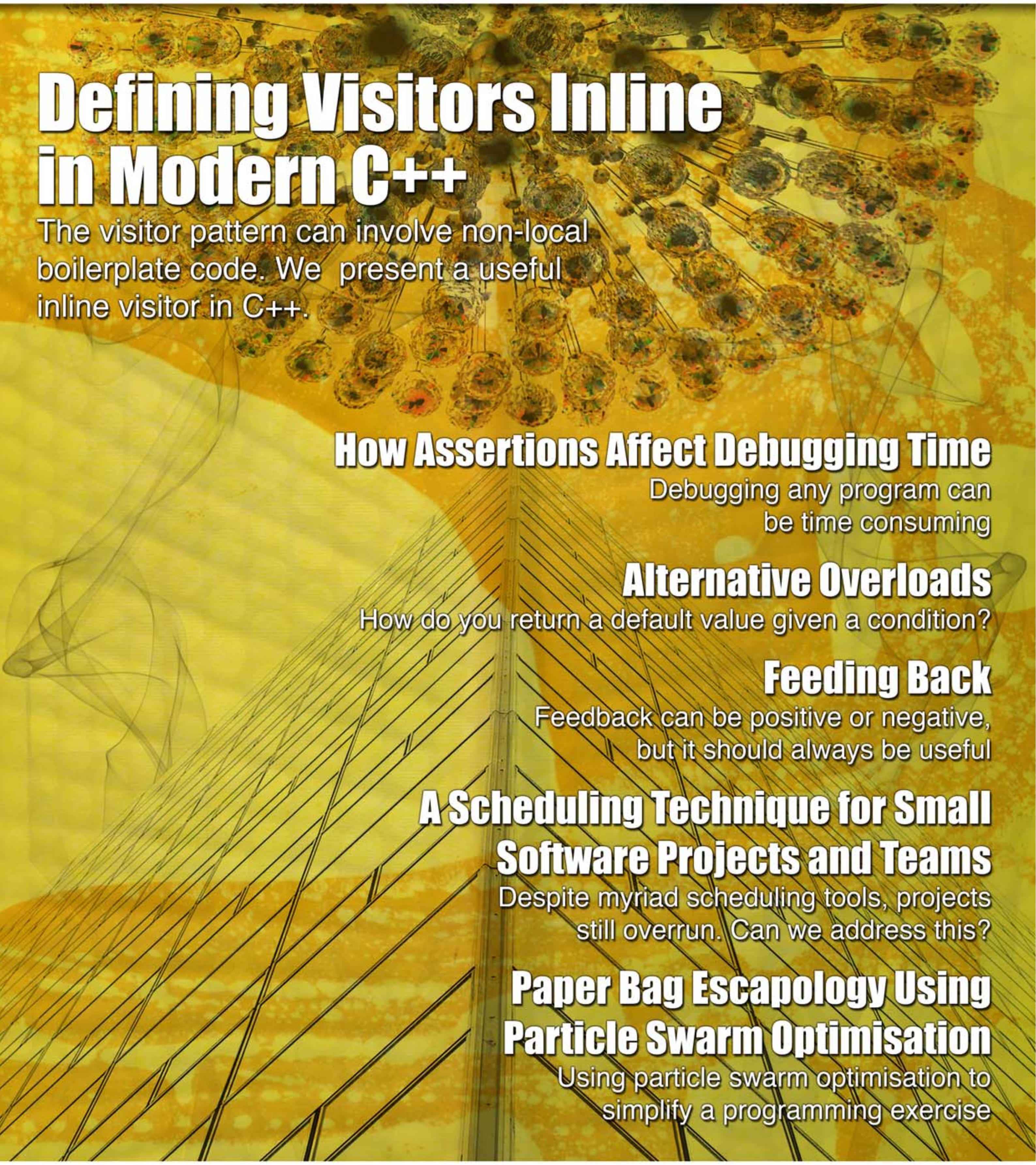
overload 123

OCTOBER 2014 £3



A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 123

October 2014

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 124 should be submitted by 1st November 2014 and those for Overload 125 by 1st January 2015.

The ACCU

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

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

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

Peer Reviewed

Nobody gets it right first time. Frances Buontempo considers the importance of the review team.

After being distracted by shopping last time, I have managed to refocus and believe I am getting closer to writing an editorial. Before I do, I would like to consider the other main duty of *Overload*'s editor – after gathering articles – making sure they are thoroughly reviewed before publication. Nowadays many people write blogs. You write the piece, and then people can argue with you in the comments after the fact. When finding a blog you frequently need to peruse the comments in order to get all the details. Something similar happens with question and answer websites like Stackoverflow. To get the full answer with all the caveats, one must read more than just the headline answer. In contrast, a peer reviewed composition will have been through a few initial drafts and the final version will be an opus incorporating all the comments from the reviewers, if the editor has done their job properly. If something does slip through the net, then letters to the editor are of course welcome, or further articles showing an alternative approach can be proffered.

This begs the questions, "Who are the reviewers?" and "What is their role?" In some sense, the reviewers act like a jury. Jury appears to stem from the Latin *jurare* meaning to swear, in the sense of a binding promise rather than a blasphemous stream of obscenities. The idea of a jury brings to mind the phrase "12 good men and true". The "good men and true" can certainly be found in Shakespeare's *Much Ado About Nothing*, though I believe woman have been allowed on juries since the 1920s in the UK and USA. *Overload* would welcome female reviewers – if any were to apply. I am not clear why or when courts decided they required 12 jurors. *Overload* currently has 7 reviewers, and from time to time seeks input from subject specialists as required. Some are more active than others – reviewers are in no way required to comment on every article, but many thanks to those who do dutifully attempt this. It seems possible that the use of twelve peers relates to Charlemagne's Twelve Peers in the old romances [Charlemagne], and the internet suggests that peer review was first recorded in the 1970s [Peers], of course, giving no suggestion 12 reviewers are required. The number 12 crops up frequently in various contexts – both with religious significance [12 tribes of Israel] and various measuring systems (for example, feet and inches, old coinage, hours in a day). I believe is often used because it breaks into many factors – 3 groups of 4 and so on. This might make it appropriate for coinage and groups of people requiring different divisions depending on context. Other examples for twelve people taking on significance include the *Magna Carta* [BL]:

> All evil customs relating to forests and warrens, foresters, warreners, sheriffs and their servants, or river-banks and their wardens, are at once to be investigated in every county by twelve sworn knights of the county

Whatever the evil customs were, it certainly makes sense that one person might not be sufficient to establish the validity of something. For example, Deuteronomy 19:15 says

> One witness is not enough to convict anyone accused of any crime or offense they may have committed. A matter must be established by the testimony of two or three witnesses.

Of course, I am not claiming that a submitted article is equivalent to a crime or offense. It is simply that more pairs of eyes, to a point, can spot errors and inconsistencies, making a final article more polished. Mind you, the Bible also says "Judge not, lest ye be judged." I digress. The important point is more than one person passing comment on a submission is more likely to cover all angles, including grammar, spelling, potential edge cases in any code presented, or all-out glaring bugs, omissions, alternatives that deserve consideration and so on. Though one can often find comments on the technical content of a blog, one frequently doesn't see suggestions on writing style or alternative wordings. This is one, if somewhat small, advantage *Overload*, as a peer reviewed journal, offers over just writing your own blog. Of course, *Overload* welcomes blog entries if the writer wishes them to be published here too, but be warned, they also go through the review process.

As programmers, many of us will be used to peer reviews in the form of code reviews, either as a formal or informal process with colleagues at work, or in order to get contributions into an open source project. This can be quite an emotional experience until you get used to it. Having someone say all the things they don't like about what you have done can be very deflating. Some submissions have caused a vast number of comments and nit-picks before finally being accepted. The same can happen in code reviews. I like to assume the amount of comments is proportional to the time people spent thinking about what you said. At least they listened, even if they missed your point. As a writer, or programmer, you will learn to take criticism, and it does become less personal the more feedback you get. However, good reviewers, be that of an article, or of code, should always say what they like as well as saying what they don't like. It is important to encourage the positive. Sometimes, places employ pair programming, and this can be considered to obviate the need for code reviews, though again the extra pair of eyes can help. An obvious difference between pair programming and a code review is the review will tend to take place once the code is complete, in the programmer's opinion, whereas the pairing approach means feedback is continuous and happens during the course of the work. *Overload* is quite willing to give feedback before a writer thinks the submission is polished and ready for a final review. Most articles do go back and forwards a few times. Mind you, so does code during a code review.

Having briefly looked at what a reviewer does, we should return to the question "Who are the reviewers?" Staying, for the moment, with the code review analogy, some organisations only allow senior people to review

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 15 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

code. Perhaps the team leader or senior architect has to approve any code changes. This begs the question of who reviews the reviewers' work. I personally am happy for less experienced or more junior people to review my code. They can still spot things I have missed, and the exchange can allow knowledge to pass in both directions. Perhaps I will be able to explain why I have taken a given approach. Perhaps they can tell me elements they don't understand. This is part of the vital team building process though and takes us away from the question of who the reviewers are. Other peer reviewed journals might choose experts in the field or throw out a periodic call for reviewers asking them to submit a review they have previously written, for example, a book review as proof of competence. As an ACCU member, you can volunteer for book reviews – just checkout the website, if this is something you would like to get good at. If you wish to volunteer to be a reviewer for *Overload*, then get in touch with me. You will not be asked for a sample of your previous work. You will simply get included on an email list and are welcome to provide feedback on any aspect of the submitted articles. We tend to keep the reviewers anonymous, though have sometimes named individuals when asked, for example if they have produced a stunning alternative code sample. Credit where credit's due.

Overload's review team does usually consistent of volunteers, though historically a jury would not necessarily have been composed of volunteers. It seems members were 'empanelled' by a sheriff in the thirteenth century [Musson97]. The eyre was a circuit travelled by the sheriff and his men in England. The 'justices' arrived unannounced at irregular intervals, forming a flash-mob review panel, of sorts. Of course, the justices' main role was raising funds for wars, rather than simply listing and reviewing the state of the eyre. As editor, I may attempt to empanel extra reviewers from time to time, if I feel we need someone with expertise in a subject not currently covered by the team, and do try to avoid wars. There has been little resistance to being 'volunteered' so far. If the jury of peers are not volunteers, in what sense are they peers? It seems the jury used to have to be nobles or high ranking, so nobles could not be judged by 'less important' people. References to the idea of peers or equals can be found in the *Magna Carta* [BL], for example

> Earls and barons shall be fined only by their equals

and

> To any man whom we have deprived or dispossessed of lands, castles, liberties, or rights, without the lawful judgement of his equals, we will at once restore these.

In a criminal court nowadays, one is supposed to be judged by peers or contemporaries though I suspect an Earl would not be allowed to insist that the panel of jurors consistent solely of Earls. Peer itself seems to be rooted in the word 'par' thereby tracing back to the idea of equal ranking. Ranking is of course, a relative term, and these history lessons remind us of the extremely hierarchical nature of groups of people in England in times gone by, though echoes still remain. I like to think every ACCU member is equally qualified to peer review articles, regardless of membership of the C++ committee, years of experience and so on. Everybody's input can be equally valuable.

