplanar group is needed for the purposes of portal generation, so using a representative tree as per their article would be overkill here.

Using this predicate, then, the code to actually build the undirected plane set is quite simple (see Listing 2). Ignoring the templated stuff (which is

```
struct PlanePred
{
  double m_angleTolerance, m_distTolerance;
  PlanePred(double angleTolerance,
     double distTolerance)
  : m_angleTolerance(fabs(angleTolerance)),
     m_distTolerance(fabs(distTolerance))
  {}

  bool operator()(const Plane& lhs,
                  const Plane& rhs) const
  {
    // If these planes are nearly the same (in
    // terms of normal direction and distance
    // value), then !(lhs < rhs) && !(rhs < lhs).
    double angle = acos(
       lhs.normal().dot(rhs.normal()));
    double dist = lhs.distance_value()
       - rhs.distance_value();
    if(fabs(angle) < m_angleTolerance &&
       fabs(dist) < m_distTolerance)
      return false;

    // Otherwise, compare the two planes
    // "lexicographically".
    const Vector3d& nL = lhs.normal(),
       nR = rhs.normal();
    const double& aL = nL.x, bL = nL.y, cL = nL.z;
    const double& aR = nR.x, bR = nR.y, cR = nR.z;
    const double& dL = lhs.distance_value(),
       dR = rhs.distance_value();

    return ((aL < aR) ||
      (aL == aR && bL < bR) ||
      (aL == aR && bL == bR && cL < cR) ||
      (aL == aR && bL == bR && cL == cR &&
         dL < dR));
  }
};
```

Listing 1

necessary so that I can pass in either textured or non-textured polygons as input), all that's happening is as follows:

1. The undirected plane set is initialised with an instance of the predicate we created above. We pass in tolerance values to this for use in deciding when two planes are sufficiently similar.
2. We determine the undirected plane for each polygon in our level, and insert it into the set.

Initial port generation

Having determined the planes in which the portals may lie, we now need to generate an initial portal on each of these planes. This should be a huge polygon which is large enough to span the entire level: the idea is that it's large enough to represent the entire plane for our purposes. We'll then clip it to the tree, which will give us the list of portals which lie on that plane (although as previously mentioned, we'll still need to make a reverse-facing copy of each of them).

Generating the initial portal itself is something that can be done in a variety of ways. The method I used (see Figure 2, which shows building an intiial portal on a plane) works as follows:

1. Generate an arbitrary unit vector, u, in the plane. To do this, we just calculate n x (0,0,1) (where n is the plane normal) and normalize the result, provided the angle between n and (0,0,1) isn't too small (since the cross product of two vectors which point in the same direction is the zero vector). If it is, we simply replace (0,0,1) by (1,0,0). Either way, we eventually end up with a vector which is perpendicular to n: it thus lies in the plane. (This gives us one axis of a coordinate system in the plane.)
2. Calculate u x n and normalize the result, to give another unit vector, v, in the plane which is perpendicular to u. (This gives us the other axis of the coordinate system.)
3. Project the world origin (0,0,0) onto the plane along the normal to give a point, o, in the plane. (This is the origin of the coordinate system.)
4. Generate a large square polygon on the planes with vertices at $o + k(-u - v)$, $o + k(u - v)$, $o + k(-u + v)$ and $o + k(u + v)$, for some arbitrarily large number k. (In practice, I chose $k = 1000000$: if you make it too large, you get small floating-point errors.)

```
template <typename Vert, typename AuxData>
typename PortalGenerator::PlaneList_Ptr
PortalGenerator::find_unique_planes(
   const std::vector<shared_ptr<Polygon<Vert,
   AuxData> > >& polygons)
{
  typedef Polygon<Vert,AuxData> Poly;
  typedef shared_ptr<Poly> Poly_Ptr;
  typedef std::vector<Poly_Ptr> PolyVector;

  const double angleTolerance = 0.5 * PI / 180;
  // convert 0.5 degrees to radians
  const double distTolerance = 0.001;
  std::set<Plane, PlanePred> planes(
     PlanePred(angleTolerance, distTolerance));

  for(PolyVector::
     const_iterator it=polygons.begin(),
     iend=polygons.end(); it!=iend; ++it)
  {
    Plane plane = make_plane(
       **it).to_undirected_form();
    planes.insert(plane);
  }

  return PlaneList_Ptr(
     new PlaneList(planes.begin(), planes.end()));
}
```

Listing 2

The code is shown in Listing 3, generating a large polygon on a plane.

Portal clipping

We now come to the most interesting bit of the portal generation algorithm: clipping the initial portal to the tree. This is done recursively, starting from the tree's root node (see Listing 4).

At each stage of the recursive process, we clip a fragment of the initial portal (initially the whole thing) against a node of the tree. If the node is

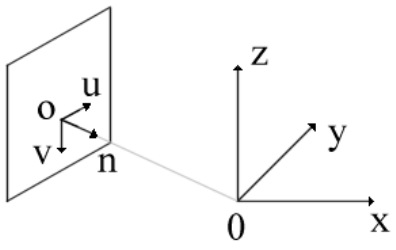


Figure 2

```
double displacement_from_plane(const Vector3d& p,
   const Plane& plane)
{
  const Vector3d& n = plane.normal();
  double d = plane.distance_value();

  // Note that this equation is valid precisely
  // because the plane normal is guaranteed to be
  // unit length by a datatype invariant of the
  // Plane class.
  return n.dot(p) - d;
}

Vector3d generate_arbitrary_coplanar_unit_vector(
   const Plane& plane)
{
  const Vector3d& n = plane.normal();
  Vector3d up(0,0,1);

  if(fabs(n.x) < EPSILON && fabs(n.y) < EPSILON)
  {
    // Special Case: n is too close to the
    // vertical, so n x up is roughly equal to
    // (0,0,0)

    // Use a different vector instead of up (any
    // different vector will do) and apply the
    // same method as in the else clause using the
    // new vector.
    return n.cross(Vector3d(1,0,0)).normalize();
  }
  else
  {
    // The normalized cross product of n and up
    // satisfies the requirements of being
    // unit length and perpendicular to n (since
    // we dealt with the special case where n x up
    // is zero, in all other cases it must be
    // non-zero and we can normalize it to give us
    // a unit vector) return
    // n.cross(up).normalize();
  }
}

template <typename AuxData>
shared_ptr<Polygon<Vector3d,AuxData>
   > make_universe_polygon(const Plane& plane,
   const AuxData& auxData)
```

Listing 3

```
{
  typedef Polygon<Vector3d,AuxData> Poly;
  typedef shared_ptr<Poly> Poly_Ptr;

 Vector3d origin(0,0,0);
  Vector3d centre = nearest_point_in_plane(
     origin, plane);

  Vector3d planarVecs[2];
  planarVecs[0] = generate_arbitrary_coplanar
     _unit_vector(plane);
  planarVecs[1] =  planarVecs[0].cross(
     plane.normal()).normalize();

  const double HALFSIDELENGTH = 1000000;
  for(int i=0; i<2; ++i) planarVecs[i]
     *= HALFSIDELENGTH;

  std::vector<Vector3d> vertices;
  for(int i=0;
     i<4; ++i) vertices.push_back(centre);
  vertices[0] -= planarVecs[0];
  vertices[0] -= planarVecs[1];
  vertices[1] -= planarVecs[0];
  vertices[1] += planarVecs[1];
  vertices[2] += planarVecs[0];
  vertices[2] += planarVecs[1];
  vertices[3] += planarVecs[0];
  vertices[3] -= planarVecs[1];

  return Poly_Ptr(new Poly(vertices, auxData));
}

Vector3d nearest_point_in_plane(const Vector3d& p,
   const Plane& plane)
{
  /*
  Derivation of the algorithm:

  The nearest point in the plane is the point we
  get if we head from p towards the plane along
  the normal.
  */

  double displacement = displacement_from_plane(
     p, plane);
  return p - displacement * plane.normal();
}
```

Listing 3 (cont'd)

a branch node, we classify the portal against the node's split plane and take different actions depending on the result; if the node is a leaf, we discard the portal fragment if the leaf is solid, and (provided the leaf doesn't straddle the portal) note the leaf index in the portal fragment if the leaf is empty (this is done so that we can keep track of which empty leaves a valid portal connects). If an empty leaf does straddle a portal fragment (something which can easily happen: see Figure 3), we discard the fragment, since it isn't a doorway between two separate leaves.

In Figure 3, the only valid portal is represented by the dotted line between α and β (in particular, the potential portal between α and itself is invalid)

```
/**
Clips the portal to the tree and returns a list
of portal fragments which survive the clipping
process.
*/
std::list<Portal_Ptr>
   PortalGenerator::clip_to_tree(
   const Portal_Ptr& portal,
   const BSPTree_Ptr& tree)
{
  return clip_to_subtree(portal, tree->root());
}
```

Listing 4

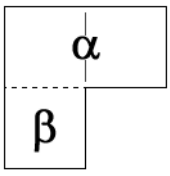


Figure 3

```
/**
Clips the portal to the subtree and returns a list
of portal fragments which survive the clipping
process.

@param portal           The portal to clip
@param subtreeRoot      The root of the subtree
@param relativeToPortal The location of the
                        subspace represented
                        by the subtree relative
                        to the portal (in front,
                        behind, or straddling
                        it)
@return                 As stated
*/

std::list<Portal_Ptr>
PortalGenerator::clip_to_subtree(
   const Portal_Ptr& portal,
   const BSPNode_Ptr& subtreeRoot,
   PlaneClassifier relativeToPortal)
{
  if(subtreeRoot->is_leaf())
  {
    const BSPLeaf *leaf = subtreeRoot->as_leaf();

    if(leaf->is_solid()) return PortalList();

    switch(relativeToPortal)
    {
    case CP_BACK:
    {
      portal->auxiliary_data().fromLeaf =
         leaf->leaf_index();
      break;
    }
    case CP_FRONT:
    {
      portal->auxiliary_data().toLeaf =
         leaf->leaf_index();
      break;
    }
    default:  // CP_STRADDLE
    {
      // The portal fragment is in the middle of a
      // leaf (this is not an error, but we do
      // need to discard the portal fragment as
      // we'd otherwise have a portal linking a
      // leaf to itself).
      return PortalList();
    }
    }
    PortalList ret;
    ret.push_back(portal);
    return ret;
  }

  else
  {
    const BSPBranch *branch =
       subtreeRoot->as_branch();
    switch(classify_polygon_against_plane(*portal,
       *branch->splitter()))
    {
```

Listing 5

```
    case CP_BACK:
    {
      return clip_to_subtree(portal,
         branch->right(), relativeToPortal);
    }
    case CP_COPLANAR:
    {
      BSPNode_Ptr fromSubtree;
      BSPNode_Ptr toSubtree;
      if(branch->splitter()->normal().dot(
         portal->normal()) > 0)
      {
        fromSubtree = branch->right();
        toSubtree = branch->left();
      }
      else
      {
        fromSubtree = branch->left();
        toSubtree = branch->right();
      }
      PortalList fromPortals = clip_to_subtree(
         portal, fromSubtree, CP_BACK);
      PortalList ret;
      for(PortalListCIter it=fromPortals.begin(),
         iend=fromPortals.end(); it!=iend; ++it)
      {
        ret.splice(ret.end(), clip_to_subtree(*it,
           toSubtree, CP_FRONT));
      }
      return ret;
    }
    case CP_FRONT:
    {
      return clip_to_subtree(portal,
         branch->left(), relativeToPortal);
    }
    case CP_STRADDLE:
    {
      // Note: The leaf links for the two half
      // polygons are inherited from the original
      // polygon here.
      SplitResults<Vector3d,PortalInfo> sr =
         split_polygon(*portal,
         *branch->splitter());
      PortalList frontResult =
         clip_to_subtree(sr.front, branch->left(),
         relativeToPortal);
      PortalList backResult =
         clip_to_subtree(sr.back, branch->right(),
         relativeToPortal);
      PortalList ret;
      ret.splice(ret.end(), frontResult);
      ret.splice(ret.end(), backResult);
      return ret;
    }
    }
  }

  // The code will never actually get here,
  // because the switch above is exhaustive,
  // but the compiler still warns us because it
  // can't tell that.
  throw Exception("This should never happen");
}
```

Listing 5 (cont'd)

The tricky bit is what to do for the various branch node cases, e.g. what action should we take if the portal fragment straddles the node's split plane? The cases where the portal is entirely behind or in front of a split plane are easy: we recurse down the appropriate side of the tree. For the straddling case, it suffices to split the portal across the plane, pass each half down the appropriate side of the tree, and then concatenate the results (see Listing 5).

we work out whether the portal is facing in the same direction as the plane or not

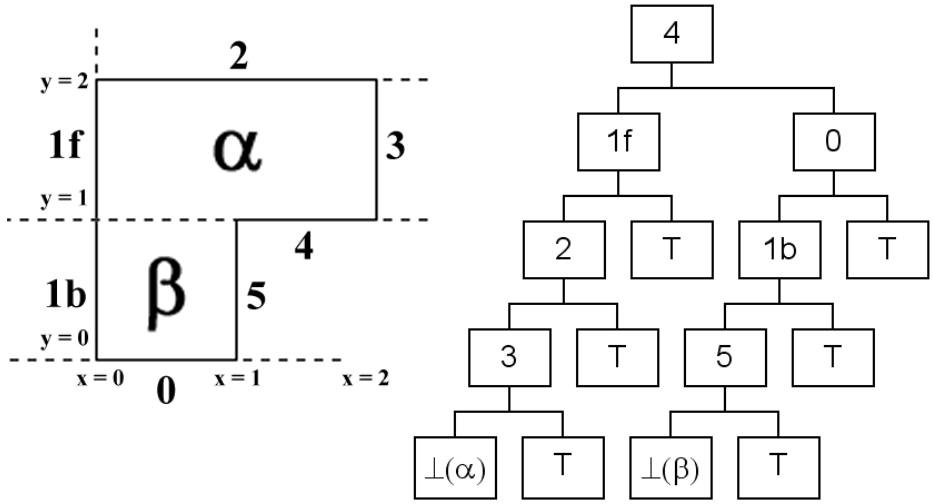


Figure 4

The coplanar case is more intricate. First of all, we work out whether the portal is facing in the same direction as the plane or not by comparing the dot product of their normals to 0 (they're facing the same way if the dot product is positive). This determines which subtree of the current node is the from subtree (i.e. its root represents a convex subspace entirely behind the portal) and which is the to subtree (its root represents a convex subspace entirely in front of the portal). Having determined this, we pass the portal down one of the subtrees (the from subtree in my code) and clip it to the tree. We then clip the portal fragments which survived that clipping process to the other subtree (the to subtree in my code), and concatenate the results. Finally, we return the list of fragments which survived being clipped down both subtrees.

It is worth remarking on the role of the `relativeToPortal` function parameter in this process: it is there to indicate whether the subspace represented by the current node is in front of, behind, or straddling the portal. It is `CP_STRADDLE` at the start of the process (since the entire world space certainly straddles any portal), and only becomes either `CP_FRONT` or `CP_BACK` when the portal lies on a branch node's split plane (i.e. in the coplanar case we've just been discussing). At this point, we use `relativeToPortal = CP_BACK` for the `from` subtree (since that's entirely behind the portal) and `relativeToPortal = CP_FRONT` for the `to` subtree (since that's entirely in front of the portal). This allows us to correctly handle what happens to the portal when it ends up in a leaf.

Bringing things together

We've now seen how to find the undirected plane set, how to generate an initial portal on each plane, and how to clip that portal to the tree. All that remains is to show the top-level code which ties all of this together and makes the reverse-facing copies of each portal (see Listing 6).

The only new bit in this is the code which makes the reverse-facing portals. This is largely trivial: all that's necessary is to flip the portal winding and switch the from and to leaves in the portal's auxiliary information structure.

Example

It would be remiss of me not to show an example of all this in action, so let's walk through a bit of the portal generation for the small L-shaped room in Figure 4. To follow along, it might be easier if you work it through on a piece of paper!

- The undirected plane set (where plane $ax + by + cz = d$ is represented by the quadruple (a,b,c,d)) is $\{(1,0,0,0), (1,0,0,1), (1,0,0,2), (0,1,0,0), (0,1,0,1), (0,1,0,2)\}$, i.e. $x = 0$, $x = 1$, $x = 2$, $y = 0$, $y = 1$ and $y = 2$.
- Portal P_1 on plane $x = 0$
  - Straddles 4 → split into P_{1f} and P_{1b} and recurse down each side
    - P_{1f} is on the plane of 1f and same facing → the back subtree is the from subtree and the front subtree is the to subtree; pass down from subtree first
      - Solid leaf → discard portal
    - P_{1b} straddles 0 → split into P_{1bf} and P_{1bb} and recurse down each side

only one portal ... has survived the clipping process

- P_{1bf} is on the plane of 1b and same facing ⟶ from := back, to := front; pass down from subtree first
  - Solid leaf ⟶ discard portal
- P_{1bb} is in a solid leaf ⟶ discard it

■ Portal P_4 on plane $y = 1$

◆ On the plane of 4 and same facing ⟶ from := back, to := front; pass down from subtree first

- In front of 0 ⟶ recurse down front subtree
- Straddles 1b ⟶ split into P_{4f} and P_{4b} and recurse down each side
  - P_{4f} straddles 5 ⟶ split into P_{4ff} and P_{4fb} and recurse
    - β is empty ⟶ becomes from leaf of P_{4ff}
    - P_{4fb} is in a solid leaf ⟶ discard it
  - P_{4b} is in a solid leaf ⟶ discard it

◆ Now pass surviving fragments (i.e. P_{4ff}) down the to subtree of 4

- In front of 1f, then 2, then 3, so ends up in leaf α, which becomes its to leaf

■ Portal P_5 on plane $x = 1$

◆ Straddles 4 ⟶ split into P_{5f} and P_{5b} and recurse down each side

- P_{5f} is in front of 1f ⟶ recurse down front subtree
- P_{5f} straddles 2, so split it into P_{5ff} and P_{5fb} and recurse
  - P_{5ff} is in front of 3 ⟶ recurse down front subtree
    - α straddles P_{5ff}, so discard it
  - P_{5fb} ends up in a solid leaf ⟶ discard it
- P_{5b} straddles 0, so split it into P_{5bf} and P_{5bb} and recurse
  - P_{5bf} is in front of 1b ⟶ recurse down front subtree
  - P_{5bf} is on the plane of 5 and opposite facing ⟶ from := front, to := back; pass down from subtree first
    - β is empty ⟶ becomes from leaf of P_{5bf}
  - Now pass P_{5bf} down the to subtree
    - Solid leaf ⟶ discard portal
  - P_{5bb} ends up in the solid leaf behind 0 ⟶ discard it

```
template <typename Vert, typename AuxData>
typename PortalGenerator::PortalList_Ptr
PortalGenerator::generate_portals(
   const std::vector<shared_ptr<Polygon<Vert,
   AuxData> > >& polygons, const BSPTree_Ptr&
   tree)
{
  PortalList_Ptr portals(new PortalList);
  PlaneList_Ptr planes =
     find_unique_planes(polygons);

  for(PlaneList::const_iterator it=
     planes->begin(), iend=planes->end();
     it!=iend; ++it)
  {
    Portal_Ptr portal = make_initial_portal(*it);
    portals->splice(portals->end(),
       clip_to_tree(portal, tree));
  }

  // Generate the opposite-facing portals.
  for(PortalList::iterator it=portals->begin(),
     iend=portals->end(); it!=iend; ++it)
  {
    Portal_Ptr portal = *it;
    // Construct the reverse portal.
    Portal_Ptr reversePortal(
       portal->flipped_winding());
    const PortalInfo& portalInfo =
       portal->auxiliary_data();
    reversePortal->auxiliary_data() =
       PortalInfo(portalInfo.toLeaf,
       portalInfo.fromLeaf);
    // Insert it after the existing portal in the
    // list.
    ++it;
    it = portals->insert(it, reversePortal);
  }
  return portals;
}
```

Listing 6

I'll leave generating portals on the remaining planes as an exercise for the reader. You'll note that so far only one portal (portal P_{4ff}, which goes from β to α) has survived the clipping process. (This is in fact the only portal – other than its reverse-facing duplicate, which goes from α to β – in this small level.)

Conclusion

In this article, we've seen how to generate portals for a level. Next time, I'll explain how to use these to generate a leaf-to-leaf visibility table for our level as a way of speeding up level-rendering.

References

[Golodetz08] 'Divide and Conquer: Partition Trees and Their Uses', *Overload* #86, August 2008.

[Simmons01] 'Advanced 3D BSP, PVS and CSG Techniques', Gary Simmons and Adam Hoult, Game Institute, 2001.

[Tampieri92] 'Grouping nearly coplanar polygons into coplanar sets', Filippo Tampieri and David Salesin. In *Graphics Gems III* (ed. David Kirk), Academic Press, San Diego, July 1992.

Orderly Termination of Programs

Ensuring a clean shutdown is important.
Omar Bashir presents some techniques.

Servers are processes, typically on networked computers, that encapsulate and manage a collection of related resources and present functionality related to these resources to client applications [Coulouris01]. Servers wait continuously for requests from clients. Upon receiving requests, servers service the requests and then wait for further requests. As many other applications depend upon servers, termination of their execution should be achieved in an orderly manner. This allows the dependencies to be brought to a state to allow an error-free restart. Also the clients in the process of being served need to either be notified or allowed to complete their sessions.

Servers can be terminated in one of the following two ways,

1. Sending a special application level termination request.
2. Sending an operating system level signal to the server.

The former is implemented completely at the application level and can be relatively conveniently synchronized with the operation of the server to perform an orderly shutdown. This method is only useful for networked applications.

However, the latter originates from the operating system (albeit triggered by another application) and needs to be handled by a callback mechanism in the destination server. Furthermore, this approach can also be used to implement orderly termination in standalone applications, as this does not require communicating program termination commands via a networked connection.

Most programming languages provide language level constructs to receive signals (including termination signals) from operating systems. Applications can register callbacks to be triggered on the occurrence of these signals. For signals indicating application termination, registered callbacks can initiate an orderly termination of the application. Most such callbacks may follow a pattern. An object-oriented generalization of one of such patterns is discussed in this article. A simple framework that implements this pattern in C++ is described. This implementation is specific to the Linux platform. Application of this framework in single threaded and multithreaded programs is also illustrated.

An orderly termination mechanism

Servers operate in a loop, i.e., wait for requests to arrive. Once the requests arrive, they process these requests and then wait for further requests. The simplest way to manage orderly termination of a server is to set a flag (referred here to as the shutdown flag) when the server is to be terminated. The server should continuously monitor the shutdown flag and if it is set, the server should initiate the termination process.

Omar Bashir had his first experiences with programming trying to interface devices in avionics systems over 15 years ago. His interests in device interfacing have evolved from networking to distributed systems and also architectures and patterns. He is currently working as a software developer for a financial services company.

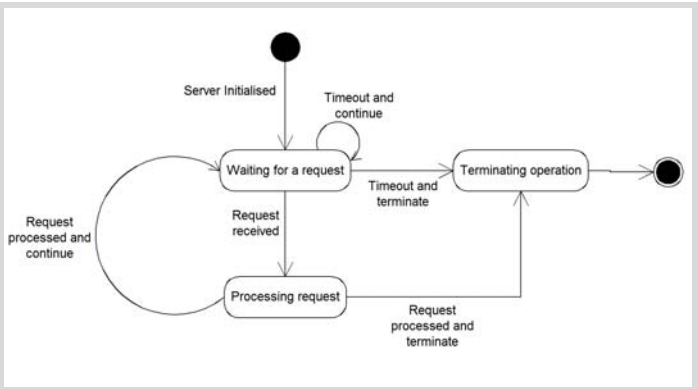


Figure 1

Stevens differentiates servers as iterative and concurrent [Stevens99]. Iterative servers handle only one request at a time. If another request arrives while the server is processing a previous request, the new request is queued and is processed once the previous request has been processed. Managing orderly termination in iterative servers can be relatively straightforward and a possible means of achieving this is shown in the state diagram in figure 1.

The server, after initialization, waits for client's requests. In case the request is received, the server processes the request. After processing the request it checks the shutdown flag to determine if a shutdown was initiated. If that flag is set, the server performs the termination operations, e.g., closing down files, database connections etc. If the flag is not set then the server waits for the next request. In case no request arrives, the server times out and checks the shutdown flag in case program termination was initiated while the server was waiting for the request. If that flag is not set then the server again waits for a client's request. Alternatively, if this flag is set, the server performs the necessary termination operations.

Concurrent servers handle multiple requests at one time. There are several approaches to this. A traditional mechanism is to call the Unix fork function to create a separate child process for each client or request. Alternatively threads are spawned instead of child processes to service each client or client's request. Orderly termination of concurrent servers may require a slightly more elaborate process but follows a pattern similar to that for iterative servers.

The state transition diagram in figure 2 shows orderly termination of concurrent servers. Upon receiving a request, the server will spawn a thread to service the request. If the shutdown flag has been set, the server will wait for all the threads it spawned earlier to terminate before initiating server termination.

Finally, a multithreaded server can have a number of continuously running threads, which may service incoming requests independently (therefore, operating as a concurrent server) or they may assist in servicing a single request (therefore, operating as a multithreaded iterative server). Figure 3 shows orderly termination of a multithreaded server with continuously running threads.

Programs will have to register a callback to receive the program termination signal from the operating system

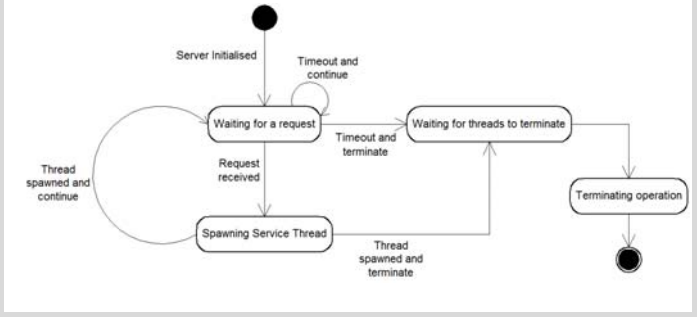


Figure 2

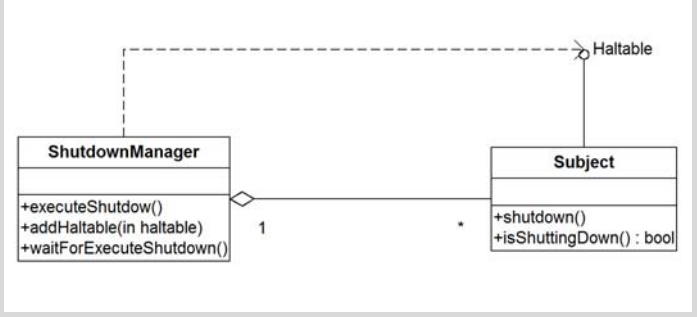


Figure 4

Once the server receives a signal to terminate operation, it signals all the threads to terminate their operations. Individual threads within the server may be viewed as iterative servers for this purpose. After signaling all the threads to terminate, the server waits for the threads to join. Once all the threads have joined, the server terminates its operation.

Class structures and dynamics

In the simplest case, a termination handler is based on two participants. As shown in the class diagram in figure 4, these are the **`ShutdownManager`** class and the list of **`Subject`**s that are notified of imminent program termination by the **`ShutdownManager`**. A **`Subject`** can be the complete application wrapper or individual classes within the application, e.g., database handler, socket wrapper, etc. A **`Subject`** should at least implement the **`Haltable`** interface, which should present a method (**`shutdown()`**) to be called by the **`ShutdownManager`** object to notify imminent program termination and a method (**`isShuttingDown()`**) to determine if the **`Subject`** is shutting down in response to a call to the **`shutdown()`** method.

Programs using this pattern will have to declare an object of the **`ShutdownManager`** class and register all instances of implementations of **`Haltable`** using the **`addHaltable()`** method. Programs will have to register a callback to receive the program termination signal from the operating system. This callback will call the **`executeShutdown()`** method of the **`ShutdownManager`** instance, which will iterate through the list of registered **`Haltable`** instances and call their **`shutdown()`** methods.

Implementation of **`Termination Handler`** as shown in figure 4 may be sufficient for single threaded programs. However, for multithreaded programs, **`Termination Handler`** needs to be extended as shown in figure 5. Here, the **`Subject`** needs to implement the **`Runnable`** interface, which extends the **`Haltable`** interface. **`Subject`**s that need to be executed in separate threads are not directly registered with the **`ShutdownManager`**. Rather they need to be decorated by objects of the **`ThreadOwner`** class (DECORATOR pattern, [Gamma95]. **`ThreadOwner`** also implements the **`Haltable`** interface, therefore allowing objects of the **`ThreadOwner`** class to be registered with the **`ShutdownManager`**. Objects of the **`ThreadOwner`** class execute implementations of **`Runnable`** in separate threads. Furthermore, they can allow orderly shutdown in a thread-safe manner. When called from **`executeShutdown()`**, the **`shutdown()`** method of the **`ThreadOwner`** class locks a mutex and then calls the **`shutdown()`** method of the

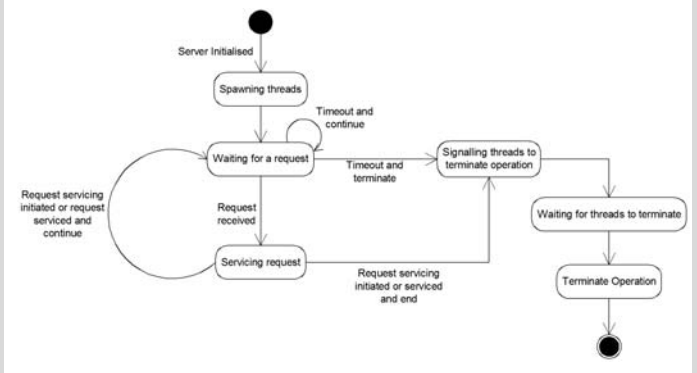


Figure 3

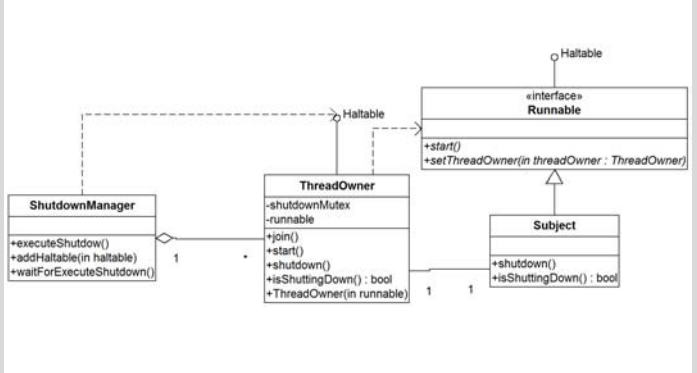


Figure 5

if they have been executing in separate threads, those threads should have joined thus allowing the server to terminate in an orderly manner

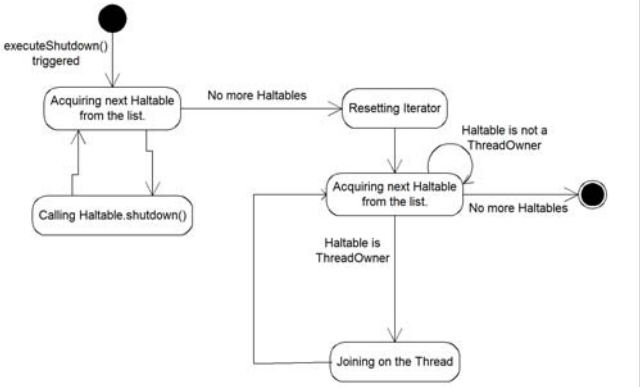


Figure 6

`Subject` allowing the thread in which the `Subject` is executing to terminate appropriately. Similarly, the `isShuttingDown()` method of the `ThreadOwner` class locks the mutex and then calls the `isShuttingDown()` method of the `Subject`. As `isShuttingDown()` method is also called by the `Haltable` implementations internally to determine when to terminate their operations, `Runnable` implementations require a reference to their respective `ThreadOwner` instances so that they can access and invoke the decorated (and thread safe) `isShuttingDown()` method provided by the `ThreadOwner`. The `setThreadOwner()` method of a `Runnable` implementation is used to establish this association.