Another reason for peer review of academic journals is to carefully validate any claims made. It is clearly important claimed advances in medicine are carefully checked and validated. A scientific journal will insist on a methods section, so that the results can be replicated. This is part of the essence of scientific discovery, though a brief study of the history of science will show a long journey to settle on this methodology. Even with a rigorous review process, things do slip through the net. There are many examples, including out and out lies, such as the falsification of data in stem cell research [Suk]. Such lies are usually eventually uncovered and in this case the journal, *Science*, editorially retracted the two papers by Hwang *et al*. Other peer reviewed articles that eventually get called into question have not been based on fabrication. For example, the measles, mumps and rubella vaccine controversy, which tried to link the jab to an increased risk of autism [MMR]. Hopefully, *Overload* has never published falsified data or caused public controversy.

Let us wrap up this review of the review process with an overview, loosely based on a guide to evaluating information sources [Lloyd Sealy]. First, the goal of peer review is to assess the quality of articles submitted for publication. This involves a, possibly iterative, process: An author submits to the editor who forwards the article to experts in the field, maybe after an initial read as a sanity check. The reviewers evaluate the submission, 'For accuracy and assess the validity of the research methodology and procedures.' The reviewers can, and often do, suggest revisions. In theory they can reject the article, though we aim for enough feedback to iron out any problems. Writing for a peer review journal, rather than self-publishing, will give you early feedback and potential guidance you will not get elsewhere. Referencing a peer reviewed article may stand your words on more solid ground than just surfing the web for quotes that match your thinking – I do realise the irony of my references being full of urls. If you feel inspired to submit an article or join the review team then contact Overload@accu.org.

References

[12 tribes of Israel] Genesis 49:28

[BL] A British Library online translation of the *Magna Carta*: http://www.bl.uk/treasures/magnacarta/translation/mc_trans.html

[Charlemagne] http://www.etymonline.com/index.php?term=peer

[Lloyd Sealy] http://guides.lib.jjay.cuny.edu/content.php?pid=209679&sid=1746812 (many thanks to Roger Orr for this link)

[MMR] http://www.bbc.co.uk/sn/tvradio/programmes/horizon/mmr_prog_summary.shtml

[Musson97] 'Twelve Good Men and True? The Character of Early Fourteenth-Century Juries' Anthony Musson *Law and History Review*, Vol. 15, No. 1 (Spring, 1997), pp. 115–144

[Peers] http://thesaurus.com/browse/peers

[Suk] See http://www.sciencemag.org/site/feature/misc/webfeat/hwang2005/

Debug Complexity: How Assertions Affect Debugging Time

Debugging any program can be time consuming. Sergey Ignatchenko and Dmytro Ivanchykhin extend their mathematical model to consider the effect of assertions.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translators and editors. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

In 'A Model for Debug Complexity' [Ignatchenko13], we started to build a mathematical model for estimating debugging efforts, and made some sanity checks of our model, in particular on relations between coupling and debug complexity. In this article, we have extended that model to see the effect of the assertions on debugging time. It should be noted that, as previously, the model should be considered to be very approximate, with several assumptions made about the nature of the code and debugging process (though we're doing our best to outline these assumptions explicitly). Nonetheless, the relations observed within the results obtained look quite reasonable and interesting, which makes us hope that the model we're working with represents a reasonable approximation of the real world.

Assumptions

1. In 'A Model for Debug Complexity' [Ignatchenko13], we considered purely linear code. However, it seems that in the context of debugging the same analysis applies to arbitrary code, as long as the execution path is fixed (which is usually the case for deterministic, repeatable debugging); in this case the execution path can be interpreted as linear code for the purposes of analysis. In this article, we'll use the term 'linear code', implying that it is also applicable to any execution path.
2. The linear code consists of (or the equivalent execution path goes through) N lines.
3. In a naive debugging model, the developer goes through the code line by line, and verifies that all the variables are correct. $\mathbf{T_{singlecheck}}$ denotes the time to check a single variable.
4. In the earlier article, we mentioned that in many cases it is possible to use a bisection-like optimization to reduce debugging time very significantly. However, such an optimization requires well-defined interfaces to be able to check the whole state of the program easily, and in such cases individual test cases can be easily built to debug an appropriate program part. For the purposes of this article, we will only consider a chunk of code which cannot easily be split into well-defined portions (in other words, a 'monolithic' chunk of code), and will not analyze it using bisection optimization.
5. Previously, it was mentioned that not all variables need to be analyzed due to coupling. For the purposes of this article, we'll use the term 'variables to be analyzed'; also we expect that for our analysis of chunks of code which cannot be easily split (see item 4 above), the chances of tight coupling are rather high, so the difference between 'variables', 'variables to be analyzed', and 'coupled variables' is not expected to be significant enough to substantially affect the relations observed within our findings.
6. We assume that the number of variables to be changed grows from the beginning to the end of the code; to simplify modeling we also usually assume that this growth is linear.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko and Dmytro Ivanchykhin using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

Dmytro Ivanchykhin has 10+ years of development experience, and has a strong mathematical background (in the past, he taught maths at NDSU in the United States). Dmytro can be contacted at d_ivanchykhin@yahoo.com

Notation

N	The total number of lines in a 'monolithic' code chunk
x	The current line
$v(x)$	The number of variables to consider at line x; $v(x) \sim k*x$ for some k (see *Assumption #6* above)
$w(x)$	The amount of work spent on a single line to analyze variables; as discussed in 'A Model for Debug Complexity' [Ignatchenko13] $w(x) = 2^{v(x)} = 2^{k*x}$
$W(N)$	The total amount of work for debugging a single bug in a code of length N; as it was shown in 'A Model for Debug Complexity':

$$W(N) = \sum_{x=1}^{N} w(x) = \sum_{x=1}^{N} 2^{k*x}$$

This sum may be estimated converting to integrals:

$$\begin{aligned} W(N) &= \int_{x=0}^{N} 2^{k*x}dx \\ &= \int_{x=0}^{N} e^{\ln 2*k*x}dx \\ &= \frac{1}{k*\ln 2}(e^{\ln 2*k*N}-1) \\ &= \frac{1}{k*\ln 2}(2^{k*N}-1) \end{aligned}$$

which is, if N is large enough is

$$O(2^{k*N}) = k_1 * 2^{k*N}$$

with some coefficient k_1. That is, it grows exponentially with the length of code (NB: we do not consider bisection-like optimization, see *Assumption #4* above).

simple assertions, say of an array index being within the array boundaries, are extremely rewarding in terms of helping to reduce debugging times

7. In the earlier article, an obvious optimization – that after the bug is found, the process of going through the code line by line can be stopped – wasn't taken into account. However, we feel that it doesn't substantially change relations observed within our findings, and as taking it into account will complicate the mathematics significantly, we'll leave such analysis for the future.
8. Our analysis is language-independent. That is, all language-specific effects such as 'in C/C++ you can easily write an assert such as `assert(a=b)` which will cause bugs', are out of scope. Also, we'll use the term `ASSERT` for assertions in any programming language.

Introducing ASSERTs

Now assume that there is an `ASSERT` that catches the bug, defining 'an `ASSERT` that catches the bug *X*' as 'an `ASSERT` which fails if bug *X* is present'.

If the `ASSERT` *A* is at line x_A, then it remains to debug only the first x_A lines, and the amount of work required will be

$$W(x_A) = O(2^{k*x_A})$$

It should be pointed out that the ratio of the amount of work without this `ASSERT` *A*, compared to that with it, will be

$$O(2^{k*(N-x_A)})$$

This suggests, in particular, that an `ASSERT` in the middle of code can save far more than 50% of the work.

For instance, in a one-thousand line code chunk with 10 variables to be analyzed at the end (that is, *N*=1000, and *k*=0.01) the total amount of work without asserts may be of the order of

$$\frac{1}{0.01*\ln 2}(2^{0.01*1000}-1) \approx 144.27*1023 \approx 147588 T_{singlecheck^S}$$

And with the `ASSERT` in the middle of the code this value will become

$$\frac{1}{0.01*\ln 2}(2^{0.01*500}-1) \approx 144.27*31 \approx 4472 T_{singlecheck^S}$$

which is 33 times less!

In practice, an `ASSERT` may catch a bug with some probability: one may assume that checking a certain condition in the `ASSERT` will catch the bug, but indeed, may not. Let's say that an `ASSERT` has a probability p_A of catching a bug. Then, the expectation of the amount of work may be estimated as a sum of

$$p_A * k_1 * 2^{k*x_A} + (1-p_A)*k_1*2^{k*N}$$

where the first term is for the case when the `ASSERT` is successful, and the second term is for the opposite case.

Quality of ASSERTs

Clearly, the greater the probability of an `ASSERT` catching the bug, the less debugging work has to be done. But is it true that an `ASSERT` with probability of catching a bug of, say, 0.3 is only twice as bad than that with probability 0.6? If two `ASSERT`s are independent and have a probability of catching a bug of 0.3, then the probability that the bug won't be caught by either of them is $(1\text{-}0.3)^2 = 0.49$. Let's use the above example, and assume that all `ASSERT`s sit in the middle of the code. Substituting, we may get for the `ASSERT` with probability 0.6:

$$\begin{aligned} W &= 0.6\frac{1}{0.01*\ln 2}(2^{0.01*500}-1)+(1-0.6)\frac{1}{0.01*\ln 2}(2^{0.01*1000}-1) \\ &= 0.6*4472+0.4*147588 \\ &= 61718 T_{singlecheck^S} \end{aligned}$$

And for two `ASSERT`s with probability 0.3 each:

$$\begin{aligned} W &= 0.51\frac{1}{0.01*\ln 2}(2^{0.01*500}-1)+0.49\frac{1}{0.01*\ln 2}(2^{0.01*1000}-1) \\ &= 0.51*4472+0.49*147588 \\ &= 74599 T_{singlecheck^S} \end{aligned}$$

Let's define an 'assert which has a high probability of catching probable bugs' as a 'high-quality assert'. Unfortunately, there seems to be no simple recipe on 'how to write high-quality asserts', though one consideration may potentially help: if an assert aims to catch one of the most common bugs it has quite a good chance of being a 'high-quality assert'. In particular, 'No Bugs' has observed that, when coding in C/C++, simple assertions, say of an array index being within the array boundaries, are extremely rewarding in terms of helping to reduce debugging times – simply because it is very easy to go beyond allocated memory, and it is very difficult to find the exact place where this has happened. Another potential suggestion is related to using asserts as a way of (enforced at runtime) documenting code [Wikipedia]; such 'code documenting' asserts (in the experience of 'No Bugs') tend to catch more subtle logical bugs.