Once the `executeShutdown()` method of the `ShutdownManager` instance is called, it first iterates through the list of registered `Haltable` instances and calls their respective `shutdown()` methods. Then it again iterates through this list and for every `Haltable`, which is also a `ThreadOwner`, it calls the `join()` method of the `ThreadOwner` to wait for the respective thread to join (Figure 6). Therefore, when `executeShutdown()` returns, all registered `Haltable` instances have been notified of termination and if they have been executing in separate threads, those threads should have joined thus allowing the server to terminate in an orderly manner.

Implementation in C++

Listing 1 shows the C++ implementation of the `Haltable` interface.

As mentioned earlier, each `Haltable` implementation has to implement the `shutdown()` method that specifies the operations to be performed for that `Haltable` implementation upon shutdown. Also `isShuttingDown()` needs to be implemented to allow the calling method to determine if `shutdown()` had already been called and the `Haltable` implementation is in the process of terminating its operation. `isShuttingDown()` will typically be called inside the main functional loop of the `Haltable` implementation so that the loop can be exited in response to a shutdown notification.

```
#ifndef HALTABLE_H_
#define HALTABLE_H_
namespace haltable{
  class Haltable
  {
    public:
      virtual void shutdown(void)  = 0;
      virtual bool isShuttingDown(void) = 0;
      virtual ~Haltable(){}
  };
}
#endif /*HALTABLE_H_*/
```

Listing 1

Listing 2 shows the `Runnable` interface. This C++ implementation of the `Runnable` interface is actually an abstract class. It holds a pointer to the `ThreadOwner` object that will execute a concrete subclass of `Runnable` in a separate thread. As mentioned earlier, this association with the `ThreadOwner` object is required to allow execution of thread safe implementations of `shutdown()` and `isShuttingdown()` methods provided by `ThreadOwner` when invoked from within the `Runnable` implementations. This association is established by calling the `setThreadOwner()` method and passing it the pointer to the `ThreadOwner` object which executes this `Runnable` implementation in a separate thread. Finally, the functionality of a `Runnable` implementation to be executed in its separate thread is implemented in the `execute()` method.

```
#ifndef RUNNABLE_H_
#define RUNNABLE_H_
#include "Haltable.h"
#include <cstdio>
namespace haltable{
  class ThreadOwner;
  class Runnable: public virtual Haltable{
     protected:
       ThreadOwner* threadOwner;
     public:
      Runnable(void):threadOwner(NULL){}
      virtual void execute(void) = 0;
      virtual void setThreadOwner(
          ThreadOwner* threadHandler){
        threadOwner = threadHandler;
      }
      virtual ~Runnable(){}
   };
};
#endif /*RUNNABLE_H_*/
```

Listing 2

```
#include "ThreadOwner.h"

void* haltable::executeInThread(void*
runnableObj){
  Runnable* runnable = (Runnable*) runnableObj;
  runnable->execute();
  pthread_exit(NULL);
  return NULL;
}
```

Listing 4

Listing 3 shows the **ThreadOwner** class which is an implementation of the **Haltable** interface. A **ThreadOwner** instance is used to invoke the **execute()** method of an instance of a **Runnable** implementation in a separate thread. Thus, for multithreaded applications, **ThreadOwner** is used as a **Subject** for the **ShutdownManager** instance. **ThreadOwner** uses a mutex, which is locked when the **shutdown()** and **isShuttingDown()** methods of the **Runnable** instance are called from the **shutdown()** and **isShuttingDown()** implementations of **ThreadOwner** thus avoiding any race condition. Constructor of **ThreadOwner** also calls the **setThreadOwner()** of the **Runnable** instance passed to the constructor as a parameter and passes its own pointer to the **Runnable** instance. This allows the instance of **Runnable** implementation to call the decorated **shutdown()** and **isShuttingDown()** methods provided by the **ThreadOwner** rather than using its own undecorated methods. **start()** method of **ThreadOwner** invokes the **executeInThread()** function in a new thread. The argument of this function is type-casted as a pointer to **Runnable** and then its **execute()** method is invoked to execute the functionality that the **Runnable** instance provides. Listing 4 shows the implmentation of **executeInThread()** function.

Listing 5 shows the **ShutdownManager** implementation. **ShutdownManager** implements a SINGLETON pattern [Gamma95] as only one instance of this class should be responsible for managing **Haltables** and **ThreadOwners** within a program. Therefore, the constructor is private. Pointer to an instance of this class is obtained by calling a static **initialise()** method. This method increments the **refCount** static variable of the class and creates an instance of this class if one has not already been created. Pointer to this instance is assigned to the instance static variable. The pointer to the instance of the class is then returned. Once the instance is no longer required, **dispose()** method of the object is called. This method decrements the **refCount** variable and once the value of the **refCount** variable is zero, the instance of this class being pointed to by the instance class variable is deleted. As both **initialise()** and **dispose()** methods access static class members, they lock a mutex (**initMutex**) to ensure thread safety. Finally, the **terminate()** static method is used to destroy **initMutex** which is statically initialized. The call to this method should be the last statement in a program..

The constructor of this class specifies the **handler()** function as the signal handler for the **SIGTERM** signal, the signal sent to a process to notify its termination. The constructor also creates a pipe [Stevens99b] to communicate that all the **Haltable**s have been notified of shutdown and all the **ThreadOwner**s have joined. The **handler()** function is invoked once the **SIGTERM** signal is received by the process. **handler()** blocks the **SIGTERM** signal and then invokes the **executeShutdown()** method of the **ShutdownManager**'s instance. **executeShutdown()** invokes the **shutdown()** methods of all registered **Haltable**s and then for all **Haltable**s that are also **ThreadOwner**s, it invokes their **join()** methods to wait for them to terminate before proceeding. **executeShutdown()** then writes a character to the pipe created in the **constructor**. The **main()** function of the program should, at the end, call the **waitForExecuteShutdown()** method of the **ShutdownManager**'s instance. **waitForExecuteShutdown()** blocks to read a character from the pipe created in the constructor of **ShutdownManager**. In a multithreaded application, this allows the **main()** function to wait for the **executeShutdown()** method to end

```
#ifndef THREADOWNER_
#define THREADOWNER_
#include "Runnable.h"
#include <pthread.h>
#include <stdexcept>
namespace haltable{
void* executeInThread(void* runnableObj);
enum ThreadStatus {
  THREAD_NOT_CREATED,
  THREAD_CREATED,
  ERROR_CREATING_THREAD
};
class ThreadOwner:public Haltable{
  private:
    std::string name;
    pthread_t threadId;
    Runnable* runnable;
    pthread_mutex_t shutdownMutex;
    ThreadStatus currentStatus;
  public:
    ThreadOwner(const std::string& threadName,
       Runnable* runnableObj):name(threadName),
       runnable(runnableObj),
       currentStatus(THREAD_NOT_CREATED){
      pthread_mutex_init(&shutdownMutex, NULL);
      runnable->setThreadOwner(this);
    }
    const std::string& getName(void){
    return name;
    }
    ThreadStatus start(void){
      if (currentStatus == THREAD_NOT_CREATED){
        if (pthread_create(&threadId, NULL,
            executeInThread, runnable) == 0){
          currentStatus = THREAD_CREATED;
        } else {
          currentStatus = ERROR_CREATING_THREAD;
        }
      }
      return currentStatus;
    }
    void join(void){
      if (currentStatus != THREAD_CREATED){
        std::domain_error exp(
           "Thread has or could not be created.");
        throw exp;
      } else {
        pthread_join(threadId, NULL);
      }
    }
    virtual void shutdown(void) {
      pthread_mutex_lock(&shutdownMutex);
      runnable->shutdown();
      pthread_mutex_unlock(&shutdownMutex);
    }
    virtual bool isShuttingDown(void) {
      bool reply = false;
      pthread_mutex_lock(&shutdownMutex);
      reply = runnable->isShuttingDown();
      pthread_mutex_unlock(&shutdownMutex);
      return reply;
    }
    virtual ~ThreadOwner(){
      pthread_mutex_destroy(&shutdownMutex);
    }
  };
};
#endif /*THREADOWNER_*/
```

Listing 3

```
#ifndef SHUTDOWNMANAGER_H_
#define SHUTDOWNMANAGER_H_
#define SHUTDOWN_MANAGER_DEBUG

#include <list>
#include "Haltable.h"
#include "ThreadOwner.h"
#include <csignal>
#include <pthread.h>
#include <unistd.h>
#include <stdexcept>
#include <iostream>

namespace haltable{
void handler(int sig);

class ShutdownManager{
  private:
    std::list<Haltable*> haltables;
    int pipeDescriptors[2];
    static ShutdownManager* instance;
    static int refCount;
    static pthread_mutex_t initMutex;
    ShutdownManager(void): haltables(){
      struct sigaction termAction;
      sigemptyset(&termAction.sa_mask);
      termAction.sa_handler = handler;
      termAction.sa_flags = 0;
      pipe(pipeDescriptors);
      sigaction(SIGTERM, &termAction, NULL);
    }

  public:
    static ShutdownManager* initialise(void){
      pthread_mutex_lock(&initMutex);
      refCount++;
      if (NULL == instance){
        instance = new ShutdownManager();
      }
      pthread_mutex_unlock(&initMutex);
      return instance;
    }
    void dispose(void){
      pthread_mutex_lock(&initMutex);
      refCount--;
      if ((NULL != instance) && (refCount == 0)){
```

Listing 5

```
        delete instance;
        instance = NULL;
      }
      pthread_mutex_unlock(&initMutex);
    }
    static void terminate(void){
      pthread_mutex_destroy(&initMutex);
    }
    void addHaltable(Haltable* haltable){
      haltables.push_back(haltable);
    }

    void executeShutdown(void){
      for (std::list<Haltable*>::iterator itr =
         haltables.begin();
        itr != haltables.end(); itr++){
          (*itr)->shutdown();
      }
      for (std::list<Haltable*>::iterator itr =
         haltables.begin();
        itr != haltables.end(); itr++){
          ThreadOwner* threadOwner =
             dynamic_cast<ThreadOwner*>(*itr);
         if (threadOwner != NULL){
          try{
            threadOwner->join();
          } catch(const std::domain_error& exp){
            std::cout << exp.what() << std::endl;
          }
        }
      }
      char continueChar = 'x';
      write(pipeDescriptors[1], &continueChar,
         sizeof(continueChar));
    }

   void waitForExecuteShutdown(void){
      char continueChar;
      while (read(pipeDescriptors[0],
         &continueChar,
         sizeof(continueChar)) != 1){}
      close(pipeDescriptors[0]);
      close(pipeDescriptors[1]);
    }
};
};
```

Listing 5 (cont'd)

before ending the `main()` function ensuring that all threads have terminated normally.

Listing 6 shows the implementation of the `handler()` function and also the initialisation of the static data members of `ShutdownManager`. `handler()` obtains the pointer to the instance of `ShutdownManager` by call its `initialise()` static method. Once it has completed its operation, it releases the instance by calling the `dispose()` method on the object.

Example – a single threaded application

Listing 7 shows an implementation of a `Haltable` called the `SingleThreadedTimeLogger`. Its `execute()` method opens a specified file and logs system time in that file after every second as long as the `isShuttingDown()` method returns `false`. Once the `isShuttingDown()` method returns `true`, the `execute()` method exits the loop and closes the file before ending. The `shutdown()` method simply sets the `shutdownFlag` whereas the `isShuttingDown()` method returns the value of that flag.

Listing 8 shows the program that uses an object of `SingleThreadedTimeLogger` class to log time into the specified file.

```
#include "ShutdownManager.h"
haltable::ShutdownManager*
haltable::ShutdownManager::instance = NULL;
int haltable::ShutdownManager::refCount = 0;
pthread_mutex_t
haltable::ShutdownManager::initMutex =
PTHREAD_MUTEX_INITIALIZER;
void haltable::handler(int sig){
  haltable::ShutdownManager* shutdownManager =
     haltable::ShutdownManager::initialise();
  struct sigaction termAction;
  sigemptyset(&termAction.sa_mask);
  termAction.sa_handler = SIG_IGN;
  termAction.sa_flags = 0;
  sigaction(SIGTERM, &termAction, NULL);
  shutdownManager->executeShutdown();
  shutdownManager->dispose();
}
```

Listing 6

```
#ifndef SINGLETHREADEDTIMELOGGER_H_
#define SINGLETHREADEDTIMELOGGER_H_
#include "Haltable.h"
#include <fstream>
#include <string>
#include <ctime>
#include <cstdlib>
#include <unistd.h>
#include <iostream>

class SingleThreadedTimeLogger:
   virtual public haltable::Haltable{
  private:
    std::ofstream outFile;
    bool shutdownFlag;
    int sleepTime;
    std::string message;
  protected:
    std::string outFileName;
    bool openFile(void){
      outFile.open(outFileName.c_str());
      return outFile.good();
    }
    void closeFile(void){
      outFile.close();
    }
    void writeTimeToFile(){
      char buffer[128];
      time_t currentTime = time(NULL);
      ctime_r(&currentTime, buffer);
      outFile << message << " :: " << buffer;
      sleep(sleepTime);
    }
  public:
    SingleThreadedTimeLogger(
       const std::string& fileName,
       const std::string& msg):outFile(),
       shutdownFlag(false),
       sleepTime(1),
       message(msg),
       outFileName(fileName){}
    virtual ~SingleThreadedTimeLogger(){
      if (outFile.is_open()){
        outFile.close();
      }
    }
    virtual void shutdown(void){
      shutdownFlag = true;
    }
    virtual bool isShuttingDown(void){
      return shutdownFlag;
    }
    void execute(void){
      if (openFile()){
        while (!isShuttingDown()){
          writeTimeToFile();
        }
        closeFile();
      } else {
        std::cout << "Error opening "
           << outFileName << std::endl;
      }
    }
};
#endif /*SINGLETHREADEDTIMELOGGER_H_*/
```

Listing 7

The **main()** function of the program instantiates **timeLogger** object of the **SingleThreadedTimeLogger** class. The **main()** function also obtains the reference of **ShutdownManager** class by calling its

```
#ifndef THREADOWNER_
#include "ShutdownManager.h"
#include "SingleThreadedTimeLogger.h"
#include <fstream>
#include <string>
#include <ctime>
#include <unistd.h>
#include <iostream>

int main(void){
  haltable::ShutdownManager* shutdownManager =
     haltable::ShutdownManager::initialise();
  SingleThreadedTimeLogger timeLogger(
     "time_log.txt", "SingleThreadedTimeLogger");
  shutdownManager->addHaltable(&timeLogger);
  timeLogger.execute();
  std::cout << "Exiting application" << std::endl;
  shutdownManager->waitForExecuteShutdown();
  shutdownManager->dispose();
  haltable::ShutdownManager::terminate();
  return 0;
}
```

Listing 8

initialise() static method and assigns it to **shutdownManager** variable. **timeLogger** is added to **shutdownManager** object's list of **Haltable**s by passing its pointer to the **addHaltable()** method. After **timeLogger** has been added to **shutdownHandler**'s list of **Haltable**s, its **execute()** method is called to start the logging process.

To perform an orderly termination of this application, the user may determine the process ID using Linux's **ps** command and then kill the application using the **kill <process ID>** command. As a result, a **SIGTERM** signal is sent to this application. Upon receipt, the **handler()** function in `ShutdownManager.h` is executed to initiate an orderly termination of the program.

The **main()** function releases the instance of the **ShutdownManager** class by calling the dispose method on the object pointed to by **shutdownManager** pointer. Finally, before returning, the **terminate()** static method of the **ShutdownManager** class is called (see Listing 8) .

Example – a multi-threaded application

Listing 9 shows the **ThreadableTimeLogger**, an extension of the **SingleThreadedTimeLogger** (Listing 7) and the **Runnable** (Listing 2) classes. Instances of this class can be executed in separate threads using instances of the **ThreadOwner** class. Implementation of the **execute()** method is similar to that of the **SingleThreadedTimeLogger** except that it uses the associated **ThreadOwner** instance to determine, by calling the **ThreadOwner**'s **isShuttingDown()** method, if its **shutdown()** method has been called. This is because the **ThreadOwner** instance calls **ThreadableTimeLogger**'s **shutdown()** and **isShuttingDown()** methods after locking a mutex to avoid race conditions. As described earlier, the association between a **ThreadableTimeLogger** instance and a **ThreadOwner** instance is achieved via the **Runnable** abstract class's **setThreadOwner()** method.

Listing 10 shows the program that uses three instances of **ThreadableTimeLogger** to log time in three different files concurrently. The **main()** function of this example creates three objects of the **ThreadableTimeLogger** class and three objects of the **ThreadOwner** class. Each **ThreadableTimeLogger** instance is associated with a **ThreadOwner** instance. **ThreadOwner** instances are then added to the list of **Haltable**s in the instance of the **ShutdownManager** class. **start()** is then called on each of these objects to start the respective threads for each of **ThreadableTimeLogger** instances to invoke their **execute()** methods in. **main()** then calls the **waitForExecuteShudown()**

```
#ifndef THREADABLETIMELOGGER_H_
#define THREADABLETIMELOGGER_H_

#include "Runnable.h"
#include "SingleThreadedTimeLogger.h"
#include <pthread.h>
#include <iostream>

class ThreadableTimeLogger:
   public SingleThreadedTimeLogger,
   public haltable::Runnable{
  public:
    ThreadableTimeLogger(
       const std::string& fileName,
       const std::string& msg):
       SingleThreadedTimeLogger(fileName, msg),
       Runnable(){}
    virtual ~ThreadableTimeLogger(){}

    void execute(void){
      if (openFile()){
        while (!threadOwner->isShuttingDown()){
          writeTimeToFile();
        }
        closeFile();
      } else {
        std::cout << "Error opening " <<
           outFileName << std::endl;
      }
    }

    virtual void shutdown(void){
      std::cout << "Shutdown called in
         ThreadableTimeLogger" << std::endl;
      SingleThreadedTimeLogger::shutdown();
    }

    virtual bool isShuttingDown(void){
      std::cout << "isShuttingDown called in
         ThreadableTimeLogger" << std::endl;
      return SingleThreadedTimeLogger::
         isShuttingDown();
    }
};

#endif /*THREADABLETIMELOGGER_H_*/
```

Listing 9

method of the **ShutdownManager**'s instance to block on the pipe internal to the **ShutdownManager**'s instance waiting for the notification by the **executeShutdown()** method to signal that all **Haltable**s have been notified of termination and all **ThreadOwner**s have joined. This application can also be signaled to terminate by using Linux's kill command. As in the previous example, **handle()** will call **executeShutdown()** method of the **ShutdownManager**'s instance. However, as all the **Haltable**s in this case are also **ThreadOwner**s, **executeShutdown()** will wait for all the respective threads to join before returning.

Concluding remarks

Ensuring orderly termination of applications, particularly servers, can end up being complicated. Various resources being used by these applications need to be brought to consistent states for subsequent error-free restart and various clients need to either be notified of the shutdown or their requests completed before the shutdown. Large applications may contain several objects of many different classes that typically are wrappers over system resources and need to be notified of an impending shutdown. This article has described a pattern that can be used in a framework to allow necessary operations to be performed by respective objects once the application has been notified of its termination. An implementation of a framework based on this pattern and two examples of its use are also described. This framework is written in C++ for Linux. ■

```
#include <fstream>
#include "ShutdownManager.h"
#include "ThreadableTimeLogger.h"
#include <ctime>
#include <string>

int main(void){
  haltable::ShutdownManager* shutdownManager =
     haltable::ShutdownManager::initialise();
  ThreadableTimeLogger timeLogger0(
     "time_log_0.txt", "InsideThreadA");
  ThreadableTimeLogger timeLogger1(
     "time_log_1.txt", "InsideThreadB");
  ThreadableTimeLogger timeLogger2(
     "time_log_2.txt", "InsideThreadC");

  haltable::ThreadOwner threadA(
     "ThreadA", &timeLogger0);
  haltable::ThreadOwner threadB(
     "ThreadB", &timeLogger1);
  haltable::ThreadOwner threadC(
     "ThreadC", &timeLogger2);

  shutdownManager->addHaltable(&threadA);
  shutdownManager->addHaltable(&threadB);
  shutdownManager->addHaltable(&threadC);

  if (threadA.start() !=
     haltable::THREAD_CREATED){
    std::cout << "Error starting thread A" <<
       std::endl;
  }
  if (threadB.start() !=
     haltable::THREAD_CREATED){
    std::cout << "Error starting thread B" <<
       std::endl;
  }
  if (threadC.start() !=
     haltable::THREAD_CREATED){
    std::cout << "Error starting thread C" <<
       std::endl;
  }
    shutdownManager->waitForExecuteShutdown();
  std::cout << "Terminating application" <<
     std::endl;
  shutdownManager->dispose();
  haltable::ShutdownManager::terminate();
  return (0);
}
```

Listing 10

Acknowledgements

I am grateful to Ric Parkin and the reviewers for their valuable feedback and encouragement.

References

[Coulouris01] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems, Concepts and Design*, Pearson Education, 2001.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object Oriented Software*, 1995.

[Stevens99] W. R. Stevens, *Unix Network Programming (Volume 1)*, Pearson Education 1999.

[Stevens99b] W. R. Stevens, *Unix Network Programming (Volume 2)*, Pearson Education 1999.

On Management: Caveat Emptor

There are many theories and practices of management. Allan Kelly offers some warnings.

The previous article in this series peeked inside the organizational form. This is a massive subject in its own right and one that determines what management roles exist and what is expected of them. Organizational form is in part a function of what the company is trying to achieve. Rather than discuss structure in detail I want to turn my attention to the roles we find in software development groups. However, before I do so I want use this article to offer some warnings.

Management isn't homogenous

How does one define *management*? Or rather, how does one know management work when we see it?

Perhaps in years gone by management work could be defined as that which was not manual labour. Rather than assembling things on a production line managers organized the production line; rather than dig minerals out of the ground managers concerned themselves with finding people to dig out minerals, selling the minerals and accounting for the profit and loss.

On this definition managers are those who work principally with their brains rather than their hands – some would distinguish between white collar workers and blue collar. But on this basis software developers are managers because their work is principally in the mind.

Alternatively, we could say than managers concern themselves with organizing the work rather than doing it. Extending the analogy to software development, managers are those who don't code. But this leaves out software testers and others. So perhaps the definition is *those who do not directly touch the product during development*.

> **Lesson 1:** What constitutes *Management* and what the dividing line is between *manager* and *worker* has never been well defined and is even more blurred today.

On this basis we might exclude Business Analysts from the management group. Yet (as we will see in a future article) the role filled by Business Analysts in some companies is filled by Product Managers in others. And since Product Managers have the word *Manager* in their title they must be managers, QED.

Unfortunately the word manager is somewhat abused. This is most obvious when writing program code. Naming an object 'Manager' – for example `SecurityManager` or `LogManager` – is usually an indication that the object is poorly understood and ends up being a collection of functions with a vague connection.

Not only is the word *manager* added to a title in an attempt to explain a vague role it is also added as a form of aggrandisement. Title inflation means that some *managers* manage little more than their own time, while those who truly do manage many people – or things – become *Directors*.

A corollary to this confusion is that managers manage different things in different ways. The work of a Product Manager is different to that of a Project Manager which is different to that of a Line Manager which in turn is different to.... – get the picture?

It is a mistake to think that there is a theory of management and that it applies to all those titled Managers. While there are theories of management, they are not universally applicable simply because there is no universal role of manager.

> **Lesson 2:** Different management roles operate in different ways.

Given that there is confusion over who is, and who is not, a manager, and that different managers operate under different conditions with different objectives it is pointless to see a *them and us* divide.

Caveat emptor: the power and danger of theory

> The ideas of economists and political philosophers, both when they are right and when they are wrong, are more powerful than is commonly understood. Indeed the world is ruled by little else. Practical men, who believe themselves to be quite exempt from any intellectual influence, are usually the slaves of some defunct economist. Madmen in authority, who hear voices in the air, are distilling their frenzy from some academic scribbler of a few years back. *John Maynard Keynes* [Keynes36]

When coding, in theory at least, there is a *right* answer – true it isn't always so straightforward in practice but most decisions are contingent. That is to say, given a set of conditions the next action can be predicted.

Even when there is a dispute over the right answer it should be possible to conduct an objective experiment and measure the result – which version executes fastest, which is easiest to understand, etc. Conducting the experiment will not change the context, conduct it again and the result will be the same. Judgement and intuition are usually used to sidestep the need for an experiment and speed things along.

Management isn't like that. Management is about dealing in ambiguous situations, there are many more variables, some variables are unknown and some defy logic – because they often concern people and emotions. Management isn't contingent; intuition and judgement are not short cuts but a way of life.

> **Lesson 3:** Management occurs in ambiguous situations with missing data and incomplete understanding of the problem. It is only sometimes possible to postpone a decision and collect this data. Other times judgement, intuition and clear thinking are required.

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

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

Even if there were a set of rules for management then managers would still need intuition and judgement. The passing of time is not a neutral event. Sometimes waiting a little while can resolve a problem but on other occasions delay can make things worse.

Instead managers seek objectivity in theory. And since management is fundamentally a social science, these theories are social theories that are difficult to replicate in an experiment. In fact, the application of these theories changes the environment.

> **Lesson 4:** There are very few hard facts in management – it is an art not a science.

There is a famous experiment [Goldman96] where pupils at one school were divided into two groups: poor performers and high achievers.

One group of teachers were told that the pupils they were teaching were underperformers and were not expected to achieve much. The other group of teachers were told the opposite; that their pupils were high achievers and great things were expected of them. In fact the pupils had been divided randomly between the two groups. When tested several months later the supposedly 'under performing' pupils did indeed under perform while the 'high achievers' did exactly that.

> **Lesson 5:** You get what you expect, expect the worst and it may well come to pass.

(This experiment was conducted in the 1950s, but today's ethical standards would not permit the 'under performers' to be treated in this way.)

Management theories exist to shape the way in which managers react. Yet they are dangerous for exactly this reason. A manager who mentally holds a poor theory in their head will make decisions based on that theory – exactly the same as they would if the theory was good.

Yet the only way to test a management theory is to use it and in using it the context is changed. There is no such thing as statelessness or side-effect free management.

For these reasons, and others, many management theories are not only unproven but unprovable by way of experimentation. Learned management journals like the *Harvard Business Review* and *MIT Sloan Review* [McFarland08] [Sull07] publish plenty of management theories and advice: many of these articles are based on case studies rather than experiments. Observation and reasoning is about as good as it gets.

Better people than me have pointed out the problems with management theory. If you want a real tour de force on the subject read *Bad Management Theories Are Destroying Good Management Practices*. [Ghoshal05]

> **Lesson 6:** Bad management theories can be very destructive.

For the purpose of this *Overload* series I will strive for objectivity, I will provide references and examples were I can, but it is impossible to be totally objective. What I write will always be coloured by the theories I believe in.

In managing software development work and in advising others on how to improve their software development, I find that it is the hundreds, thousands, of small decisions made every day that make the difference. While it is nice for managers to consider big ideas, like organizational structure and corporate strategy, this only forms a small part of management work. Most management work is in the small everyday decisions.

The thousand small decisions made everyday are usually made based on intuition and our own underlying beliefs. Each big difference is made by a thousand small decisions. Every decision is an opportunity to affect the direction, means and effectiveness of work.

> **Lesson 7:** Making the most of every decision opportunity requires managers to have a clear goal and vision of how the goal can be achieved. A manager's personal philosophy of management plays a key role in ensuring consistent decision making.

Software management

In *Mythical Man Month,* [Brooks75] Fred Brooks wrote:

> In many ways, managing a large computer programming project is like managing any other large undertaking – in more ways than most programmers believe.

Brooks was right; there is much in modern management literature that applies to the management of software projects. Those charged with managing software development can learn a lot by looking beyond the software community for ideas and practices.

Brooks continued to say:

> But in many other ways it is different – in more ways than most professional managers expect.

Perhaps surprisingly some in the management community look to the software profession for examples of good management practices. Software development is both very forgiving of poor practice and very sensitive to it. Even poor management with poor development practices can produce software that generates revenue, but to produce great software and to continue creating great products requires excellence in both domains.

So while managing software work is a lot like managing anything else it is also more different. Which raises the question: does one need experience as a software developer to manage software development? There is no simple *Yes* or *No* answer to this question.

Rather than single out software development let us try some alternative questions:

- Does one need a background as an accountant in order to manage the financial operations of a company?
- Does one need a background in marketing in order to manage the marketing department?

There will always be individuals who are so able, so skilled, that they will be able to turn their hand to managing anything. Such individuals will have quick minds, be good people-managers and be excellent at listening. And it is a truism to say such people are few and far between.

Lesson 8: In general, managing IT work is best done by those who have experience of IT work.

So, on the whole a software development background is necessary to manage a software development group. However such a background is not in itself sufficient to manage a group. Different skills are required, skills such as people-management, listening, political acumen, organization and others. Some will posses these skills already while others will need to learn them.

But, the skills that make someone a good software developer can also trap that person when they move to management. Understanding code is good; however, jumping in to change someone else's code is not good, it undermines trust and responsibility. A fascination with technology can drive a good developer but the same fascination can mislead a manager.

As a developer the first thought upon hearing of a problem should be: it is a problem with the code. As a manager it pays to look beyond the immediate problem, Jerry Weinberg [Weinberg85] has said: Its always a people problem. Looking for a technical problem when the real issue is a people or process issue can be comforting but also wasteful.

In a way, technical problems are the easy bit. Compilers don't get upset when you find a bug in the code, computers don't sulk when your software causes them to crash again and switches don't have bad days and only route half the packets.

The hard bit is the people bit, it's the soft stuff. The truly difficult stuff is the interlocking processes and systems that we find in organizations. The same systems and processes that allow the organization to exist in the first place are also the ones which cause the problems.

Maybe that's why, 25 years after *Mythical Man Month*, Fred Brooks [Brooks95] wrote:

> Some readers have found it curious that *The Mythical Man Month* devotes most of the essays to the managerial aspects of software engineering, rather than the many technical issues. This bias ... sprang from [my] conviction that the quality of the people on a project, and their organization and management, are much more important factors in the success than are the tools they use or the technical approaches they take. ■

References

[Brooks75] Brooks, F. 1975. *The mythical man month: essays on software engineering*: Addison-Wesley.

[Brooks95] Brooks, F. 1995. *The mythical man month: essays on software engineering*. Anniversary edition: Addison-Wesley.

[Ghoshal05] Ghoshal, S. 2005. 'Bad Management Theories Are Destroying Good Management Practices' *Academy of Management Learning & Education* 4(1).