Multiple bugs

In general, `ASSERT` *A* may catch more than a single bug, and we may talk about the probability $p_A{}^B$ of the `ASSERT` *A* catching a specific bug *B* residing in the code. Thus, if there are *n* bugs, then `ASSERT` *A* may have probabilities $p_A{}^{Bi}$ of catching bug *Bi* for each *i* from 1 to *n*. With this assumption, for instance, the probability that `ASSERT` *A* is useless is the product:

$$\prod_{i=1}^{n}(1-p_A^{Bi})$$

We may call this product the value ineffectiveness of an `ASSERT`. It is clear to see that if, with time, some bugs are caught and therefore the number of bugs decreases, the value ineffectiveness of `ASSERT`s tend to increase.

catching 'the last bug' will usually require far more work than the first one

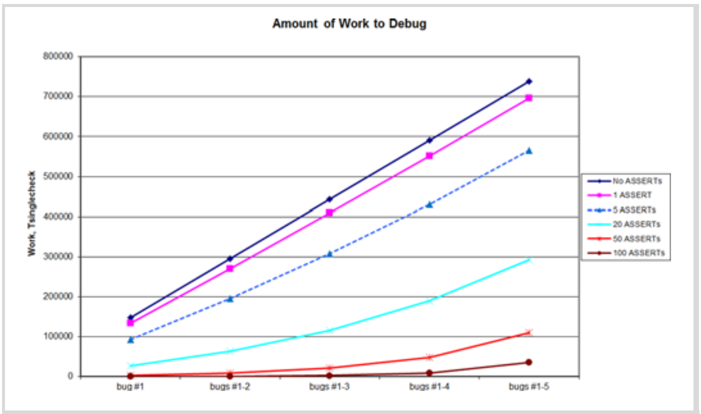


Figure 1

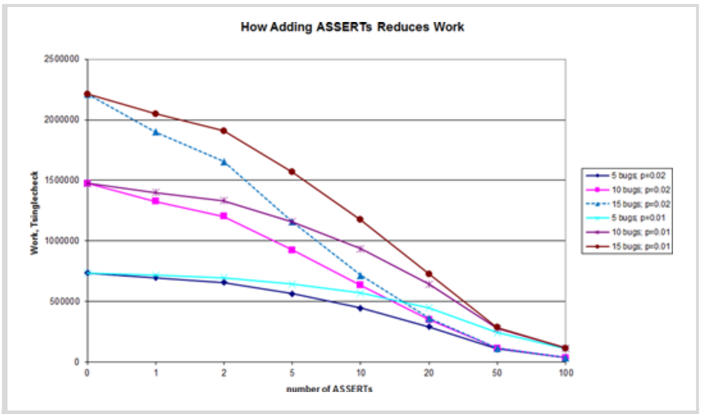


Figure 2

Let's denote it by *IL*. The complimentary probability, 1-*IL*, gives a chance that the **`ASSERT`** catches at least one bug.

Multiple bugs – multiple ASSERTs

In a real program, there is often (alas!) more than a single bug, and it is (luckily!) possible to place more than a single **`ASSERT`**. Then the amount of work to catch a single bug in a code with *n* bugs and *m* **`ASSERT`**s placed at lines x_i, respectively, may be estimated (assuming for simplicity that **`ASSERT`**s are enumerated in the order of lines they are placed at) as:

$$\begin{aligned} W &= (1-IL_{A1})*W(x_{A1})+IL_{A1}*((1-IL_{A2})*W(x_{A2}) \\ &\quad +IL_{A2}*(...((1-IL_{Am})*W(x_{Am})+IL_{Am}*W(N))...)) \quad (*) \end{aligned}$$

For instance, in the above example with three **`ASSERT`**s at lines 250, 500, and 750, respectively, and values of ineffectiveness of 0.5 each, to catch a single bug the amount of work will be:

$$\begin{aligned} W &= 0.5\frac{1}{0.01*\ln 2}(2^{0.01*250}-1) \\ &\quad +0.5*(0.5*\frac{1}{0.01*\ln 2}(2^{0.01*500}-1) \\ &\quad +0.5*(\frac{1}{0.01*\ln 2}(2^{0.01*750}-1) \\ &\quad +0.5*\frac{1}{0.01*\ln 2}(2^{0.01*1000}-1))) \\ &\approx 0.5*672+0.5*(0.5*4472+0.5*(0.5*25971+0.5*147588)) \\ &= 23149T_{singlecheck^S} \end{aligned}$$

which is more than 6 times less than without any **`ASSERT`**s.

To illustrate the effect of using asserts from slightly different point of view, for simplicity we may make another assumption that for any **`ASSERT`** the probability *p* of catching any specific bug is the same:

$$\forall i, A: p_A^{Bi} = p$$

Then in the above notation, the value of ineffectiveness *IL* may be written as a function of number of remaining *k* bugs:

$$IL(k) = (1-p)^k$$

Then, using (*) above, we may calculate the work for finding a bug when only *k* bugs remain:

$$\begin{aligned} W(k) &= (1-IL_{A1}(k))*W(x_{A1}) \\ &\quad +IL_{A1}(k)*((1-IL_{A2}(k))*W(x_{A2}) \\ &\quad +IL_{A2}(k)*(...((1-IL_{Am}(k))*W(x_{Am})*\mathrm{W(N)})...)) \quad (**) \end{aligned}$$

Adding up the amounts of work *W*(*k*) for each *k* from *n* to 1 will give us a total expected amount of work to debug all *n* bugs:

$$W = \sum_{k=n}^{1} W(\mathrm{k})$$

To get some taste of what these formulae mean, we have calculated a few samples based on the example that we considered above: a chunk of 1000 lines of 'monolithic' code, a linear increase of the number of variables to be analyzed along the code from 1 to 10, 5 bugs, and certain number of **`ASSERT`**s with a bit more realistic probability of catching a bug of 0.02; the resulting graph of 'cumulative amount of work as debug progresses through finding bugs' is shown on Figure 1.

In particular, this graph illustrates that, as we have mentioned above, the **`ASSERT`** effectiveness tends to 'degrade' as debugging goes ahead. This finding is consistent with what we observe in practice, where catching 'the last bug' will usually require far more work than the first one. One way that is derived from practical experience, and which follows from the above reasoning, is to add **`ASSERT`**s… or to follow a good habit of using them in any place where conditions may be in doubt whilst coding.

Another example of calculation is shown on Figure 2 and illustrates how increasing the number of **`ASSERT`**s helps to reduce amount of work necessary to debug the program (note that for presentation purposes, the number of **`ASSERT`**s on the graph is near-logarithmic).

Note that while the nature of our analysis is very approximate, the relations observed within our results are expected to be reasonably close to reality; that is, while real-world debugging time can easily differ from the results calculated using our formulae, reduction of the real-world debugging time as number of **`ASSERT`**s increases, should be reasonably close to those calculated and shown on the graphs.

Conclusion

Good is better than bad,
Happy is better than sad,
My advice is just be nice,
Good is better than bad

~ Pink Dinosaur from *Garfield and Friends*

Within the debug complexity model previously introduced [II2013], we have analyzed the impact of asserts on debugging time. Our results seem to be quite consistent with debugging practice:

- **`ASSERT`**s can reduce debugging time dramatically (making it several times less)
- debugging-wise, there are 'high-quality asserts' and 'not-so-high-quality asserts'
- purely empirical suggestions for 'high-quality asserts' were given in the 'Quality of ASSERTs' section
- the time for debugging 'the last bug' is significantly higher than the time for debugging the first one.

In addition, it should be noted that the impact of **`ASSERT`**s on the program is not limited to a reduction in debugging time. As such effects are well beyond the scope of this paper, we'll just mention a few of them very briefly. On the negative side: depending on the programming language (and especially for the languages where an **`ASSERT`** is a mere function/macro, such as C/C++) it may be possible to write an **`ASSERT`** which changes the state of the program (see also *Assumption #8* above). On the positive side, **`ASSERT`**s can be used to create documentation of the program, where such documentation (unlike, say, comments) cannot become out of date easily.

Overall, 'No Bugs' highly recommends the using of **`ASSERT`**s, though feels that creating any kind of metrics such as 'number of **`ASSERT`**s per 1000 lines of code', as a result of Goodhart's Law [Goodhart], will become as useless as 'number of comments per 1000 lines of code'. As **`ASSERT(1==1)`** is as useless as it gets, it is certainly not about sheer numbers, so it is important to use high-quality **`ASSERT`**s. This still seems to be an art rather than science, though a few hints for 'high-quality **`ASSERT`**s' were provided above, and most likely there are many more other such hints in existence. ■

References

[Goodhart] http://en.wikipedia.org/wiki/Goodhart%27s_law

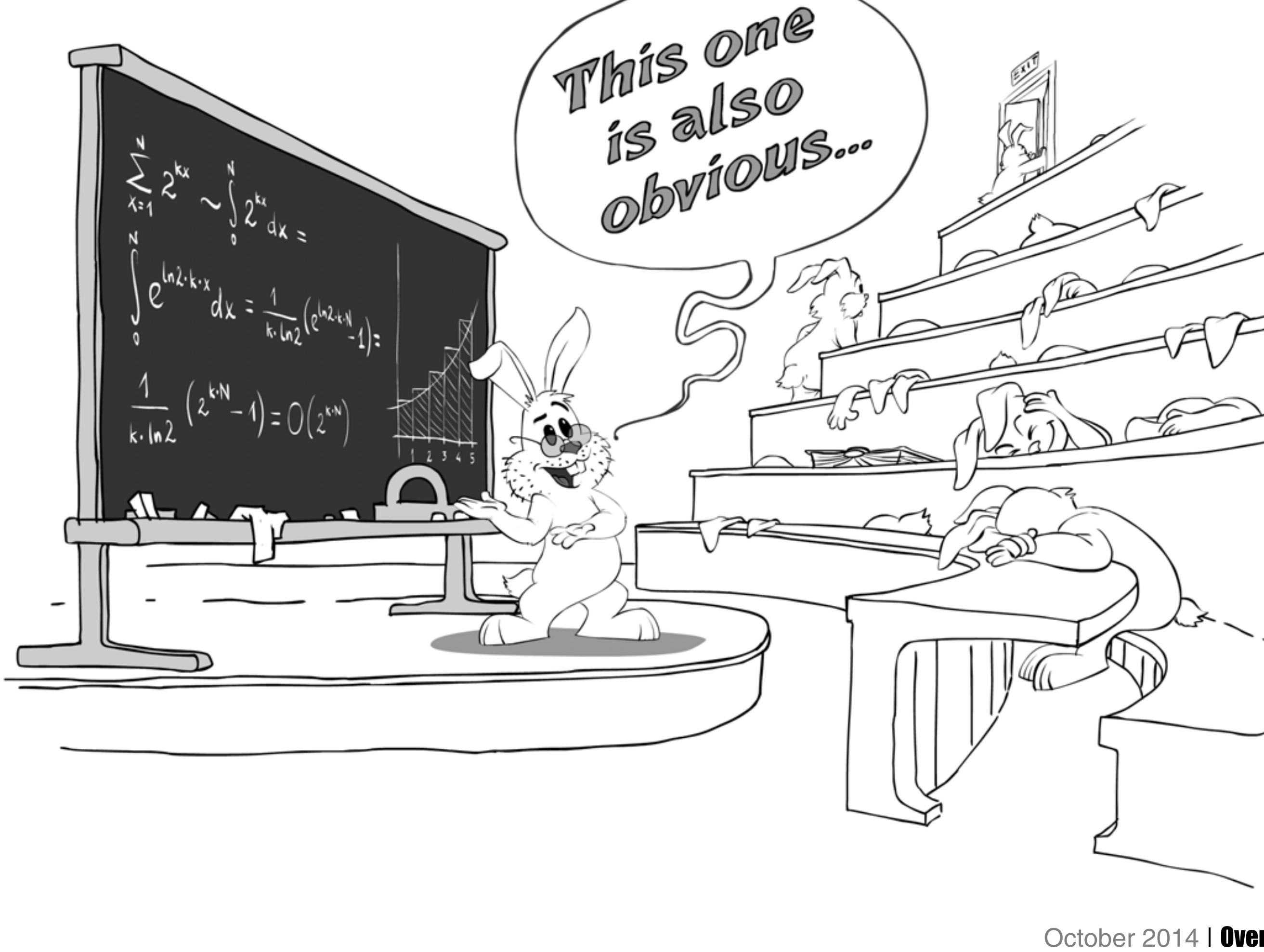
[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Ignatchenko13] 'A Model for Debug Complexity', Sergey Ignatchenko and Dmytro Ivanchykhin, *Overload* 114, April 2013

[WikiAssertion] http://en.wikipedia.org/wiki/Assertion_(software_development)#Assertions_in_design_by_contract:
Assertions can function as a form of documentation: they can describe the state the code expects to find before it runs (its preconditions), and the state the code expects to result in when it is finished running (postconditions); they can also specify invariants of a class.

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.


This one is also obvious...
$\sum_{x=1}^{N} 2^{kx} \sim \int_0^N 2^{kx} dx =$
$\int_0^N e^{\ln 2 \cdot k \cdot x} dx = \frac{1}{k \cdot \ln 2}(e^{\ln 2 \cdot k \cdot N} - 1) =$
$\frac{1}{k \cdot \ln 2}(2^{k \cdot N} - 1) = O(2^{k \cdot N})$
1 2 3 4 5
EXIT

Alternative Overloads

How do you return a default value given a condition? Malcolm Noyes presents solutions using older and newer C++ techniques.

Recently I came across a blog post by Andrzej Krzemieński [Krzemieński] outlining how to apply an overload so that the behaviour of a function could depend on the type passed to it; if the type passed was convertible to the value type of the containing object the function could return the type passed, otherwise it could attempt to use a function object to return the value.

The blog post demonstrates two solutions for existing C++11 compilers and also presents a possible solution if/when 'Concepts Lite' [Sutton13] makes it into the standard; his post explains this in some detail so I won't repeat here what he has already said.

When I read it, I wondered whether it might be possible to solve the problem the other way around, in other words if the passed type was a function object whose function call operator returned a suitable type, then the function should use that, otherwise it should assume the type was convertible and attempt to return the value (with a compile error if that didn't work). Although the C++11 solutions that Andrzej presented would also work with C++98/03 and Boost type traits, I thought it might be possible to build a solution that used only features available in C++98/03 (some overloads with a little bit of type matching). It turns out that we can get quite close to a solution in C++98/03 with some constraints on the function call operator.

During the review of this article for *Overload*, Jonathan Wakely showed how easily the same thing could be done in C++11; it seems that this solution also works with several versions of Visual C++ (at least back as far as VS2010), so I'll show his code at the end.

The problem

The underlying problem identified is fairly simple; allow a function to return a 'default value' if some condition is met; in the example in Listing 1 if the 'optional' class hasn't had an initialised value then return the default (this example is taken from Andrzej's original blog, slightly simplified).

```
#define REQUIRE(x) std::cout << #x << std::endl;
assert(x);
template <typename T>
class optional
{
  // ...
  template <typename U>
  T value_or(U const& v) const
  {
    // condition changed "if(*this)" in Andrzej's
   // example since people found that confusing...
    if (m_initialized)
      // get contained value
      return this->value();
    else
      // get v converted to T
      return v;
  }
};
...
optional<int> v1(20);
// value was initialised, return it...
REQUIRE( v1.value_or(42) == 20 );

// value was not initialised, return default...
optional<int> v2;
REQUIRE( v2.value_or(42) == 42 );
```

Listing 1

Andrzej then extends the problem such that the `value_or()` function can also accept a type that could be 'callable', so it might be either a function pointer or a functor [Wikipedia]. He presents two solutions for C++11; one using `std::is_convertible` to provide tag dispatching [Boost-a] or `std::enable_if` [Boost-b] to enable/remove functions from the overload set. It is this additional requirement that I want to look at...

C++98/03 overloads – calling with a function pointer

To call with a function pointer we can simply provide a function overload that matches a function pointer. This will match any free/static function that takes no parameters (so will give a compile error if the return type is not convertible to the value type of the containing object). Note that a class with a conversion operator also 'just works'... (see Listing 2).

Malcolm Noyes has worked as a software developer/author for several years; just how many can be deduced from the information that he started programming C++ using a Zortech compiler. He wrote several string classes before discovering the STL and several thread classes before multi-threading got standardised. He has never written a Unit Test framework but probably would have done if Phil Nash hadn't got there first!

Calling with a function object

To allow calling with a function object, we can provide an additional function with two overloads that will either use a function object or return the value. This is a variation of a commonly used idiom used by tag dispatching but in our case instead of creating a 'tag' we will allow the compiler to match a function overload if the supplied object has a function call operator (i.e. `operator()`).

To see how this will work, consider the more general case of a function that accepts the supplied value as the first argument and 'anything' as the second (see Listing 3).

This works because the ellipsis (`...`) matches anything and consequently the function `functor_or_default` will match the supplied arguments, (`v` and `0`).

Fortunately, the compiler considers ellipsis to be the worst possible match so now all we need to do is provide an overload that is a better match than ellipsis if the type is a function object with a matching function call operator. Although our goal is to match the function call operator but we

```
// function to be called...
int function() { return -1; }
// struct with conversion operator doesn't
// compile...
struct conversion
{
  operator double() const
  {
    return 13.0;
  }
};
template <typename T>
class optional
{
  // ...
  // overload for free/static functions
  template <typename U>
  T value_or(U (*fn)()) const
  {
    if (m_initialized)
      // get contained value
      return this->value();
    else
      // call function..
      return fn();
  }
  // default for all other types, as before...
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      return v;
  }
};
...
  optional<double> v2;
  // as before...
  REQUIRE( v2.value_or(42) == 42 );

  // calls passed function...
  REQUIRE( v2.value_or(&function) == -1 );

  conversion conv;
  // fine, conversion just works...
  REQUIRE( v2.value_or(conv) == 13.0 );
```

Listing 2

will start with something a little simpler to show how this works. For example, Listing 4 shows the code if we wanted to match **`int`**.

In this case the first overload will be called since we passed zero (an **`int`** with **`value == 0`**) as the second argument to **`functor_or_default`**.

Now all we need to do is replace the overload taking an **`int`** with one that matches the function call operator. We'll go in two steps; first, let's imagine that the passed object has a function called **`default_value`** (this makes the syntax slightly more readable...). Ideally, we would like to provide a version of **`functor_or_default`** that matched a pointer to the member function, so we could replace **`functor_or_default`** with something like Listing 5.

Whilst this overload matches any class type and works for our functor, the body of the function now fails to compile for a class without a **`default_value`** function so our class with a conversion operator no longer compiles; what we need is something more specific for the compiler to match so we provide a helper that gives another level of indirection (Listing 6).

The **`has_functor`** helper allows us to declare a type that matches only if it has a matching function pointer; if the member function does not exist

```
template <typename T>
class optional
{
  // ...other functions as before...

  // ellipsis matches everything...
  template <typename U>
  static T functor_or_default(const U& v, ...)
  {
    return v;
  }
  // default for all other types
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      // get v converted to T or functor...
      return functor_or_default<U>(v, 0);
  }
};
...
  optional<double> v2;
  // still works but now calls
  // 'functor_or_default'...
  REQUIRE( v2.value_or(42) == 42 );
```

Listing 3

then the template overload is invalid and the compiler removes it from the candidate list of functions; this is known as 'substitution failure is not an error' [SFINAE]. This is a standard trick often used in tag dispatching, where it is usually passed to **`sizeof()`** but here we're using it directly as a parameter to the overload.

```
template <typename T>
class optional
{
  public:
  // ...
  // overload called for 'int'...
  template <typename U>
  static T functor_or_default(const U& v, int)
  {
    return v;
  }
  // ellipsis still matches everything else...
  template <typename U>
  static T functor_or_default(const U& v, ...)
  {
    return v;
  }
  // default for all other types
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      // pass 'int' with value 0
      return functor_or_default<U>(v, 0);
  }   // ...
};
...
optional<double> v2;
 // as before...
REQUIRE( v2.value_or(42) == 42 );
```

Listing 4

```
// works...
struct functor
{
  double default_value() const
  {
    return 3.142;
  }
};
// conversion struct as before...
template <typename T>
class optional
{
  public:
  // ...
  // replace call with 'int' with function call ptr
  template <typename U>
  static T functor_or_default(const U& v,
                              T (U::*)() const)
  {
    return v.default_value();
  }
  // as before...
  template <typename U>
  static T functor_or_default(const U& v, ...)
  {
    return v;
  }
  // default for all other types
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      return functor_or_default<U>(v, 0);
  }
  // ...
};
...
optional<double> v2;
functor fn;
// fine, fn has 'default_value()'
REQUIRE( v2.value_or(fn) == 3.142 );
// fine, calls ellipsis overload as before
REQUIRE( v2.value_or(42) == 42 );

conversion conv;
// doesn't compile...'conversion' doesn't have
// 'default_value' function
REQUIRE( v2.value_or(conv) == 13.0 );
```

Listing 5

Now that we have this overload, we just need to fix up the syntax for calling a function call operator instead of a function named **`default_value`**...so the final version looks like Listing 7.

Fixing the problems using C++11

If we pass a pointer to a free/static function then the compiler will attempt to convert the return type for us. Unfortunately the type matching helper for the function object requires an exact match for the function call operator – both the return type and any const/volatile qualifiers must be the same or the pointer won't match and the 'anything' overload gets selected instead.

We would like to be able to match a function call operator that returns something convertible to the value type of 'optional' type instead of having a match for a specific function call operator and in C++11 we can do that with **`decltype`** (Listing 8).

If I've understood this correctly, if type **`U`** has a function call operator with a return type that is convertible (via **`static_cast<>()`**) to type **`T`** then the return type of this function overload of **`functor_or_default`** will be valid; the function will be part of the overload set and the second parameter (**`int`**) will be a better match than ellipsis (**`...`**). Note that we can also remove the overload that takes a pointer to a free/static function as this overload handles both cases.

It turns out that this also works with many versions of Visual C++ even though their C++11 support is limited. So the final solution for C++11 (or VS2010 or later) looks like Listing 9.

Jonathan also pointed out that although it is rarely useful, it can also successfully match things that aren't quite function objects (for example, see Listing 10).

```
  ...
  template <typename U, T (U::*)() const>
    struct has_functor {};
  // now a specific match...
  template <typename U>
  static T functor_or_default(const U& v,
     has_functor<U, &U::default_value>*)
  {
    return v.default_value();
  }
  ...
conversion conv;
// now fine, has_functor can't match
// 'default_value' for conversion type
// so overload removed from candidate functions...
// ...calls function with ellipsis and operator
// double()
REQUIRE( v2.value_or(conv) == 13.0 );
```

Listing 6

```
// as before...
struct functor
{
  double operator ()() const { return 3.142; }
};
struct conversion
{
  operator double() const { return 13.0; }
};
int function() { return -1; }
template <typename T>
class optional
{
 public:
  optional()
  : m_initialized(false)
  {}
  explicit optional(const T& v)
  : m_initialized(true)
  , t(v)
  {}
  template <typename U, T (U::*)() const>
     struct has_functor {};
  template <typename U> static
  T functor_or_default(const U& v,
     has_functor<U, &U::operator()>*)
  {
    return v();
  }
  template <typename U>
  static T functor_or_default(const U& v, ...)
  {
    return v;
  }
```

Listing 7

```
  // overload for free/static functions
  template <typename U>
  T value_or(U (*fn)()) const
  {
    if (m_initialized)
      return this->value();
    else
      return fn();
  }
  // default for all other types
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      return functor_or_default<U>(v, 0);
  }
  T value() const { return t; }
 private:
  bool m_initialized;
  T t;
};
...
  optional<double> v2;
  // calls passed function...
  REQUIRE( v2.value_or(&function) == -1 );
  functor fn;
  // fine, fn has function call operator
  REQUIRE( v2.value_or(fn) == 3.142 );
  // fine, calls ellipsis overload
  REQUIRE( v2.value_or(42) == 42 );
  conversion conv;
  // fine, calls ellipsis overload and
  // operator double()
  REQUIRE( v2.value_or(conv) == 13.0 );
```

Listing 7 (cont'd)

Wrap up

The solution presented by Andrzej answers the question 'How can we select an overload for a type that is convertible?' In this article I've tackled the problem from the opposite direction, i.e. 'How can we select an overload that matches something passed that looks like a function?'