[Goldman96] Goldman, D. 1996. *Emotional Intelligence*: Bloomsbury.

[Keynes36] Keynes, John Maynard. 1936. *The general theory of employment, interest and money*. [S.l.]: Macmillan.

[McFarland08] McFarland, K.R. 2008. 'Should you build strategy like you build software?' *MIT Sloan Management Review* 49(3):7.

[Sull07] Sull, D. 2007. 'Closing the Gap Between Strategy and Execution.' *MIT Sloan Management Review* 48(4):8.

[Weinberg85] Weinberg, G.M. 1985. *The secrets of consulting*. New York: Dorset House.

The Model Student: A Rube-ish Square (Part 1)

We all have nostalgia for favourite childhood toys. Richard Harris looks at the maths behind a classic.

For those of us of a certain age, the creak-creak-creak of an orchestra of Rubik's Cubes [Rubiks] provided the soundtrack to many a childhood lunch break. There are no doubt a few amongst us who were elevated from socially awkward outsiders to school yard super heroes after having learnt the *Secrets of the Cube Masters* [Taylor81]. You may be surprised to read that I was not one of this number. Socially awkward yes, but I never learnt how to solve that fiendish puzzle.

What I, and I suspect many of us, did not realise at the time was that Rubik's Cube is far more than just a toy. It is in fact a physical manifestation of the mathematical subject of group theory.

Group theory can be thought of as the mathematics of symmetry. It was to group theory that I turned when considering the number of tours in the regular travelling salesman problem [Harris07], although I didn't discuss it in those terms in the article. In the case of Rubik's Cube, each rotational manipulation leaves the cube in a state symmetric to the previous; namely a cube. The coloured stickers are simply labels to inform us of which of these symmetric states the cube is currently in.

Rubik's Cube has a staggering number of states; 43,252,003,274,489,856,000 all told [Rubiks]. Indeed, there are so many that the original manufacturer downplayed the figure because they thought that the public wouldn't believe it [Rubiks2]. It is still an open question as to what is the largest number of moves required to return the cube to its initial state from any other state, with the latest estimate being 26 [Kunkle07].

Because of this complexity, I propose that rather than analysing Rubik's Cube itself, we should study a simpler, analogous, problem. That problem is a two-dimensional version of the cube, which I shall call the Rube-ish Square.

The Rube-ish Square is defined as a three by three grid containing the numbers one to nine. Each row and column can be rotated left or right and up or down respectively with the number being pushed out of the grid returning on the opposite side, as illustrated in figure 1.

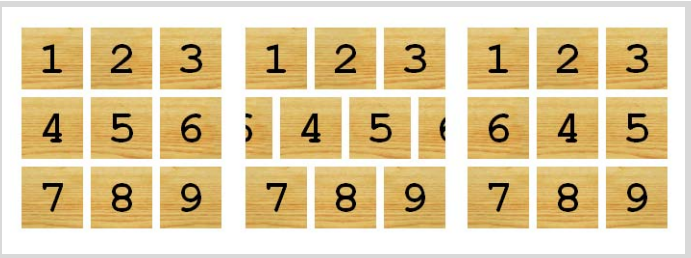


Figure 1

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

The joy of group ... theory

In order to understand the properties of this puzzle we must determine what kind of group it represents, and in order to do that we must first understand what a group actually is.

A group is defined by a set of elements and an associative (i.e. independent of the order of application) binary operator (usually denoted by $\circ$) that uniquely maps a pair of elements from the set onto a third, also from the set. Furthermore, there must be a unique identity element, i, that when combined via the operator with any other element of the set results in that element. Finally, for each element of the set, x, there must be a unique inverse element, denoted x^{-1}, such that when they are both combined, the identity element results. Formally, for a group G this can be expressed as:

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 $a \circ a^{-1} = a^{-1} \circ a = i$

Now, I understand that this definition is rather loaded with mathematical jargon. To clarify, the upside down A means *for all*, the backwards E means *there exists* and the rounded E means *within*.

Translating, 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 $a \circ a^{-1} = a^{-1} \circ a = i$

I suspect that this may not have entirely cleared up matters, so let's look at a specific example; modular arithmetic. Modular arithmetic, otherwise known as clock arithmetic, is defined as the addition of non-negative integers less than a given upper bound with the rule that when a result is equal to or greater than the upper bound, the upper bound is subtracted from it to return it to a number less than the upper bound.

Specifically, if we have an upper bound of n and a and b both less than n (and hence members of our group):

If $a + b < n$ then $a \circ b = a + b$
Else $a \circ b = a + b - n$

This rule ensures that the result of the addition must be greater than or equal to zero and less than n and hence within the group. Since addition is associative, our operator must also be associative. The number zero acts as the identity and for any integer a less than n, the number $n - a$ acts as the inverse, yielding the identity after addition under this rule.

The symmetry of clock arithmetic reveals itself if we connect the numerals with straight lines to form a regular twelve sided polygon

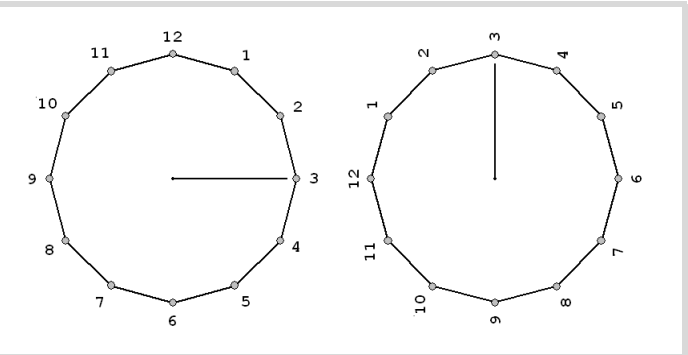


Figure 2

If we choose n to be twelve, we can see why this is also known as clock arithmetic. Working with hours on a twelve hour clock and counting from the previous twelve o'clock we have, for example:

Closed: 5 hours + 9 hours = 2 hours
Associative: 5 hours + (6 hours + 3 hours) = 2 hours
= (5 hours + 6 hours) + 3 hours
Identity: 5 hours + 0 hours = 5 hours
Inverse: 5 hours + 7 hours = 0 hours

Whilst it is clear that clock arithmetic follows the rules required for a group, it is perhaps not so obvious what this has to do with the mathematics of symmetry that I asserted was at the heart of group theory.

The symmetry of clock arithmetic reveals itself if we connect the numerals with straight lines to form a regular twelve sided polygon. Now, instead of moving the hour hand by a given number of hours, we keep it pointing up and rotate the clock beneath it, as illustrated in figure 2.

Each rotation yields an identical, albeit relabelled, polygon, showing that the clock arithmetic captures the rotational symmetry of the dodecagon.

So, now that we have described what groups are, let's take a look at some of their properties.

Firstly, we shall show that my requirement that the identity be unique was superfluous. To see why, let's assume that there are two identities for a given group, i and i'. Consider:

$$i' \circ i$$

Now, by the rule for identities, this implies both:

$$i' \circ a = a \Rightarrow i' \circ i = i$$

and:

$$a \circ i = a \Rightarrow i' \circ i = i'$$

and hence:

$$i' = i$$

So the rule describing identities ensures that they must be unique for any given group.

Secondly, the requirement that the inverse of an element of a group is unique is also redundant. Assuming two inverses of the element a, a^{-1} and a^*, we have by the rule for inverses:

$$a^{-1} \circ a =$$
$$a \circ a^* = i$$

Now the associative rule further implies:

$$(a^{-1} \circ a) \circ a^* = a^{-1} \circ (a \circ a^*)$$
$$i \circ a^* = a^{-1} \circ i$$

and so by the identity rule we have:

$$a^* = a^{-1}$$

proving that each element can have only one inverse.

Groups within groups within groups man

An important concept in group theory is that of the subgroup. A subgroup is comprised of a subset of the elements of a group that within themselves conform to the rules governing groups. The mathematical notation to indicate that a group H is a subgroup of a group G is:

$$H \subseteq G$$

As an example of a subgroup, let's again take the clock arithmetic but this time only with the even-numbered hours. To show that this forms a group, consider each of the rules.

- Since the sum of two even numbers is always even and subtracting twelve from an even number greater than or equal to twelve also results in an even number, this set of numbers is closed under addition modulo twelve.
- We are still dealing with addition, so associativity is a given.
- Zero is an even number, so we have an identity element.
- Subtracting an even number less than twelve from twelve results in an even number less than twelve, so we have an inverse for every number in the set.

Hence the even numbered hours in the clock arithmetic form a subgroup of the clock arithmetic.

The clock arithmetic group has an important additional property that it shares with many, but not all, groups; that of commutativity. Translating into English, this means that the order in which the elements of the group are presented to the operator is irrelevant; that $a \circ b = b \circ a$. Such groups are known as abelian groups and this additional property is formally expressed as:

Commutative: $\forall a, b \in G \Rightarrow a \circ b = b \circ a$

Examples of groups that do not share this property are those of the permutation groups. Permutation groups describe the properties of reordering sets of elements and formed the original definition of groups

now that we are familiar with the basic properties of groups we are ready to investigate the properties of the Rube-ish Square

when group theory was first developed in the early nineteenth century [Baumslag68].

To informally show that permutations can form groups we again consider the rules defining groups.

- If we reorder a set of elements and then reorder it again we trivially have another reordering of the elements and hence permutations are closed.
- If we reorder the elements once and then, considered together, a twice and third time we will have the same permutation that we would have had if we had instead reordered them with the first and second permutations considered together and then the third permutation and hence they are associative.
- We can leave the elements in the order we found them, so they have an identity element.
- Finally, we can sort the elements back into their original order, so each permutation has an inverse.

The set of all possible permutations of a set of n elements is known as the symmetric group of degree n, or S_n. The usual notation for a permutation is two rows of numbers, the first listing the positions of elements before they are reordered and the second listing their positions after they are reordered. For example, one permutation of a set of three elements is:

$$\begin{bmatrix}1&2&3\\3&1&2\end{bmatrix}$$

Applied to the set of elements (a, b, c) this results in (b, c, a). The first element becomes the third, the second becomes the first and the third becomes the second. Applying it again would yield (c, a, b), or the permutation:

$$\begin{bmatrix}1&2&3\\2&3&1\end{bmatrix}$$

Applying it a third time results in (a, b, c), giving us the identity permutation:

$$\begin{bmatrix}1&2&3\\1&2&3\end{bmatrix}$$

The symmetric group of degree three has the following six elements:

$$\begin{bmatrix}1&2&3\\1&2&3\end{bmatrix}\begin{bmatrix}1&2&3\\2&3&1\end{bmatrix}\begin{bmatrix}1&2&3\\3&1&2\end{bmatrix}$$

$$\begin{bmatrix}1&2&3\\1&3&2\end{bmatrix}\begin{bmatrix}1&2&3\\3&2&1\end{bmatrix}\begin{bmatrix}1&2&3\\2&1&3\end{bmatrix}$$

We can show that this group is not abelian by taking a pair of permutations and applying them in both orders:

$$\begin{bmatrix}1&2&3\\2&3&1\end{bmatrix}\circ\begin{bmatrix}1&2&3\\3&2&1\end{bmatrix}=\begin{bmatrix}1&2&3\\2&1&3\end{bmatrix}$$

$$\begin{bmatrix}1&2&3\\3&2&1\end{bmatrix}\circ\begin{bmatrix}1&2&3\\2&3&1\end{bmatrix}=\begin{bmatrix}1&2&3\\1&3&2\end{bmatrix}$$

Note that I've adopted the convention that the permutation on the left hand side of the operator is applied to the set first and the permutation on the right hand side second. The opposite convention, with the permutation on the right hand side applied first, would also describe a group, albeit one in which combining pairs of elements would yield different results.

The first row of elements in the full group above contains the three permutations first described. These are the permutations that swap two pairs of elements whilst the second row contains those permutations that result swap just one pair.

In a similar fashion to the even-numbered hours of the clock arithmetic, permutations with an even number of swapped pairs form a subgroup of a symmetric group, known as the alternating group of degree n, or A_n.

Calling the first permutation described above p, we have already shown that applying it three times results in the identity, so:

$$p \circ p \circ p = p^3 = i$$

and hence:

$$p \circ p = p^2 = p^{-1}$$

giving us inverses for both p and p^2, thus confirming that this is indeed a group.

The permutation p is known as a generator of the group since by repeatedly applying it we generate every element of the group. Formally, a generator of a group is a set of elements that, together with their inverses, yield every element of the group through repeated application of the operator.

Ladies and gentlemen: the point!

So now that we are familiar with the basic properties of groups we are ready to investigate the properties of the Rube-ish Square. The first step is to determine which group captures its symmetries. We can begin by noting that each rotation of a row or column results in different permutation of the squares, so we can represent it using the permutation notation.

By definition the rotations of the rows to their left, the rotations of the columns upwards and their inverse rotations must, by repeated application, generate every possible state of the square since they are the only operations with which we can manipulate it and hence are a generator for our group. We name these r_1, r_2, r_3 and c_1, c_2, c_3 as illustrated in figure 3.

These are represented as resulting states, in permutation notation and by exchanges of elements as follows:

we can exchange any pair of two sequentially adjacent elements of the Rube-ish Square

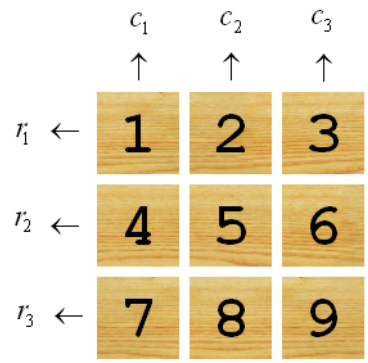


Figure 3

$$r_1 = \begin{bmatrix}2&3&1\\4&5&6\\7&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\3&1&2&4&5&6&7&8&9\end{bmatrix} = (1\leftrightarrow 2)\circ(2\leftrightarrow 3)$$

$$r_2 = \begin{bmatrix}1&2&3\\5&6&4\\7&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&3&6&4&5&7&8&9\end{bmatrix} = (4\leftrightarrow 5)\circ(5\leftrightarrow 6)$$

$$r_3 = \begin{bmatrix}1&2&3\\4&5&6\\8&9&7\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&3&4&5&6&9&7&8\end{bmatrix} = (7\leftrightarrow 8)\circ(8\leftrightarrow 9)$$

$$c_1 = \begin{bmatrix}4&2&3\\7&5&6\\1&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\7&2&3&1&5&6&4&8&9\end{bmatrix} = (1\leftrightarrow 4)\circ(4\leftrightarrow 7)$$

$$c_2 = \begin{bmatrix}1&5&3\\4&8&6\\7&2&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&8&3&4&2&6&7&5&9\end{bmatrix} = (2\leftrightarrow 5)\circ(5\leftrightarrow 8)$$

$$c_3 = \begin{bmatrix}1&2&6\\4&5&9\\7&8&3\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&9&4&5&3&7&8&6\end{bmatrix} = (3\leftrightarrow 6)\circ(6\leftrightarrow 9)$$

The first thing that we should note is that all of these are even permutations since they each swap two pairs of elements of the square. Our Rube-ish Square must therefore be described by either the alternating group of degree nine, or a subgroup of it.

The question as to whether the Rube-ish Square is described by the alternating group is equivalent to the question as to whether any two pairs of elements of the square can be exchanged using a combination of elements from the generator set. We can answer *that* question by examining permutations of the form:

$$a \circ b \circ a^{-1}$$

and, since an element is by definition the inverse of its inverse, also of the form:

$$a^{-1} \circ b \circ a$$

Specifically, we're interested in four of these permutations:

$$c_1 \circ r_1 \circ c_1^{-1} = \begin{bmatrix}1&3&4\\2&5&6\\7&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&4&2&3&5&6&7&8&9\end{bmatrix} = (2\leftrightarrow 3)\circ(3\leftrightarrow 4)$$

$$c_3^{-1} \circ r_2 \circ c_3 = \begin{bmatrix}1&2&4\\5&3&6\\7&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&5&3&4&6&7&8&9\end{bmatrix} = (3\leftrightarrow 4)\circ(4\leftrightarrow 5)$$

$$c_1 \circ r_2 \circ c_1^{-1} = \begin{bmatrix}1&2&3\\4&6&7\\5&8&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&3&4&5&7&6&8&9\end{bmatrix} = (5\leftrightarrow 6)\circ(6\leftrightarrow 7)$$

$$c_3 \circ r_3 \circ c_3^{-1} = \begin{bmatrix}1&2&3\\4&5&7\\8&6&9\end{bmatrix} = \begin{bmatrix}1&2&3&4&5&6&7&8&9\\1&2&3&4&5&8&6&7&9\end{bmatrix} = (6\leftrightarrow 7)\circ(7\leftrightarrow 8)$$

By chaining these and r_1, r_2 and r_3 together we can exchange any pair of two sequentially adjacent elements of the Rube-ish Square. For example, to exchange element 2 and 3 and 5 and 6, we apply the chain:

$$\begin{aligned}(c_1 \circ r_1 \circ c_1^{-1}) \circ (c_3^{-1} \circ r_2 \circ c_3) \circ (r_2) &= (2\leftrightarrow 3)\circ(3\leftrightarrow 4)\circ \\ &\quad (3\leftrightarrow 4)\circ(4\leftrightarrow 5)\circ \\ &\quad (4\leftrightarrow 5)\circ(5\leftrightarrow 6) \\ &= (2\leftrightarrow 3)\circ \\ &\quad (3\leftrightarrow 4)\circ(3\leftrightarrow 4)\circ \\ &\quad (4\leftrightarrow 5)\circ(4\leftrightarrow 5)\circ \\ &\quad (5\leftrightarrow 6) \\ &= (2\leftrightarrow 3)\circ(5\leftrightarrow 6)\end{aligned}$$

We can also manipulate the square itself to confirm that this works:

$$(c_1 \circ r_1 \circ c_1^{-1}): \begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} \rightarrow \begin{bmatrix}1&3&4\\2&5&6\\7&8&9\end{bmatrix}$$

$$(c_3^{-1} \circ r_2 \circ c_3): \begin{bmatrix}1&3&4\\2&5&6\\7&8&9\end{bmatrix} \rightarrow \begin{bmatrix}1&3&2\\5&4&6\\7&8&9\end{bmatrix}$$

$$r_2: \begin{bmatrix}1&3&2\\5&4&6\\7&8&9\end{bmatrix} \rightarrow \begin{bmatrix}1&3&2\\4&6&5\\7&8&9\end{bmatrix}$$

The notation here is that of a mapping and indicates that the permutation on the left of the colon maps, or transforms, the state immediately to the right of it to the one following the arrow. As predicted, the elements 2 and 3 and 5 and 6 have been swapped by this permutation.

The next question is whether being able to exchange any two sequentially adjacent pairs of elements implies that we can exchange any two pairs of elements. To answer this, consider a chain of exchanges of sequentially adjacent pairs in which one pair appears at every step. For example:

$$(1 \leftrightarrow 2) \circ (4 \leftrightarrow 5)$$
$$(1 \leftrightarrow 2) \circ (5 \leftrightarrow 6)$$
$$(1 \leftrightarrow 2) \circ (6 \leftrightarrow 7)$$

The effect of this permutation upon the state of the square is as follows:

$$(1 \leftrightarrow 2) \circ (4 \leftrightarrow 5): \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 3 \\ 5 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$(1 \leftrightarrow 2) \circ (5 \leftrightarrow 6): \begin{bmatrix} 2 & 1 & 3 \\ 5 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$$

$$(1 \leftrightarrow 2) \circ (6 \leftrightarrow 7): \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 3 \\ 5 & 6 & 7 \\ 4 & 8 & 9 \end{bmatrix}$$

With this chain of permutations, we have moved the 4th element to the 7th position, keeping all the elements between the 4th and the 7th in their original order. We can move the original 7th element to the 4th position by applying a similar chain in the opposite direction:

$$(1 \leftrightarrow 2) \circ (5 \leftrightarrow 6): \begin{bmatrix} 2 & 1 & 3 \\ 5 & 6 & 7 \\ 4 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 6 \\ 4 & 8 & 9 \end{bmatrix}$$

$$(1 \leftrightarrow 2) \circ (4 \leftrightarrow 5): \begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 6 \\ 4 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 3 \\ 7 & 5 & 6 \\ 4 & 8 & 9 \end{bmatrix}$$

with the result that we have swapped elements 1 and 2 and 4 and 7.

We could repeat the operation with another non-sequentially adjacent pair together with elements 1 and 2 to exchange two arbitrary pairs since the elements 1 and 2 would be swapped back to their original order.

Unfortunately, this scheme breaks down when we want to move either element 1 or element 2. To employ this approach in this situation we need simply note that there is nothing special about these elements; we could just as easily have chosen elements 8 and 9. As we march a given element up or down through the sequence, we should employ an adjacent pair that will not interfere with the intended step. As the moving element approaches the currently employed pair we can simply swap them with another pair since we have already shown that we can construct a permutation that swaps any two sequentially adjacent pairs of elements. For example, if we wish to swap elements 4 and 2 we could employ the following sequence of exchanges:

$$(1 \leftrightarrow 2) \circ (4 \leftrightarrow 3): \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 4 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$(1 \leftrightarrow 2) \circ (8 \leftrightarrow 9): \begin{bmatrix} 2 & 1 & 4 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 4 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$(8 \leftrightarrow 9) \circ (3 \leftrightarrow 2): \begin{bmatrix} 1 & 2 & 4 \\ 3 & 5 & 6 \\ 7 & 9 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 2 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$(8 \leftrightarrow 9) \circ (3 \leftrightarrow 4): \begin{bmatrix} 1 & 4 & 2 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 3 \\ 2 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Since we only move a single element within the sequence during each step, we can always find a sequentially adjacent pair to exploit that won't interfere with the element's journey. Hence we can construct a permutation that will exchange any two arbitrary pairs of elements using our generator set.

Therefore, the generator set for the Rube-ish Square can generate any even permutation of the elements of the square and so the group that describes it must be the alternating group of degree nine. This means that the number of states of the Rube-ish Square must be equal to the number of elements of the alternating group of degree nine, or:

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

Now, I am well aware that this article has been rather maths heavy and for that you have my sincerest apologies. Thankfully, the next question I wish to raise about the Rube-ish Square is that of determining which state requires the most moves to return it to the initial state and, much as this is an unsolved question for Rubik's Cube, we will not be able to answer this mathematically. Next time, dear reader, there will be code. ■

Acknowledgements

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

References & Further Reading

[Baumslag68] Baumslag, B. and Chandler, B., Group Theory, McGraw-Hill, 1968

[Harris07] Harris, R., 'The Model Student: The Regular Travelling Salesman', *Overload* #82, ACCU, 2007

[Kunkle07] Kunkle, D. and Cooperman, G., 'Twenty-six Moves Suffice for Rubik's Cube', *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ACM Press, 2007

[Rubiks] http://www.rubiks.com

[Rubiks2] http://en.wikipedia.org/wiki/Rubik's_cube

[Taylor81] Taylor, D., *Mastering Rubik's Cube*, Henry Holt & Co, 1981

An Introduction to Fast Format (Part 1): The State of the Art

Writing a good library is hard. Matthew Wilson compares some existing formatting libraries, and promises to do better.

This article series describes FastFormat, a new open-source C++ formatting library that offers a maximal blend of robustness, efficiency and flexibility.

This first instalment will look at the state of the art in C++ formatting, including standard and leading open-source libraries. It will assess the alternatives in terms of software quality characteristics, and consider how they compare with FastFormat.

Examination of FastFormat's extensibility mechanisms and performance will be examined later in the series.

Introduction

FastFormat is one of a generation of libraries that I've been working on over the last few years whose overarching design principle is a refusal to make undesirable and unnecessary compromises between (what I deem to be) the essential characteristics of good software. These characteristics, their tradeoffs, the refusal to compromise, and the full technical details of the concepts, patterns, practices and principles that support this aim will be discussed in my next book, *Breaking Up The Monolith: Advanced C++ Design without Compromise*, which is in preparation and aimed for publication in 2009.

The characteristics I'm interested in include robustness, efficiency, expressiveness, flexibility, discoverability and transparency, portability, and modularity. (If you're unfamiliar with any of these, they are also documented in the Prologue of my second book, *Extended STL, volume 1: Collections and Iterators* [XSTLv1], which is freely available from http://www.extendedstl.com/, along with the preface and several sample chapters.) In the case of formatting, there's an important additional characteristic: internationalisation (I18N) and localisation (L10N).

The basic philosophy of *Monolith*, which FastFormat and its sister libraries uphold, is:

> C++ is hard; so only use it if you must. The (primary) reason you must use it in preference to easier languages is that it affords extremely high efficiency; the reason you use it in preference to C (which is usually equally fast) is that it is massively more expressive (though still less so than many other, easier languages). Consequently, general-purpose C++ libraries must be extremely efficient.
>
> Nonetheless, you shouldn't have to sacrifice expressiveness and flexibility half as much as you might think in order to get that efficiency.

For a subject with such scope, the presentation is necessarily truncated. Some of the topics will be expounded on in a follow on article; others rely on previously published work; some may not recieve the full treatment until *Monolith* is published.

Although this article focuses on FastFormat, it's appropriate to mention its (older) sister library, Pantheios, a logging API library that offers a similar set of characteristics (and is up to two orders of magnitude faster than the competition), and which uses the same technology. In fact, FastFormat came about from a suggestion by Walter Bright (of Digital Mars C++ and the D Programming Language renown) to apply the Pantheios design to string formatting in general. Though FastFormat is less mature than Pantheios, which is already established in large-scale, high-performance commercial systems throughout the world, I hope that it will achieve similar significance. As well as providing an introduction to the library, these articles are also a call to any interested engineers who might like to get involved with the project. (I intend to cover the specifics of Pantheios in a later article.)

Design parameters

Formatting is a core aspect of many C++ programs. A formatting library exists at a relatively low layer of abstraction within an application and should not exhibit characteristics that cause it to intrude in a deleterious manner on application code, or application programmer consciousness. It must not compromise on robustness, efficiency or flexibility, because if it's flaky, slow, or has limited/no compatibility with your application types or with other libraries, you won't use it.

Without any compromise of these factors, it must be expressive and discoverable so it is easy to understand and use, and must facilitate the writing of code that is transparent (and beautiful!). If not, you won't enjoy using it and will be distracted from your real purpose: writing your application. It must also have high modularity and be portable, so you can use it in a wide range of contexts and with a wide range of compilers; if it only works with certain compilers on certain operating systems/architectures, you won't use it for any software that might need to be ported to them.

I believe that FastFormat meets these (sometimes conflicting) obligations more optimally than any other formatting library, and I will illustrate the reasons why this is so throughout these articles, in comparison with two standard libraries, C's Streams and C++'s IOStreams, and two open-source libraries, Loki.SafeFormat [LOKI1, LOKI2] (version 0.1.6) and Boost.Format [BF] (version 1.36.0). Each is a very impressive piece of software engineering, but each also has crucial flaws, as we will see.

Streams, Loki.SafeFormat and Boost.Format are replacement-based formatting APIs, where a format string is used to specify the number, type and location of parameters that will be replaced by arguments presented in the statement. By contrast, IOStreams is a concatenation-based formatting API, where each argument in turn is converted into string form and concatenated together to form the result.

FastFormat provides two different APIs:

- The Format API (hereafter FastFormat.Format), is a replacement-based API
- The Write API (hereafter FastFormat.Write), is a concatenation-based API.

Matthew Wilson is a software development consultant, columnist, and author of *Imperfect C++* and *Extended STL*. He is the creator of the FastFormat, Pantheios and STLSoft libraries. Matthew is curently working on *Breaking Up The Monolith: Advanced C++ Design Without Compromise*.

There are use cases where having **unreferenced arguments** is valid

Terminology

Sink

A sink is an entity that will receive the results of the formatting. Sinks have traditionally been file streams (including console) and character buffers, but can in principle be any type that can make use of a string.

Format

A format is a string, or an instance of a type interpretable as a string, that defines a format. It is required only for replacement-based APIs.

Argument

An argument is a value inserted/concatenated to form the output. For some libraries it may be only built-in types, for others only strings. For most it can be of arbitrary type, requiring translation into a form understood by the library, usually via a user-defined function.

Replacement Parameter

A replacement parameter is a replacement specification within a format. It may specify an argument index, and may also specify width and/or alignment and/or special formatting.

Example

First, let's take a look at the libraries in use, with a simple example statement. Listing 1 shows it with a notional **`AcmeFormat()`** function.

To emulate this functionality with Streams we'd use **`sprintf()`**, as in Listing 2.

Note the mistake in the calculation: **`forename.size()`** should be multiplied by 2. Tellingly, this was a genuine error made during the preparation of the example, nicely illustrating one of the dangers of the **`printf()`**-family.

The other five are as shown in Listing 3. All the requisite includes are elided for space. Furthermore, in the rest of the article, I will assume the inclusion of the optional header **`fastformat/ff.hpp`**, which does nothing more than include the main header **`fastformat/fastformat.hpp`** and then alias the namespace **`fastformat`** to the more succinct **`ff`**.

Software quality characteristics

We'll now consider each of the libraries in turn, against the quality characteristics.

```
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)
```

Listing 1

```
#include <stlsoft/memory/auto_buffer.hpp>
#include <string>
#include <stdio.h>

const size_t  total = 39 // the literal part(s)
                    + forename.size()
                    + ::strlen(surname)
                    + 21 // enough for any number
                   + 1; // for the nul-terminator
stlsoft::auto_buffer<char>  buff(total);
// allocate space, on stack if poss.

int r = ::sprintf( &buff[0], "My name is %s %s; I
am %d years old; call me %s", forename.c_str(),
surname, age, forename.c_str());
// TODO: handle r < 0
result.assign(buff.data(), size_t(r));

assert("My name is Professor Yaffle; I am 134 years
old; call me Professor" == result);
```

Listing 2

```
// IOStreams:
std::stringstream sstm;
sstm << "My name is " << forename << " " <<
surname << "; I am " << age << " years old; call
me " << forename;

result = sstm.str();
. . . // assert assumed in all other examples

// Boost.Format:
result = boost::str(boost::format("My name is %1%
%2%; I am %3% years old; call me %1%") % forename
% surname % age);

// Loki.SafeFormat
Loki::SPrintf(s, "My name is %s %s; I am %d years
old; call me
%s")(forename)(surname)(age)(forename);

// FastFormat.Format:
fastformat::fmt(result, "My name is {0} {1}; I am
{2} years old; call me {0}", forename, surname,
age);

// FastFormat.Write:
fastformat::write(result, "My name is ", forename,
" ", surname, "; I am ", age, " years old; call me
", forename);
```

Listing 3

the robustness of a piece of software is usually concerned with whether it operates correctly when used in accordance with its own design

Robustness

Discussion of the robustness of a piece of software is usually concerned with whether it operates correctly when used in accordance with its own design. In other words, whether it is defective. With libraries it is worthwhile to consider also whether the library engenders correct use: it should be easy to use correctly, and hard to use incorrectly.

In the case of the robustness of formatting libraries, we can identity several aspects to robustness:

- Defective format specification
- Defective arguments
- Atomicity

Defective format specification

There are three kinds of defective format specification:

1. (For libraries whose replacement parameters specify types) the required types may not match the argument types
2. Too few arguments are specified for the format being used
3. One or more specified arguments are unreferenced in the format.