We've seen that this can be done in C++98/03 with no additional requirements from the standard library (or boost) but we do need to be very specific about exactly what function call operator will match. The C++11 version is very neat (thanks Jonathan!) and has the added bonus of working with many versions of Visual C++.

I know many organisations are still limited to using C++98/03 and I hope that this article has shown that alternative techniques are often possible even for older compilers. ■

```
template <typename T>
class optional
{
  ...
  template <typename U> static
     auto functor_or_default (const U& v, int) ->
     decltype(static_cast<T>(v()))
  {
    return v();
  }
  ...
};
```

Listing 8

```
template <typename T>
class optional
{
 public:
  optional()
  : m_initialized(false)
  {}
  explicit optional(const T& v)
  : m_initialized(true)
  , t(v)
  {}
  template <typename U> static auto
     functor_or_default(const U& v, int) ->
     decltype(static_cast<T>(v()))
  {
    return v();
  }
  template <typename U> static T
     functor_or_default(const U& v, ...)
  {
    return v;
  }
  // default for all other types
  template <typename U>
  T value_or(U const& v) const
  {
    if (m_initialized)
      return this->value();
    else
      return functor_or_default<U>(v, 0);
  }
  T value() const { return t; }
 private:
  bool m_initialized;
  T t;
};
```

Listing 9

References

[Boost-a] Generic Programming Techniques: http://www.boost.org/community/generic_programming.html#tag_dispatching

[Boost-b] Boost.EnableIf (Jaakko Järvi, Jeremiah Willcock, Andrew Lumsdaine, Matt Calabrese) : http://www.boost.org/doc/libs/1_55_0/libs/utility/enable_if.html

[Krzemieński] Clever Overloading: Andrzej's C++ blog http://akrzemi1.wordpress.com/2014/06/26/clever-overloading

[SFINAE] Substitution Failure Is Not An Error (SFINAE): http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/SFINAE

[Sutton13] 'Concepts Lite: Constraining Templates with Predicates' Andrew Sutton, Bjarne Stroustrup, Gabriel Dos Reis at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3580.pdf

[Wikipedia] Function object: http://en.wikipedia.org/wiki/Function_object

```
struct not_quite_functor
{
  using func = int(*)();
  operator func() const
  {
    return [] { return 1; };
  }
};
```

Listing 10

Everyone Hates build.xml

Using the Ant build tool can be tricky. Andy Balaam shows how to structure and test the build code.

If you're starting a new Java project, I'd suggest considering the many alternatives to Ant, including Gant [Gant], Gradle [Gradle], SCons [SCons] and, of course, Make [Make]. This article covers how to bend Ant to work like a programming language, so you can write good code in it, and how to test that code.

It's seriously worth considering a build tool that makes structured programming easier, but if you've chosen Ant, or you're stuck with Ant, read on.

Most projects I've been involved with that use Ant have a hateful `build.xml` surrounded by fear. Many projects' build files grow to enormous sizes, for example becoming responsible for deployment, system test execution, notifications and many other jobs.

The most important reason for the fear is that the functionality of the build file is not properly tested, so you never know whether you've broken it, meaning you never make 'non-essential' changes: changes that make it easier to use or read.

But, before we can write tests, we must address a pre-requisite:

Can you write good code in Ant, even if you aren't paralysed by fear?

Everyone hates build.xml (code reuse in Ant)

One of the most important aspects of good code is that you only need to express each concept once. Or, to put it another way, you can re-use code.

I want to share with you some of the things I have discovered recently about Ant, and how you should (and should not) re-use code.

But first:

What is Ant?

Ant is 2 languages:

- A declarative language to describe dependencies
- A procedural language to proscribe actions

In fact, it's just like a Makefile (ignore this if Makefiles aren't familiar). A Makefile rule consists of a target (the name before the colon) with its dependencies (the names after the colon), which make up a declarative description of the dependencies, and the commands (the things indented by tabs) which are a normal procedural description of what to do to build that target.

```
# Ignore this if you don't care about Makefiles!
target: dep1 dep2    # Declarative
    action1          # Procedural
    action2
```

Andy Balaam is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

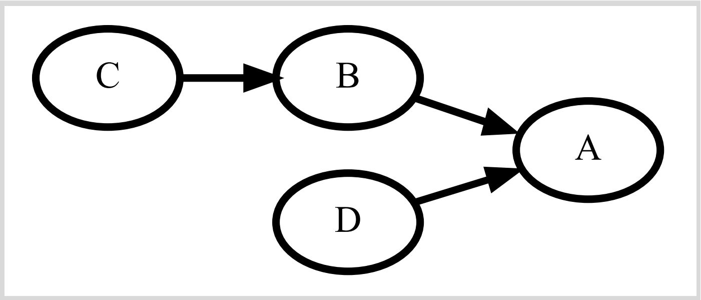


Figure 1

The declarative language

In Ant, the declarative language is a directed graph of targets and dependencies, shown graphically in Figure 1:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="A"/>
```

This language describes a directed graph of dependencies. I.e. they say what depends on what, or what must be built before you can build something else. Targets and dependencies are completely separate from what lives inside them, which are tasks.

The procedural language

The procedural language is a list of tasks:

```
<target ...>
    <javac ...>
    <copy ...>
    <zip ...>
    <junit ...>
</target>
```

When the dependency mechanism has decided a target will be executed, its tasks are executed one by one in order, just like in a programming language. Except that tasks live inside targets, they are completely separate from them. Essentially each target has a little program inside it consisting of tasks, and these tasks are a conventional programming language, nothing special (except for the lack of basic looping and branching constructs).

I'm sorry if the above is glaringly obvious to you, but it only recently became clear to me, and it helped me a lot when thinking about how to improve my Ant files.

Avoiding repeated code

Imagine you have two similar Ant targets (see Listing 1).

The **classpath** and **debug** information are the same in both targets, and we would like to write this information in one single place. Imagine with

Any dependencies of the compile target will be run even if they've already been run, meaning some of your assumptions about order of running could be incorrect

```
<target name="A">
  <javac
    srcdir="a/src" destdir="a/bin"
    classpath="myutil.jar" debug="false"
  />
</target>

<target name="B">
  <javac
    srcdir="b/code" destdir="b/int"
    classpath="myutil.jar" debug="false"
  />
</target>
```

Listing 1

me that the code we want to share is too complex for it to be possible to store it as the values of properties in some properties file.

How do we share this code?

The wrong way: antcall

Listing 2 shows the solution we were using in my project until I discovered the right way.

Here we put the shared code into a target called `compile`, which makes use of properties to access the varying information (or the parameters, if we think of this as a function). The targets **A** and **B** use the `<antcall>` task to launch the compile target, setting the values of the relevant properties.

This works, so why is it wrong?

```
<target name="compile">
  <javac
    srcdir="${srcdir}" destdir="${destdir}"
    classpath="myutil.jar" debug="false"
  />
</target>

<target name="A">
  <antcall target="compile">
    <param name="srcdir" value="a/src"/>
    <param name="destdir" value="a/bin"/>
  </antcall>
</target>

<target name="B">
  <antcall target="compile">
  ...
```

Listing 2

```
// ... imports here ...
public class MyCompile extends Task {
  String srcdir;
  public void setSrcDir(String s) {
    srcdir = s;
  }
  String destdir;
  public void setDestDir(String d) {
    destdir = d;
  }
  public void execute() throws BuildException
  {
    Project p = getProject();
    Javac javac = new Javac();
    javac.setSrcdir( new Path( p, srcdir ) );
    javac.setDestdir( new File( destdir ) );
    javac.setClasspath( new Path( p,
                             "myutil.jar" ) );
    javac.setDebug( false );
    javac.execute();
  }
}
```

Listing 3

Why not antcall?

`antcall` launches a whole new Ant process and runs the supplied target within that. This is wrong because it subverts the way Ant is supposed to work. The new process will re-calculate all the dependencies in the project (even if our target doesn't depend on anything) which could be slow. Any dependencies of the compile target will be run even if they've already been run, meaning some of your assumptions about order of running could be incorrect, and the assumption that each target will only run once will be violated. What's more, it subverts the Ant concept that properties are immutable, and remain set once you've set them: in the example above, `srcdir` and `destdir` will have different values at different times (because they exist inside different Ant processes).

Basically what we're doing here is breaking all of Ant's paradigms to force it to do what we want. Before Ant 1.6 you could have considered it a necessary evil. Now, it's just evil.

The horrific way: custom tasks

Ant allows you to write your own tasks (not targets) in Java. So our example would look something like Listing 3 (Java) and Listing 4 (Ant).

Here we write the shared code as a Java task, then call that task from inside targets **A** and **B**. The only word to describe this approach is 'cumbersome'. Not only do we need to ensure our code gets compiled before we try to use it, and add a `taskdef` to allow Ant to see our new task (meaning every target gets a new dependency on the 'first' target), but much worse, our re-used code has to be written in Java, rather than the Ant syntax we're using for everything else. At this point you might start asking yourself why

```
<target name="first">
  <javac srcdir="mycompile"/>
  <taskdef name="mycompile" classname="MyCompile"
    classpath="mycompile"/>
</target>

<target name="A" depends="first">
  <mycompile srcdir="a/src" destdir="a/bin"/>
</target>

<target name="B" depends="first">
  <mycompile srcdir="b/code" destdir="b/int"/>
</target>
```

Listing 4

you're using Ant at all – my thoughts start drifting towards writing my own build scripts in Java ... anyway, I'm sure that would be a very bad idea.

The relatively OK way: macrodef

So, enough teasing. Listing 5 shows the Right Way.

Since Ant 1.6, we have the **macrodef** task, which allows us to write our own tasks in Ant syntax. In any other language these would be called functions, with arguments which Ant calls attributes. You use these attributes by giving their name wrapped in a `@{}` rather than the normal **${}** for properties. The body of the function lives inside a sequential tag.

This allows us to write re-usable tasks within Ant. But what about re-using parts from the other language – the declarative targets and dependencies?

Avoiding repeated dependencies?

Imagine we have a build file containing targets like this:

```
<target name="everyoneneedsme"...
<target name="A" depends="everyoneneedsme"...
<target name="B" depends="everyoneneedsme"...
<target name="C" depends="everyoneneedsme"...
<target name="D" depends="everyoneneedsme"...
```

In Ant, I don't know how to share this. The best I can do is make a single target that is re-used whenever I want the same long list of dependencies, but in a situation like this where everything needs to depend on something, I don't know what to do. (Except, of course, drop to the Nuclear Option of the **<script>** tag, which we'll see later.)

I haven't used it in anger, but this kind of thing seems pretty straightforward with Gradle. I believe Listing 6 is roughly equivalent to my example above, but I hope someone will correct me if I get it wrong.

(Disclaimer: I haven't run this.)

So, if you want nice features in your build tool, like code-reuse and testability, you should consider a build tool that is integrated into a grown-

```
<macrodef name="mycompile">
  <attribute name="srcdir"/>
  <attribute name="destdir"/>
  <sequential>
    <javac
      srcdir="@{srcdir}" destdir="@{destdir}"
      classpath="myutil.jar" debug="false"
    />
  </sequential>
</macrodef>

<target name="A">
  <mycompile srcdir="a/src" destdir="a/bin"/>
</target>

<target name="B">
  <mycompile srcdir="b/code" destdir="b/int"/>
</target>
```

Listing 5

```
task everyoneneedsme
tasks.whenTaskAdded { task ->
    task.dependsOn everyoneneedsme
}
task A
task B
task C
task D
```

Listing 6

up programming language where all this stuff comes for free. But, if you're stuck with Ant, you should not despair: basic good practice is possible if you make the effort.

Everyone loves build.xml (test-driven Ant)

Of course, if you're going to have any confidence in your build file you're going to need to test it. Now we've learnt some basic Ant techniques, we're ready to do the necessary magic that allows us to write tests.

First, let me clear up what we're testing:

What do we want to test?

We're not testing our Java code. We know how to do that, and to run tests, if we've written them using JUnit [JUnit], just needs a **<junit>** tag in our `build.xml`. (Other testing frameworks are available and some people say they're better.)

The things we want to test are:

- **build artifacts** – the 'output' of our builds i.e. JAR files, zips and things created when we run the build,
- **build logic** – such as whether dependencies are correct, whether the build succeeds or fails under certain conditions, and
- **units of code** – checking whether individual macros or code snippets are correct.

Note, if you're familiar with the terminology, that testing build artifacts can never be a 'unit test', since it involves creating real files on the disk and running the real build.

Below we'll see how I found ways to test build artifacts, and some ideas I had to do the other two, but certainly not a comprehensive solution. Your contributions are welcome.

Before we start, let's see how I'm laying out my code:

Code layout

```
build.xml       - real build file
asserts.xml     - support code for tests
test-build.xml - actual tests
```

I have a normal build file called `build.xml`, a file containing support code for the tests (mostly macros allowing us to make assertions) called `asserts.xml`, and a file containing the actual tests called `test-build.xml`.

To run the tests I invoke Ant like this:

```
ant -f test-build.xml test-name
```

`test-build.xml` uses an **include** to get the assertions:

```
<include file="asserts.xml"/>
```

Tests call a target inside `build.xml` using subant, then use the code in `asserts.xml` to make assertions about what happened.

Simple example: code got compiled

If we want to check that a **<javac ...>** task worked, we can just check that a `.class` file was created. Here's the test, in `test-build.xml`:

```
<target name="test-class-file-created">
  <assert-target-creates-file
    target="build"
    file="bin/my/package/ExampleFile.class"
  />
</target>
```

```
<macrodef name="assert-file-exists">
  <attribute name="file"/>
  <sequential>
    <echo message=
       "Checking existence of file: @{file}"/>
    <fail message=
       "File '@{file}' does not exist.">
      <condition>
        <not><available file="@{file}"/></not>
      </condition>
    </fail>
  </sequential>
</macrodef>
```

Listing 7

We run it like this:

```
ant -f test-build.xml test-class-file-created
```

The **assert-target-creates-file** assertion is a **macrodef** in `asserts.xml` like this:

```
<macrodef name="assert-target-creates-file">
  <attribute name="target"/>
  <attribute name="file"/>
  <sequential>
    <delete file="@{file}" quiet="true"/>
    <subant antfile="build.xml" buildpath="."
            target="@{target}"/>
    <assert-file-exists file="@{file}"/>
  </sequential>
</macrodef>
```

It just deletes a file (if it exists), runs the target using **subant**, then asserts that the file exists, which uses the **macrodef** in Listing 7.

This uses a trick I've used a lot, which is the **fail** task, with a condition inside it, meaning that we only fail if the condition is satisfied. Here we use **not available** which means fail if the file doesn't exist.

Harder example: JAR file

Now let's check that a JAR file was created, and has the right contents. Listing 8 is the test.

This just says after we've run the target, the file `MyProduct.jar` exists, and it contains a file called `MANIFEST.MF` that has the right **Main-Class** information in it.

assert-file-in-jar-contains looks like Listing 9, which basically unzips the JAR into a directory, then searches the directory using fileset for a file with the right name and contents, and fails if it's not found (i.e. if the resourcecount of the fileset is zero). These are the kinds of backflips you need to do to bend Ant to your will.

Or, you can choose the Nuclear Option.

The Nuclear Option

If ant tasks just won't do, since Ant 1.7 and Java 1.6 we can drop into a **<script>** tag. You ain't gonna like it:

```
<script language="javascript"><![CDATA[
  system.launchMissiles(); // Muhahahaha
]]></script>
```

```
<target name="test-jar-created-with-manifest">
  <assert-target-creates-file
    target="build"
    file="dist/MyProduct.jar"
  />
  <assert-file-in-jar-contains
    jarfile="dist/MyProduct.jar"
    filename="MANIFEST.MF"
    find="Main-Class: my.package.MyMain"
  />
```

Listing 8

```
<macrodef name="assert-file-in-jar-contains">
  <attribute name="jarfile"/>
  <attribute name="filename"/>
  <attribute name="find"/>
  <sequential>
    <!-- ... insert checks that jar exists, and
     contains file -->
    <delete dir="${tmpdir}/unzip"/>
    <unzip src="@{jarfile}"
           dest="${tmpdir}/unzip"/>
    <fail message="@{jarfile}:@{filename} should
          contain @{find}">
      <condition>
        <resourcecount when="equal" count="0">
          <fileset dir="${tmpdir}/unzip">
            <and>
              <filename name="**/@{filename}"/>
              <contains text="@{find}"/>
            </and>
          </fileset>
        </resourcecount>
      </condition>
    </fail>
    <delete dir="${tmpdir}/unzip"/>
  </sequential>
</macrodef>
```

Listing 9

The script tag allows us to use a scripting language as provided through the JSR 223 Java feature directly within our Ant file [DrDobbs], meaning we can do anything.

In all the JVMs I've tried, the only scripting language actually available is JavaScript, provided by the Rhino virtual machine [MDN], which is now part of standard Java.

When using the script tag, expect bad error messages. Rhino produces unhelpful stack traces, and Ant doesn't really tell you what went wrong.

So now we know how to test the artifacts our build produces, but what about directly testing the logic in `build.xml`?

Testing build logic

We want to:

- Confirm that targets succeed or fail under certain conditions
- Check indirect dependencies are as expected
- Test a unit of Ant logic (e.g. a macrodef)

Success and failure

Listing 10 is a little macro I cooked up to assert that something is going to fail.

```
<macrodef name="expect-failure">
  <attribute name="target"/>
  <sequential>
    <local name="ex.caught"/>
    <script language="javascript"><![CDATA[
      try {
        project.executeTarget( "@{target}" );
      } catch( e ) {
        project.setProperty( "ex.caught", "yes" )
      }
    ]]></script>
    <fail message="@{target} succeeded!!!"
          unless="ex.caught"/>
  </sequential>
</macrodef>
```

Listing 10

I resorted to the Nuclear Option of a script tag, and used Ant's Java API (through JavaScript) to execute the target, and catch any exceptions that are thrown. If no exception is thrown, we fail.

Testing dependencies

To check that the dependencies are as we expect, we really want to run ant's dependency resolution without doing anything. Remarkably, ant has no support for this. But we can hack it in (see Listing 11).

(See 'Dry run mode for Ant' [Balaam] for more.)

Now we need to be able to run a build and capture the output. We can do that like Listing 12.

We use ant to run the build, telling it to write to a file `cdeps.txt`. Then, to assert that **`C`** depends on **`A`**, we just fail if `cdeps.txt` doesn't contain a line indicating we ran **`A`**. (To assert a file contains a certain line we use a load of **`fail`**, **`condition`**, **`resourcecount`** and **`fileset`** machinery as before. This could do with some improvement to cover target names that overlap – for example 'compile-a' will be wrongly found if 'test-compile-abc' was run.)

So, we can check that targets depend on each other, directly or indirectly. Can we write proper unit tests for our **`macrodef`**s?

Testing ant units

To test a **`macrodef`** or target as a piece of logic, without touching the file system or really running it, we will need fake versions of all the tasks, including **`<jar>`**, **`<copy>`**, **`<javac>`** and many more.

If we replace the real versions with fakes and then run our tasks, we can set up our fakes to track what happened, and then make assertions about it.

If we create a file called `real-fake-tasks.xml`, we can put things like this inside:

```
<macrodef name="jar">
  <attribute name="destfile"/>
  <sequential>
    <property name="jar.was.run" value="yes"/>
  </sequential>
</macrodef>
```

and, in `build.xml` we include something called `fake-tasks.xml`, with the optional attribute set to **`true`**:

```
<include file="fake-tasks.xml" optional="true"/>
```

If the target we want to test looks like this (in `build.xml`):

```
<target name="targetA">
  <jar destfile="foo.jar"/>
</target>
```

Then we can write a test like this in `test-build.xml`:

```
<target name="test-A-runs-jar"
        depends="build.targetA">
  <fail message="Didn't jar!"
        unless="jar.was.run"/>
</target>
```

and run the tests like this:

```
cp real-fake-tasks.xml fake-tasks.xml
ant -f test-build.xml test-A-runs-jar
rm fake-tasks.xml
```

If `fake-tasks.xml` doesn't exist, the real tasks will be used, so running your build normally should still work.

This trick relies on the fact that our fake tasks replace the real ones, which appears to be an undocumented behaviour of my version of Ant. Ant complains about us doing this, with an error message that sounds like it didn't work, but actually it did (on my machine).

If we wanted to avoid relying on this undocumented behaviour, we'd need to write our real targets based on special **`macrodef`**s called things like do-jar and provide a version of do-jar that hands off to the real jar, and a version that is a fake. This would be a lot of work, and pollutes our production code with machinery needed for testing, but it could work with Ant's documented behaviour, making it unlikely to fail unexpectedly in the future.

```
<target name="printCdeps">
  <script language="javascript"><![CDATA[
    var targs = project.getTargets().elements();
    while( targs.hasMoreElements() )
    {
      var targ = targs.nextElement();
      targ.setUnless( "DRY.RUN" );
    }
    project.setProperty( "DRY.RUN", "1" );
    project.executeTarget( "targetC" );
  ]]></script>
</target>
```

Listing 11

```
<target name="test-C-depends-on-A">
  <delete file="${tmpdir}/cdeps.txt"/>
  <ant
    target="printCdeps"
    output="${tmpdir}/cdeps.txt"
  />
  <fail message="Target A did not execute when
        we ran C!">
    <condition>
      <resourcecount when="equal" count="0">
        <fileset file="${tmpdir}/cdeps.txt">
          <contains text="targetA:"/>
        </fileset>
      </resourcecount>
    </condition>
  </fail>
  <delete file="${tmpdir}/cdeps.txt"/>
</target>
```

Listing 12

Summary

You can write Ant code in a test-driven way, and there are even structures that allow you to write things that might be described as unit tests.

At the moment, I am using mostly the 'testing artifacts' way. The tests run slowly, but they give real confidence that your build file is really working.

Since I introduced this form of testing into our build, I enjoy working with `build.xml` a lot more, because I know when I've messed it up.

But I do spend more time waiting around for the tests to run. ■

References

[Balaam] 'Dry run mode for Ant' on *Andy Balaam's blog* http://www.artificialworlds.net/blog/2013/01/31/dry-run-mode-for-ant-ant-n-ant-dry-run/

[Gant] http://gant.codehaus.org/

[Gradle] http://www.gradle.org/

[DrDobbs] http://www.drdobbs.com/jvm/jsr-223-scripting-for-the-java-platform/215801163

[JUnit] http://junit.org/

[Make] http://www.gnu.org/software/make/

[MDN] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

[SCons] http://www.scons.org/

Defining Visitors Inline in Modern C++

The VISITOR pattern can involve non-local boilerplate code. Robert Mill and Jonathan Coe present an inline VISITOR in C++.

The VISITOR pattern can be useful when type-specific handling is required and tight coupling of type-handling logic and handled types is either an acceptable cost or desirable in its own right. We've found that selective application of the classical VISITOR pattern adds strong compile-time safety, as the handling of new types needs explicit consideration in every context where type-specific handling occurs. The VISITOR pattern presents an inversion of control that can feel unnatural and often requires introduction of considerable non-local boilerplate code. We've found that this slows adoption of the VISITOR pattern especially among engineers and scientists who traditionally write their type-handling logic inline. Here we present a solution for defining VISITORs inline.

The problem

In object-oriented programming, we may need to perform a function on an object of polymorphic type, such that the behaviour of the function is specific to the derived type. Suppose that for the abstract base class **`Polygon`** we derive the concrete classes **`Triangle`** and **`Square`**. The free function **`CountSides`**, returns the number of sides in the polygon, **`p`** (see Listing 1).

`CountSides` will need the derived type of the polygon **`p`** to compute its result, which is problematic, because its argument is conveyed by a reference of the base class type, **`Polygon`**.