The first is easy to illustrate, using the Streams library:

```
char const* name = "The Thing";
int         mass = 200;

printf("name=%s, mass=%skg\n", name, mass);
```

This is defective, and will not produce the intended output: It may well fault in a way that will stop your process (and you must hope that it does!). Some compilers proffer warnings in such cases, but can only do so if the format string is a literal in the same statement, so the help is limited. We may claim that Streams is not robust because it so readily facilitates the writing of defective code. Furthermore, it is possible to use it in a manner that violates its design, leading (hopefully) to hard faults. In neither case is the compiler able to prevent you.

Loki fails a little more gracefully; it detects the mismatch, stops any further argument processing and output, and the statement evaluates to -1. Boost.Format and FastFormat.Format are not vulnerable to this issue.

All four are subject to the other kinds. First, too few arguments:

```
printf("name=%s, mass=%skg\n", name);
std::cout << (boost::format("name=%1%,
   mass=%2%kgln") % name);
Loki::FPrintf("name=%s, mass=%skg\n")(name);
ff::fmtln(std::cout, "name={0},
   mass={1}kg", name);
```

Boost.Format and FastFormat.Format both throw exceptions, to ensure that client code cannot fail to be informed of the defective format specification. Loki.SafeFormat output stops at the point in the formatting corresponding to the first missing argument, and the statement returns the value -1. `printf()` will fault in some way or another, hopefully fatally.

It is important to note a significant difference between the software contracts of the libraries, insofar as where the contract violation occurs. It is a precondition of `printf()` (and its relatives) that every replacement parameter in the format string has a corresponding argument of the same type, or a type for which there is a known good conversion (e.g. `short`⇒`int`, `float`⇒`double`). Failure to provide such a correspondence is to have violated the contract, and thereby written a defective program. This is quite different from the case of Boost.Format and FastFormat.Format. They do not deem a case of mismatched format and arguments as a violation of the library's software contract. Rather, the libraries provide the means to detect and report such mismatches in a precisely defined way: it is part of their (well-functioning) behaviour. The onus on recognising this condition is on the client code, which, in all likelihood, is defective, and should be terminated accordingly. But that determination is outside the purview of the formatting library. (Note: an important side effect of this 'raising the defect level' is that such libraries are far more amenable to the application of automated testing.)

Finally, let's consider the case of too many arguments:

```
printf("name=%s", name, mass);
std::cout << (boost::format("name=%1%") % name
   % mass);
Loki::FPrintf("name=%s")(name)(mass);
ff::fmtln(std::cout, "name={0}", name, mass);
```

Once again, such a circumstance is likely to be as a result of a defective application. In the case of the Streams library, this is not deemed to be defective, and the function operates as if the extra arguments were not there. Loki.SafeFormat appends unreferenced arguments on to the 'completed' formatted string, which I assume is accidental. With Boost.Format and FastFormat.Format (in default mode) an exception is raised and sent to the caller.

There are use cases where having unreferenced arguments is valid, and both Boost.Format and FastFormat.Format support these. With Boost.Format, you can change the exceptional conditions on a per-formatter basis. With FastFormat, you can either change it on a per-program basis at compile-time, or on a per-thread/per-process basis by changing the process/thread mismatch handler. We'll look in more detail at this subject in a subsequent article. (Both libraries also support the suppression of exception reporting when there are too-few arguments, but the use cases for this are pretty few and far between.)

Defective argument types

It may surprise you to learn that some libraries allow you to pass variables of the wrong type, leading to a fault in operation of the application. Consider the following code:

```
wchar_t const* name = L"The Thing";
int            mass = 200;

std::cout << "name=" << name << ",
   mass=" << mass  << "kg" << std::endl;
```

you find out about programmer error at runtime: this is too late

This does not print what the programmer wanted. In fact, it will print something along the lines of

```
name=001237f0, mass=200kg
```

This is a side effect of the ability of the IOStreams to manipulate pointers. Good intentions; terrible consequences. In my opinion, this 'feature' fairly justifies the claim that the IOStreams are not type-safe, and are unfit for purpose.

What I was surprised to learn during the research of this article is that Boost.Format suffers from exactly the same design flaw, and produces similarly useless output. Loki.SafeFormat fares a little better, in at least being aware of the mismatch. However, its weak defective format specification mechanism of returning a result code rather than throwing an exception means that it just prints nothing past the first literal fragment, and unless you're diligently checking the return you won't know it has failed. In all three cases you find out about programmer error at runtime: this is too late.

Neither FastFormat API suffers from this issue. Both of the following lines precipitate a compilation error because the generic components that interpret the arguments into a canonical representation are not defined for wide string types in a multibyte string build (and vice versa).

```
ff::fmtln(std::cout, "name={0}, mass={1}kg",
   name, mass);
ff::writeln(std::cout, "name=", name, ",
   mass=", mass, "kg");
```

The programmer finds out about the error before it becomes a defect in the code. This is a good thing.

Alas, this issue is not limited to defects of mixed character string encodings. Some libraries provide extensibility mechanisms to allow user-defined types to be passed as arguments. Passing an instance of, say, a **`Person`** type by reference will result in a compiler-error unless you've provided a suitable definition of the requisite extensibility mechanism. This is a good thing.

However, if you pass a pointer to a **`Person`** instance, the picture changes significantly. In the following example the statements using IOStreams and Boost.Format will compile whether or not you've provided a definition of how to print a **`Person*`**,

```
Person* pw = new Person("Wilson", . . .

std::cout << "person: " << pw << "\n";
// Compiles!

std::cout << (boost::format(
   "person: %1%\n") % pw); // Compiles!
```

If you have, then it will work according to the programmer's intent. If not, however, it will proceed to write out the pointer value of **`person`**, which is unlikely to be of any use to your users. This is due to the insertion operator overload taking **`void cv*`**. Once again, the worst part of this problem is that you find out that your code is defective only after running the program. This is a bad thing.

With FastFormat, such defects are reported at the earliest possible moment, because they will fail to compile.

```
ff::fmtln(std::cout, "person: {0}", pw);
// Does not compile

ff::writeln(std::cout, "person: ", pw);
// Does not compile
```

It goes without saying that this is a *very* good thing. And it goes further: even if you introduce the extensions that allow FastFormat to understand void pointer arguments, doing so will still not allow the **`Person*`** arguments to be (incorrectly) understood. Hence:

```
#include
   <fastformat/shims/conversion/void_pointers.hpp>

Person* pw = . . .
void*   pv = person;

ff::fmtln(std::cout, "person: {0}", pw);
// Still does not compile
ff::fmtln(std::cout, "pv: {0}", pv);
// Now compiles
```

Atomicity

With IOStreams, Boost.Format and Loki.SafeFormat, each statement element is presented to the stream in turn, with the unfortunate consequence that when the stream is a file/console (either **`std::basic_ostream`** or, where supported, **`FILE*`**) the output from multiple threads/processes can interleave at the granularity of the statement element rather than of the statement.

Library		Is Atomic?
Streams		Yes
IOStreams		No
Boost.Format	stdout (via boost::str())	Yes
	std::cout	No
Loki.SafeFormat	stdout	No
	std::cout	No
FastFormat.Format	stdout	Yes
	std::cout	Yes
FastFormat.Write	stdout	Yes
	std::cout	Yes

This characteristic means that IOStreams, Boost.Format and Loki.SafeFormat are unsuitable for use in multi-threaded environments,

high flexibility would mean facilitating output to different destinations

unless you first convert to a string and send that to the output stream. None of the other libraries considered here suffer from this critical flaw.

Underneath the covers, the other, seemingly atomic, libraries are also converting to a local buffer before presenting that to the low-level I/O layer en bloc. But the crucial point is that they do this for you implicitly, thereby engendering correct use.

Flexibility

Flexibility is about how easily a library lets you do what you need to do, with the types with which you need to do it. For a formatting library, this comes in three areas:

1. The sink types
2. The argument types
3. The format types (for replacement-based APIs only)

Sink types

In terms of sinks, high flexibility would mean facilitating output to different destinations. We're all familiar with writing to console, file and strings, but there's much more to it than that. We might want to write output to a speech synthesiser, a compression component, a GUI message box, or anything else you can think of. Even if you're writing to a 'string', there are many forms of string beyond `std::string`: it might be a character buffer, a string stream, an ACE `ACE_CString`, and so on.

Streams allows for only character buffer and `FILE*` stream sinks. It is not extensible. IOStreams allows for extension to any type of sink via the `streambuf` mechanism [L&K]. There are many examples of such in the canon, from spawned process I/O [PSTREAMS] to speech synthesis [SHAVIT]. It's quite involved, requiring implementing a whole class (with less than obvious semantics), although helper libraries are available [BSTMS].

Boost.Format outputs to `std::basic_ostream` and `std::basic_string`, and is therefore indirectly extensible via IOStreams extension mechanisms. Loki.SafeFormat allows for stream (`FILE*`), IOStream (`std::ostream`), character buffer (`char*`) and string (`std::string`) sinks out of the box, and it also allows for general extension by requiring a single method to be implemented to match the custom sink type.

Similarly, sink flexibility is a first-class aspect of FastFormat's design. By default, the library understands only sink types that provide the `reserve(size_t)` and `append(char const*, size_t)` methods, of `std::string` and other conformant types (e.g. `stlsoft::simple_string`). However, adding support for other sink types is easy, and several stock sinks are provided in the FastFormat distribution (see Table 1). To use them, you need only `#include` the requisite header in your compilation unit.

Sink type	Required #include
Fixed-capacity character buffers	fastformat/sinks/char_buffer.hpp
Fixed-capacity C-style strings	fastformat/sinks/c_string.hpp
STLSoft's auto_buffer	fastformat/sinks/auto_buffer.hpp
FILE*	fastformat/sinks/FILE.hpp
std::ostream (incl. std::cout/cerr)	fastformat/sinks/ostream.hpp
Speech (currently Windows only, using SAPI)	fastformat/sinks/speech.hpp
Vectored file (using UNIX's writev())	fastformat/sinks/vectored_file.hpp
std::stringstream	fastformat/sinks/stringstream.hpp
ACE's ACE_CString	fastformat/sinks/ACE_CString.hpp
ATL's CComBSTR	fastformat/sinks/CComBSTR.hpp
MFC's CString	fastformat/sinks/CString.hpp

Table 1

Argument types

Argument flexibility is undoubtedly the most important. We're all familiar with the limited flexibility of the Streams library: arguments can only be integer, floating-point and character types, C-style strings and pointers (as addresses). Loki.SafeFormat adds to this the ability to pass `std::string`.

The IOStreams, Boost.Format and both FastFormat APIs expand on this by providing the ability to pass instances (either via reference or via pointer) of user-defined types, by defining suitable extension functions. I'm assuming for brevity that you know how to overload insertion operators for your type(s) for IOStreams and Boost.Format.

FastFormat goes much further. Its application layer function templates apply string access shims [IC++, XSTLv1], which define a protocol for generalised representation of objects as strings. Consequently, all types for which string access shim overloads have been defined are understood implicitly. So a large number of types are already compatible with FastFormat out of the box, including `std::basic_string`, `std::exception`, `ACE_CString`, `VARIANT`, `struct dirent`, `struct tm`, `struct in_addr`, `CString`, `FILETIME`, `SYSTEMTIME`, and many more. Because shims are able to introduce compatibility without incurring coupling, you can define shim overloads for your own types and they will automatically work with FastFormat (and with STLSoft, and Pantheios, and any other libraries that use string access shims).

Furthermore, FastFormat provides a second, higher-level, filtering mechanism for extension: it understands any types for which the overloads of the conversion shim [IC++, XSTLv1] `fastformat::filtering::filter_type` have been defined, and considers this before resolving arguments based on string access shim overloads. This *type-filter mechanism* facilitates FastFormat-specific conversion of types for which string access shim overloads have not been defined, e.g. an application-specific user-defined type. The mechanism can also be used for types whose conversion form does not suit your purposes: if you don't like the way, say, `struct tm`, is represented then you can override it.

Expressiveness is 'how much of a given task can be achieved clearly in as few statements as possible'

We'll look at how these mechanisms work, and examples of how they facilitate extension to user-defined types, in a subsequent article.

Format types

The last area of flexibility, for replacement-based APIs, is the format string. With Streams, the format string must be a C-style string. Boost.Format and Loki.SafeFormat also support `std::basic_string`.

FastFormat.Format applies string access shims to its format parameter, which means a potentially infinite set of types. In practice, this flexibility has been most helpful in cases using string classes from other libraries (e.g. ACE, ATL), resource strings, and localised format bundles (again, a follow-on article issue).

Expressiveness

Expressiveness is 'how much of a given task can be achieved clearly in as few statements as possible' [XSTLv1]. Both succinctness and clarity are important, each without trespassing too much on the other.

With a formatting library, expressiveness can be judged in terms of:

- Direct syntactic support for built-in and standard types
- Direct syntactic support for user-defined types
- Specification of width and alignment
- Special formatting, e.g. hexadecimal for integral/pointer types

Direct syntactic support for built-in and standard types

Streams, IOStreams, Boost.Format and Loki.SafeFormat all provide good support for built-in types.

By nature, FastFormat does not understand built-in types, any more than it understands any types that are not, or cannot be represented (via string access shims) as, strings. As noted in the previous section, however, it can be easily extended to understanding any type via the type-filter mechanism.

The library comes with stock type-filters for:

- All integral types (including `int64` / `long long`)
- `float` and `double` floating-point types
- `bool` type
- `char` and `wchar_t` types (except for compilers that define `wchar_t` as a typedef)
- void pointer types (`void*` and its cv-variants)

They're each defined in their requisite header located in the `fastformat/shims/conversion/filter_type` include directory. As a convenience, the type-filter header for integral types is included into `fastformat/fastformat.hpp` by default. This can be switched off via the pre-processor. Automatic inclusion for the other types can be switched on in the same way, if you don't want to have to explicitly include them in your application code.

As for other standard types, all except Streams understand `std::string` (or `std::wstring`): our Streams example illustrates the annoying requirement to explicitly invoke the `c_str()` method.

As mentioned in the section on FastFormat also understands several other standard types. If you want to pass an exception as argument to a format, all other libraries will require you to explicitly invoke the `what()` method.

Direct syntactic support for user-defined types

This one's simple. Streams and Loki.SafeFormat do not allow for arguments of user-defined type. All the others do. To format strings representing instances of user-defined types with Streams and Loki.SafeFormat you have two choices. One option is to perform explicit formatting in application code, which is obviously anything but expressive.

```
printf("person: %s %s, %d\n", bob.forename.c_str(),
   bob.surname.c_str(), bob.age);
Loki::Printf("person: %s %s,
   %d\n")(bob.forename)(bob.surname)(bob.age);
```

The other option is to use a conversion function, which requires more code, is inefficient and still somewhat lacking in expressiveness:

```
std::string Person2String(Person const& person);

printf("person: %s\n", Person2String(bob).c_str());
Loki::Printf("person: %s\n")(Person2String(bob));
```

Specification of width and alignment

All of the libraries except FastFormat.Write offer some ability to specify width and/or alignment. The statements in Listing 4 all print a left-aligned integer in a width of 5, and a right-aligned string in a width of 12 ("`[-3   ,  abcdefghi]`")

```
int         i = -3;
std::string s = "abcdefghi";

printf("[%-5d, %12s]\n", i, s.c_str());

std::cout << (boost::format("[%|-5|, %|12|]\n") %
i % s);

std::cout << "[" <<
std::setiosflags(std::ios::left) << std::setw(5)
<< i << ", " << std::setiosflags(std::ios::right)
<< std::setw(12) << s << "]" << std::endl;

Loki::Printf("[%-5d, %12s]\n")(i)(s);

ff::fmtln(std::cout, "[{0,5,,<}, {1,12}]", i, s);
```

Listing 4

To the FastFormat core, everything is just a string slice

Four of the libraries acquit themselves well in this case, with Loki.SafeFormat probably taking the biscuit. However, the IOStreams statement is a pig. Personally, I've always loathed the IOStreams, and avoided using them wherever possible, and this is a perfect illustration of why.