```
struct Triangle : Polygon
{
  // members
}

struct Square : Polygon
{
  // members
}

int CountSides(Polygon& p)
{
  // implementation
}
```

Listing 1

Visitor pattern

The VISITOR design pattern offers a mechanism for type-specific handling using virtual dispatch [Gamma95]. In the words of Scott Meyers: "VISITOR lets you define a new operation without changing the classes of the elements on which it operates" [Meyers06]. The pattern uses the **`this`** pointer inside the class to identify the derived type. Each derived object must accept a VISITOR interface which provides a list of **`visit`** members with a single argument overloaded on various derived types. To continue our illustration, the **`PolygonVisitor`** is able to visit **`Triangle`**s and **`Square`**s, and all these polygons must be able to accept a **`PolygonVisitor`**. (See Listing 2.)

```
struct Triangle;
struct Square;

struct PolygonVisitor
{
  virtual ~PolygonVisitor() {}

  virtual void visit(Triangle& tr) = 0;
  virtual void visit(Square& sq) = 0;
};

struct Polygon
{
  virtual void accept(PolygonVisitor& v) = 0;
}
```

Listing 2

`Square`s and **`Triangle`**s accept the VISITOR as shown in Listing 3. Observe that the **`this`** pointer is used to select the appropriate overloaded function in the VISITOR interface.

```
struct Triangle : Polygon
{
  void accept(PolygonVisitor& v) override
  {
    v.visit(*this);
  }
};

struct Square : Polygon
{
  void accept(PolygonVisitor& v) override
  {
    v.visit(*this);
  }
};
```

Listing 3

A VISITOR object, **`SideCounter`**, which counts the number of sides of a polygon and stores the result, is implemented and used as in Listing 4.

Robert Mill received his bachelor and Ph.D. degrees in Computer Science from the University of Sheffield. He now works in industrial process engineering as a mathematical developer, and retains an interest in machine learning and signal processing.

Jonathan Coe has been programming commercially for about 6 years. He has worked in the energy industry on process simulation and optimisation and is currently employed in the financial sector.

it requires the **creation of a new visitor object** type for **each algorithm** that operates on the derived type

```
struct SideCounter : PolygonVisitor
{
  void visit(Square& sq) override
  {
    m_sides = 4;
  }

  void visit(Triangle& tr) override
  {
    m_sides = 3;
  }

  int m_sides = 0;
};

int CountSides(Polygon& p)
{
  SideCounter sideCounter;
  p.accept(sideCounter);
  return sideCounter.m_sides;
}
```

Listing 4

Inline Visitor pattern

One potential drawback of the VISITOR pattern is that it requires the creation of a new visitor object type for each algorithm that operates on the derived type. In some cases, the class created will not be reused and, much like a lambda, it would be more convenient to write the visitor clauses inline. Listing 5 shows how this can be accomplished in a form that resembles a `switch` statement.

```
int CountSides(Polygon& p)
{
  int sides = 0;

  auto v = begin_visitor<PolygonVisitor>
    .on<Triangle>([&sides](Triangle& tr)
    {
      sides = 3;
    })
    .on<Square>([&sides](Square& sq)
    {
      sides = 4;
    })
    .end_visitor();

  p.accept(v);
  return sides;
}
```

Listing 5

In Listing 6, we demonstrate generic code that permits the `begin_visitor` ... `end_visitor` construction to be used with any `visitor` base. The initial `begin_visitor` call instantiates a class which defines an inner object inheriting from the visitor interface; each subsequent call of the `on` function instantiates a class whose inner class inherits from the previous inner class implementing an additional `visit` function. Finally the `end_visitor` call returns an instance of the inner visitor class.

```
template <typename T,
          typename F,
          typename BaseInner,
          typename ArgsT>
struct ComposeVisitor
{
  struct Inner : public BaseInner
  {
    using BaseInner::visit;
    Inner(ArgsT&& args) :
      BaseInner(move(args.second)),
      m_f(move(args.first))
    {
    }
    void visit(T& t) final override
    {
      m_f(t);
    }
  private:
    F m_f;
  };
  ComposeVisitor(ArgsT&& args) :
    m_args(move(args))
  {
  }
  template <typename Tadd,
            typename Fadd>
  ComposeVisitor<
    Tadd,
    Fadd,
    Inner,
    pair<Fadd, ArgsT>> on(Fadd&& f)
  {
    return ComposeVisitor<
      Tadd,
      Fadd,
      Inner,
      pair<Fadd, ArgsT>>(
        make_pair(
          move(f),
          move(m_args)));
  }
```

Listing 6

That inline visitors cannot be constructed when clauses are missing may also be considered desirable in some contexts

```
  Inner end_visitor()
  {
    return Inner(move(m_args));
  }

  ArgsT m_args;
};
template <typename TVisitorBase>
struct EmptyVisitor
{
  struct Inner : public TVisitorBase
  {
    using TVisitorBase::visit;
    Inner(nullptr_t) {}
  };