It's worth nothing that in terms of width and alignment, Boost.Format provides extended facilities for centred alignment and absolute tabulations over multiple fields. FastFormat.Format provides left/right/centred alignment, and can also do absolute tabulations, although it requires a certain indirection. We'll see how in a later article.

Without compromising robustness or efficiency, FastFormat.Format is able to support a good range of formatting/alignment instructions, by defining replacement parameter syntax as:

```
index[, [minWidth][, [maxWidth][, [alignment]]]]
```

The **index** is required, but each of the other fields is optional. The index and widths must be non-negative decimal numbers. The alignment field is zero or one of '**<**' (left-align), '**>**' (right-align), '**^**' (centre-align). The minimum width can be anything up to 999, for implementation reasons; once again, we'll see why in a subsequent article.

Special formatting

Streams is able to format integers as decimal, octal and hex, to select precision for floating-point types, to use zero padding instead of spaces, and so on. Boost.Format and Loki.SafeFormat all provide the same functionality with equal expressiveness. IOStreams also provides these facilities, though you'll find yourself in the same kind of chevron-hell as with width and alignment.

As you can see from all the examples presented thus far, FastFormat's expressiveness is pretty good, on a par with the best performers of the other libraries. Here is where we reach its limit. I have made a strong case for FastFormat's superior robustness characteristics (and will do so regarding its performance characteristics), and the cost is in lower expressiveness in the area of special formatting.

Currently (as of 0.3.1), FastFormat supports no special formatting at all. The two I'm considering adding to FastFormat.Format both involve the case where the argument exceeds the parameter's maximum width, if specified. One option is to fill the whole field with a (per-thread/per-process) customisable character, which would probably default to the hash/pound character '**#**'. The other option is to insert an ellipsis "**...**" into the result. Both cases could be accommodated without compromising robustness or performance. The following example, using syntax that is speculative at this time, shows both options, giving the output **"-3, ########, ...efghi"**:

```
ff::fmtln(std::cout, "[{0}, {1,,8,>#}],
   {1,,8,>.}]", i, s);
```

Three features that have no hope of being accommodated within the current design are:

- Leading zeros (or any other non-space padding)
- Octal/hexadecimal encoding
- Runtime width/alignment specification

To the FastFormat core, everything is just a string slice. It doesn't know anything about integers, floating-points, or user-defined types. So we cannot zero-pad. Well, actually, to add support for this would be trivially simple, since we already support space padding (for minimum width). Unfortunately, it would mean that you could do something like the following (again, the syntax is speculative).

```
ff::fmtln(std::cout, "[{0,5,,>0}, {1,12,^}]",
   i, s);
```

This would produce the result **"[000-3, abcdefghi]"**, rather than the intended **"[-0003, abcdefghi]"**. Because an overriding principle of FastFormat is that it does not allow you to easily do the wrong thing, this will not be supported. The correct way to do this is to use an inserter class, which we'll discuss in detail in a subsequent article. For now, let's look how to do it with the Pantheios integer inserter class. I do this to illustrate the implicit, uncoupled interoperability between FastFormat and other libraries that use string access shims. (I also do it because, at the time of writing, there are not yet any inserter classes written for FastFormat; I've been using the Pantheios ones, and getting on with trickier problems.)

```
#include <pantheios/pan.hpp>
// API; alias namespace pantheios -> pan

#include <pantheios/inserters/integer.hpp>
// pantheios::integer class

ff::fmtln(std::cout, "[{0}, {1,12,^}]"
        , pan::integer(i, 5, pan::fmt::zeroPad), s);
```

Octal/hexadecimal representation is not possible because the arguments have already been turned into string form before the format string is collated. If you want a number to be represented in this way, you need to use an inserter class. Again, until such time as FastFormat has its own, you can 'borrow' the **integer** class from Pantheios:

```
ff::fmtln(std::cout, "10 in hex={0}"
        , pan::integer(10, 8, pan::fmt::fullHex));
```

Finally, specifying widths at runtime is also not possible, again because all arguments are treated as strings. If you need to do that, you must create the format string on the fly. The good news is that you can do this using FastFormat, and without significantly compromising performance. We'll

I'm hoping to interest people in joining the project

see such examples of using FastFormat in 'recursive' mode in a subsequent article.

Discoverability and transparency

Discoverability and *transparency* are the two sides of the comprehensibility of a software component. Essentially, discoverability is how easy the component is to understand in order to use it (including customisations); transparency is how easy it is to understand in order to change it. With both characteristics, judgements are subjective, though not wholly so.

In terms of discoverability, I honestly believe that FastFormat is very good in the majority of its features, though I would have to concede that its more esoteric ones are likely just as undiscoverable as those of Boost.Format. Through force of habit, perhaps, Streams is very discoverable, and Loki.SafeFormat, being very similar to Streams, has that same characteristic. I have always found IOStreams to be the opposite of obvious, and am never able to do any non-trivial IOStreams programming without consulting the documentation. Score them last, in my opinion.

Any non-trivial C++ library, such as these, will suffer in the transparency stakes. Having spent a lot of time delving inside implementations of them all in recent weeks, I would have to say that none are scoring all that well. In my opinion, Loki.SafeFormat is slightly more transparent than the rest, and IOStreams and Boost.Format are considerably worse. Both are effectively opaque to anyone with less patience than a saint. (As, perhaps, is FastFormat too, to anyone other than its creator.)

It is no accident that the discoverability and transparency of Streams and Loki.SafeFormat seem to be superior to the rest, because they are the least flexible libraries: the two characteristics are usually in inverse proportion.

Portability

Being standard, Streams and IOStreams are available on just about every platform you're going to come across. (The only exceptions, pardon the pun, will be certain embedded platforms that don't support exceptions and/or templates.) Loki.SafeFormat is highly generic and contains no compiler-dependencies; as long as your compiler is modern enough to support static array-size determination [IC++] it works just fine. Boost.Format also has extremely high coverage. In terms of compiler capabilities, FastFormat is very portable, and will work with any modern compiler, and several not-so modern ones: It even works with Visual C++ 6!

FastFormat does not rely on compiler/operating-system specific constructs (although it may use them where available), and has been used successfully on Linux, Mac OS-X, Solaris, and Windows, including 32-bit and 64-bit variants of most. Nonetheless, it's likely that there are platforms and/or compilers that are not yet supported, but I'm highly confident that such can be accommodated readily. Part of the reason for writing this article is that I'm hoping to interest people in joining the project to help with such things (and to drive the design to new places, of course).

Modularity

Modularity is about dependencies, usually unwanted ones. This tends to have two forms:

- What else do I need to do/have in order to work with the library
- What else do I need to do/have in order to use the library to work with other things

In terms of the first, we can immediately stipulate that, being standard, the Streams and IOStreams libraries are perfectly modular by definition.

Boost.Format comes as part of Boost, and requires nothing else. Loki.SafeFormat comes as part of Loki, and requires nothing else. Both of these require only the usual download/unpack/build/install aspects of any open-source library.

FastFormat is less modular than the others, in that it requires the STLSoft libraries. However, since STLSoft is 100% header-only, this is a pretty small burden; the only impost is that you define the **`STLSOFT`** environment variable that the FastFormat makefiles expect.

When it comes to the other aspect, only FastFormat offers true modularity. Because its default argument interpretation is done via *string* access *shims* [XSTLv1], it is automatically compatible with any other libraries/applications that use them. For example, you can report results of API functions from the Open-RJ library in FastFormat statements, as in:

```
openrj::ORJRC rc =
   openrj::ReadDatabase(databasePath, . . .

if(ORJ_RC_SUCCESS != rc)
{
  ff::fmtln(std::cerr, "failed to open {0}: ",
     databasePath, rc);
}
```

The resultant string will be formed from the format, the database path (C-style string) and the string form of the result code. If, say, **`databasePath`** is **`"myfile.rj"`** and **`rc`** is **`ORJ_RC_CANNOTOPENJARFILE`**, then the result will be **`"failed to open myfile.rj: the given file does not exist, or cannot be accessed"`**. This all works without the FastFormat and Open-RJ libraries knowing anything about each other. In fact, it works without Open-RJ even having any dependency on STLSoft!

I18N/L10N

Depending on where you get your information, you may see slightly conflicting definitions of Internationalisation (aka I18N) and Localisation (I10N). The definitions I prefer are that I18N is the business of giving software the capability to support different locales, and L10N is the business of using that capability and actually providing support for one or more specific locales. We'll consider the libraries on that basis.

There are two major features required for I18N in a formatting library:

- The ability to convert arguments in a form suitable to the locale
- The ability to arrange arguments in an order suitable to the locale

a new formatting library with big aims: to provide the highest possible quotient from the software quality equation

When it comes to argument conversion, FastFormat is not yet fully internationalised: the converter classes and localised integer conversion functions are not yet written; the only integer conversions currently provided are not I18N, and just do vanilla integer to string conversion. The good news is that all these are addressable, and FastFormat is totally customisable to provide full I18N support: by string access shims, by the `filter_type()` mechanism and by inserter classes. All other libraries are, platform/compiler/standard-library permitting, already fully I18N compatible in how they convert arguments.

Arranging arguments necessitates a replacement-based API, whose format string may contain positional identifiers, such that arguments may be utilised in arbitrary order – determined at runtime, if necessary – dependent on the locale. Boost.Format and FastFormat.Format are the only two libraries from our set that support this requirement. FastFormat also comes with several 'bundles' – user-defined types that associate format strings with keys – from which format strings can be elicited dependent on locale; this will be discussed in a later article.

Efficiency

By this point may be wondering whether the mechanisms that enforce total robustness and allow infinite extensibility impose a performance cost. I am pleased to be able to tell you that this is not so: far from it, in fact.

The next article will take a deeper look into issues of performance, but I want to show you a sneak peek of the performance results for the Professor Yaffle example. (This test is included in the performance tests in the FastFormat distribution.) Table 2 shows the times for 100,000 iterations of the string formatting operation, compiled with GCC and Visual C++ on 32-bit and 64-bit machines.

Library	Time (ms) for 100,000 Yaffles			
	VC++ 9 (x86)	VC++ 9 (x64)	GCC 4.2 (x86)	GCC 4.1 (x64)
Streams	257	175	209	83
IOStreams	734	378	233	186
Boost.Format	2,005	1,145	706	736
Loki.SafeFormat	356	235	342	235
FastFormat.Format	129	88	153	112
FastFormat.Write	112	84	63	66

Table 2

Let's look at the memory allocations involved with the example statement. Table 3 shows the results for four compilers.

It's clear that Boost.Format is the greedy sluggard of the group, reflected in the amount of memory allocations it makes and in the time it takes to prepare statements. Streams is consistently quicker than IOStreams and Loki.SafeFormat, but their relative performance is dependent on compiler/platform (IOStreams on UNIX is surprisingly quick.) But the clear winners

Library	# allocations			
	VC++ 7.1	VC++ 9	GCC 4.2 (x86)	CodeWarrior 8
Streams	2	1	2	3
IOStreams	8	8	2	11
Boost.Format	16	19	16	41
Loki.SafeFormat	3	3	4	6
FastFormat.Format	1	1	1	3
FastFormat.Write	1	1	1	3

Table 3

are the two FastFormat APIs, which (thankfully!) live up to their name; the Write API is somewhat quicker, as we'd expect.

A last word on Loki

Along with Streams, IOStreams and Boost.Format, Loki.SafeFormat has come in for a fair grilling. I would like to point out that it differs from the other three in being a knowingly research/alpha project. Its original author, Andrei Alexandrescu, has pointed out on more than one occasion that it's not yet a polished idea nor of production status: its version number, 0.1.6, is a good indication of that. I've included it in this test because (i) I did not want to be accused of singling out Boost for criticism, and (ii) it's got an interesting interface layer. Once you've read *Monolith* (hint, hint), you'll see how some of the library's deficiencies are not necessarily of the basic design, merely a consequence of a weak tunnel mechanism, and could be remedied by adopting a different one. Consequently, most (though not all) criticisms of its robustness and flexibility issues may be taken with the reservation that such things are improvable.

Summary

In this article we've looked at several formatting libraries and compared at how well they fared against some important characteristics of software quality, with decidedly mixed results; see Table 4. We also introduced Fast Format, a new formatting library with big aims: to provide the highest possible quotient from the software quality equation in this crucial area of almost all programs.

And the impartial recommendation is ...

Well, if it's not obvious by now, I mustn't have belaboured the point quite enough. As I said at the start of the article, a formatting library 'must not compromise on robustness, efficiency or flexibility'. All four established libraries fail this test for the most important of these, robustness. In my opinion, that is the fatal blow. By contrast, FastFormat.Format is as robust as it is possible for a replacement-based format library to be, and FastFormat.Write is completely robust: it is impossible to compile defective code using it!

That FastFormat offers better flexibility (although only slightly in the case of Boost.Format and IOStreams) and substantially better performance is

the cherry on the cake. That FastFormat is permanently a little less expressive than Boost.Format is a small price to pay for the robust+flexible+fast trifecta.

The key, then, is to finish off its I18N support, sort out its packaging and ensure its full portability. I am hoping that readers of these articles will be motivated to help me get it over the line. Then we can just enjoy flexible, reliable formatting that also happens to be exceedingly fast.

The next article(s) will look in detail at FastFormat's extensibility mechanisms and cover some of the ways in which it achieves its high performance. ■

References

[BF] The Boost.Format library; http://www.boost.org/doc/libs/1_36_0/libs/format/index.html

[BSTMS] The Boost.IOStreams library; http://www.boost.org/doc/libs/1_36_0/libs/iostreams/doc/index.html

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

[L&K] *Standard C++ IOStreams and Locales*, Langer & Kreft, Addison-Wesley, 2000

[LOKI1] The Loki library; http://www.sourceforge.net/projects/loki-lib

[LOKI2] 'Typesafe Formatting', Andrei Alexandrescu, *C/C++ Users Journal*, August 2005; http://www.ddj.com/cpp/184401987

[PSTREAMS] http://pstreams.sourceforge.net/

[SHAVIT] audio_stream: A Text-to-Speech ostream, Adi Shavit, March 2007; http://www.codeproject.com/KB/audio-video/audio_ostream.aspx

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

	Streams	IOStreams	Boost.Format	Loki.SafeFormat	FastFormat Write	FastFormat Format
Robustness (type-safety)	Very Low	Medium	Medium	Medium	100%	100%
Robustness (format)	Very Low	n/a	High	Low	High	n/a
Robustness (atomicity)	Yes	no	No	No	Yes	Yes
Flexibility (sink)	Medium	High	High	High	High	High
Flexibility (format)	Low	N/a	Medium	Medium	n/a	High
Flexibility (argument)	Low	High	High	Low	Very High	Very High
Redefines operator semantics	No	Yes	Yes	No	No	No
Expressiveness (UDTs)	Built-in types: high UDTs: n/a	High	High	Built-in types: high UDTs: n/a	High	High
Expressiveness (Width/Align)	Medium	Low	Very High	Medium	n/a	High
Expressiveness (Special Fmt)	Built-in types: high UDTs: n/a	Low	Very High	Built-in types: high UDTs: n/a	n/a	Low
I18N/L10N (conversions)	yes	yes	yes	yes	currently incomplete	currently incomplete
I18L/L10N (ordering)	no	no	yes	no	yes	yes
Portability	Total (Standard)	Total (Standard)	High	High	Medium now High possilbe	Medium now High possible
Modularity (required)	Total (Standard)	Total (Standard)	Relies only on Boost	Relies only on Loki	Relies on STLSoft	Relises on STLSoft
Efficiency	High	Medium	Low	High	Very High	Very High

Table 4