  template <typename Tadd, typename Fadd>
  ComposeVisitor<
    Tadd,
    Fadd,
    Inner,
    pair<Fadd, nullptr_t>> on(Fadd&& f)
  {
    return ComposeVisitor<
      Tadd,
      Fadd,
      Inner,
      pair<Fadd, nullptr_t>>(
        make_pair(
          move(f),
          nullptr));
  }
};
template <typename TVisitorBase>
EmptyVisitor<TVisitorBase> begin_visitor()
{
  return EmptyVisitor<TVisitorBase>();
}
```

Listing 6 (cont'd)

The consistency between the list of types used with **`on`** and those in the visitor base is verified at compilation time. Since the **`override`** qualifier is specified on the **`visit`** member function, it is not possible to add a superfluous **`visit`** which does not correspond to a type overload in the visitor base. Similarly, because the **`final`** qualifier is specified on the **`visit`** member function it is not possible to define a **`visit`** member function more than once. That inline visitors cannot be constructed when clauses are missing may also be considered desirable in some contexts. For instance, if a new type **`Hexagon`** is derived from **`Polygon`**, then the code base will compile only when appropriate **`visit`** functions been introduced to handle it. In large code bases, this may serve maintainability. If it is deemed that a visitor clause should have some default behaviour (e.g., no operation), a concrete visitor base can be passed into **`begin_visitor`**.

Performance

With optimizations turned on MSVC 2013, GCC 4.9.1 and Clang 3.4.2 compile the inline visitor without introducing any cost. GCC and Clang produce identical assembly code in the case when a **`visitor`** class is explicitly written out. MSVC produces different assembly code for the inline visitor and explicit visitor class; the inline visitor has been measured to run marginally faster.

Other visitors

Loki's Acyclic Visitor [Martin] [Loki] removes compile-time coupling from visiting and visited classes but at the cost of introducing dynamic casts and run-time detection of unhandled types. When run-time performance and compile-time detection of unhandled types are favoured over shorter compile-times then we would recommend use of the inline visitor. The inline visitor does not have the flexibility of the Cooperative Visitor [Krishnamoorthi07], which allows different method names and return types, but as it is intended to be lightweight this flexibility is not needed: the **`visit`** functions are not explicitly named and variables in local scope can be modified by lambda capture alleviating the need for a return value.

Conclusion

We have presented a method for defining inline visitors in standard C++. The method does not, by design, remove the tight coupling between visited and visiting class hierarchies. Performance, portability and convenience of the inline visitor mean that we would encourage its use where tight-coupling is acceptable and type-specific handling is logically localized. ■

References

[Gamma95] E. Gamma et al., *Design Patterns*, Addison-Wesley Longman, 1995.

[Krishnamoorthi07] A. S. Krishnamoorthi, 'The Cooperative Visitor: A Template Technique for Visitor Creation', 11 July 2007, *Artima Developer* http://www.artima.com/cppsource/cooperative_visitor.html

[Loki] Loki library http://loki-lib.sourceforge.net/

[Martin] R. C. Martin, 'Acyclic Visitor' http://www.objectmentor.com/resources/articles/acv.pdf

[Meyers06] S. Meyers, 'My Most Important C++ Aha! Moments...Ever', *Artima Developer* http://www.artima.com/cppsource/top_cpp_aha_moments.html

A Scheduling Technique for Small Software Projects and Teams

Despite myriad scheduling tools, projects still overrun. Bob Schmidt presents some tips for accurate scheduling.

With all of the scheduling programs on the market today one might wonder why software project deadlines are still missed and over budget. These programs look at the tasks and schedule employees accordingly. What these programs don't take into account is the human factor: software professionals need food, drink and rest, and don't always operate according to plan. The irony is that a scheduling program developed by software professionals may not completely meet the needs of software professionals teamed to complete a programming project.

This article introduces tips and techniques for creating and maintaining a more accurate and easy-to-use schedule, in the context of small software teams.

The systems and practices you are about to see occurred at a real company. The names have been changed to protect, well, everyone but me.

The scenario

Your company has been contracted to develop a real-time process control system with a combination of software and hardware deliverables and a pre-determined schedule with a completion date fixed by contract.

These contracts are bid by marketing and sales with assistance from project managers, using minimal input from people who will do the actual work. The bids contain salary and overhead for *X* software bodies of differing levels. These are 'virtual' bodies; in most cases actual personnel aren't assigned to the project until after contract award and notice to proceed. Too often a large number of the bodies belong to people who are coming free from a prior job, rather than people whose skill sets match the job's requirements.

The project team consists of a project manager (who is responsible for more than one project); a system engineer responsible for the day-to-day running of the project, primarily on the hardware side; one or more hardware specialists who report to the system engineer; a lead analyst responsible for the software side of the project, who reports to the project manager but coordinates through the system engineer; and one or more software specialists who report to the lead analyst.

The problem

The system engineer is given the task of scheduling the project, usually before all of the other individuals have been assigned to the project. The result of this work is a detailed critical path chart, with finish-to-start dependencies neatly laid out in detail: purchase hardware items A and B; receive hardware items A and B; assemble hardware items A and B; test assembly; etc. Buried somewhere in the midst of all of this hardware mumbo jumbo is one item – Software – which starts with the project's notice to proceed and ends at the factory acceptance test.

Time estimates

I estimate projects using days as the unit of time, with no task taking less than one day to complete; a colleague advocated scheduling no task to take less than a week. Either way, your time estimates should be based on what can be accomplished in eight hour days and/or five-day work-weeks.

Harumph, you say – in the real world we all work (at least) six ten-hour days. This may be true, but if you schedule people based on the true number of hours they work, then you allow your project no slack time with which to take care of all of the things you can't anticipate. It may be that your team members work 60 plus hours a week because they are true workaholics; more likely they work crazy hours because they feel forced to do so because of unrealistic scheduling. It is far better to base your initial estimates on a forty-hour week (or whatever your cultural norm). Your team members will love you for being realistic, and they will be more willing – and able – to put in those long hours when a true crisis calls for it..

The system engineer, who is usually a hardware person, concerns himself very little with the software end of things. He's been given the project's start and end dates, and hardware to deal with, and, well, the rest is SMOP – a Simple Matter of Programming – and left up to the software team.

The software team is faced with producing the software, in a finite period of time, with a fixed set of requirements, without knowing the true scope of the software development effort. Nothing the team says or does will change management's attitude toward the deadlines imposed by the project schedule, especially because it is common for the contracts for projects of this type to include provisions for liquidated damages.

Liquidated damages – two words guaranteed to keep a C-level executive up at night. Liquidated damages are monetary penalties imposed on the contractor for missed deadlines. Keep in mind that these types of projects are often all-or-nothing affairs. Partial solutions that do not fulfill the entire specification just aren't good enough. A control system for a water treatment plant that does not implement the dual-media-filter backwash process is not going to be of any use.

The solution is for someone on your team, who best knows the strengths and weakness and dependencies of your team, to take control of the software schedule. Ideally, the lead analyst should be the one to do this; however, any of the more experienced analysts can do the job as long as the support is there.

The road to schedule happiness

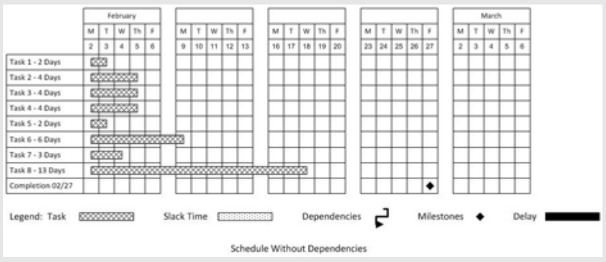
The first step on the road to schedule nirvana is to get a grasp of what needs to be done. I recommend developing a short description of each task to be performed (which may map to a program or programs to be written or modified on the coarse-grained end of the detail spectrum, to more specific functionality on the fine-grained end). Record pertinent information about the task, including the task name, a short description of the task, references to the purchase specification (the customer's document) and the system

Bob Schmidt is president of Sandia Control Systems, Inc. in Albuquerque, New Mexico. In the software business for 33 years, he specializes in software for the process control and access control industries, and dabbles in the hardware side of the business whenever he has the chance. He can be contacted at bob@sandiacontrolsystems.com.

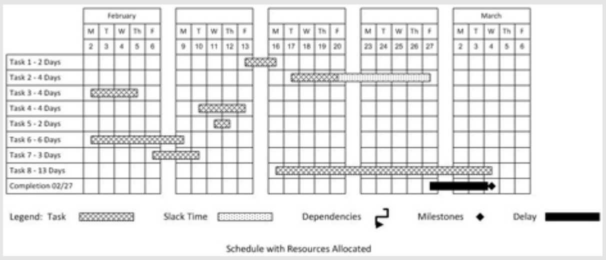
Dependencies vs. resource allocation

Scheduling packages allow tasks to be allocated to resources (people), and they allow dependencies between tasks to be defined. A task dependency simply states that one task needs to be completed before another can be started.

If tasks are entered into the scheduling software without resource allocation or explicit dependencies, you wind up with a schedule that looks like this:



All tasks start at the same time; this is not very useful. By adding resource definitions to the tasks, the scheduler can spread the tasks out in time. This shows what this might look like:



This figure looks an awful lot like Figure 2, but is even less readable. It is not immediately obvious which tasks are in whose timeline.

I like to use explicit dependencies on tasks assigned to a person. My experience is that it is easier to use the scheduler this way than assigning resources and letting the scheduler make its own decisions about which tasks should be done first. Plus, there are always real dependencies between tasks, both within the timeline of a single person and between the timelines of different persons. Sticking with one method of defining the ‘end-to-start’ relationships between tasks simplifies my job, by eliminating the modification of the additional resource data within the scheduler.

functional specification or high-level design document (your response to the purchase spec), any items or other tasks the task depends on, and an estimated time to perform the task. (See the ‘Time estimates’ and ‘High vs. low tech’ boxes.) There should also be a space for the person assigned to the task, a starting date and a completion date.

Your first pass at filling in this data is just that – a first pass. You may not be able to fill in the details of all tasks right away; create a placeholder for it, and fill in the details when you know them. (You most likely will add tasks during the course of the project, as those inevitable gotchas spring up.) Where appropriate, scribble in the name or names of the people on the team who have to do the task (because of their familiarity with the existing code, or areas of expertise, for example) or who are able do the task (based on what you already know about the rest of the project team members).

I usually defined these tasks at a fairly high level of abstraction. Typically, a task for a multitasking process control system correlated to a task running in that system. In some cases a task was broken down further; the need to integrate a new piece of process I/O gear might have been broken into a communications section, an input section (for analog and discrete inputs, pulse accumulators, etc.), and an output section (for analog and discrete outputs).

At this point you should be ready to discuss your results with the rest of the project team. You should encourage them to inspect your results, and comment on them. What you want is to receive feedback on your time estimates from the people who are going to be doing the work. You also will learn of additional tasks; you may also find out that two or more tasks can be better stated as one.

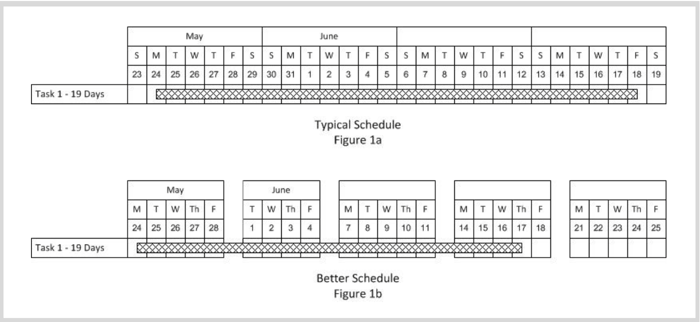


Figure 1

The schedule

Now that you have a grasp on the ‘what’, it is time to worry about the ‘how long’. For this you need a software package that creates schedules, such as Microsoft Project or Primavera. These typically create a schedule with a calendar along the top, tasks down the left hand side, and which shows when a task is to be performed by printing a bar to the right of the task aligned with the actual dates across the top. The package you choose should allow you to enter dependencies, milestones, etc. and should also calculate slack time and critical paths.

Most scheduling packages are highly configurable, so this is where you should start. I recommend the following initial assumptions:

- Configure the package to plot tasks on a daily basis,
- Configure the package to reflect all of the standard holidays your company schedules, and
- Configure the package to not count (or plot) weekends and holidays when scheduling.

If you follow the first suggestion you will wind up with a much larger chart, but it will contain more detail; you will be thankful for that detail later. The last two items are very important for a very simple reason: there is nothing that will deflate morale quicker than to see that work has been scheduled for every weekend and every holiday, even if the package automatically adds days to a task which spans weekends and holidays. I believe the chart in Figure 1b is much more desirable than the chart in Figure 1a, because there is a more visible one-to-one relationship between your time estimates (in days) and the number of days shown on the chart. If your package does not allow you to filter out weekends and holidays, then you should go out of your way to show your fellow team members that the duration of the task is based on a five-day work-week.

Creating the first pass of the schedule requires time and patience. The first step is to take all of the task assignments and sort them by the people who are able to perform the tasks. If a task can be accomplished by more than one team member, take a guess and assign it to one member. (The guess can be based on who you think is best for the job; or, if team members are equally talented, whom you think is the more lightly loaded.)

The next step is to sort each team member’s initial assignments by priority and by any dependencies. Once you have performed this initial sort it is time to plug all of the data into the scheduler.

If your previous schedules have been like the ones I have received from ‘on high’, the tasks all are entered into the package, without regard to any previously known dependencies, or the person or persons who are going to do the work. This results in a schedule that is very difficult to follow, requiring the use of mental calisthenics to follow a dependency path from start to finish as you bounce up and down the chart. Look at the sample schedule in Figure 2, and imagine what it would take to follow if there were 100 tasks scheduled over 12 months or more.

I believe in creating a schedule that can be followed by mere mortals, so I base my chart on people. Although this is a more difficult way to use these types of scheduling programs, the results are worth the effort.

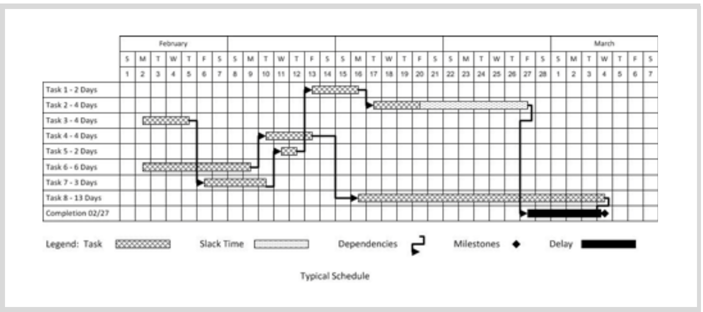


Figure 2

High vs. low tech

I developed this technique back in the mid-1980s when IBM PCs were under-powered and over-priced, so a lot of the work of defining tasks was done by hand, written with pencil on paper placed in a 3-ring binder. We used Quicknet for producing schedules; the resulting charts were printed on green-bar, with pages taped together to form a horizontally-oriented calendar.

This scheduling technique works independently of the level of technology used to implement it. I used the project timeline stencils that come with Visio to produce my last schedule, working from paper-based forms filled in by hand. (The diagrams accompanying this article were prepared that way.) A higher tech solution was not required.

The level of technology you apply to this technique should be tailored to your situation. Do you have a distributed team? A low-tech 3-ring binder approach to task definitions may not work, and a printed schedule pinned to the wall will not be accessible to all team members. Already have a software-based manner of defining and tracking tasks or change requests? Use the tool to accumulate the required data and then create your schedule. (I have a customer that uses Borland's StarTeam for this purpose. Their change request entries contain task descriptions and time estimates perfectly suited as inputs to the scheduling process.)

Start with the assignments for one team member only. Enter this team member's assignments into the scheduler, in an order that makes sense. Make each task's start date dependent on the prior task's end date, as well as any other dependencies you have noted. (See the 'Dependencies vs. resource allocation' box.) If a task has a hard start date (which might be caused by the known arrival date of a prerequisite for the task) or a hard end date, enter these dates into the scheduler, too. Repeat this task for each team member's assignments. What you want to end up with is an initial schedule that looks like the one in Figure 3.

The format of the schedule in Figure 3, which is people-based, is much easier to follow than the one in Figure 2 (which contains the same data). It is also easier to detect an imbalance between the loadings of each of the team members, places where dependencies between team members cause slack time in one member's schedule, and where the critical path lies.

The last step (and perhaps the most lengthy) is to minimize the slack time in each team member's schedule, and to attempt to equalize the end date of each member's schedule, by switching tasks between members (where possible) and rearranging the order of arbitrarily ordered tasks. (Don't kill yourself trying to get all of the slack time out of your schedule; a little slack time scattered throughout each team member's schedule can help absorb some of the inevitable delays.) For the simple example in Figure 3, if Task 4 can be performed by either of the team members (and assuming there are no dependencies which would prevent the reassignment), it can be reassigned to the first team member to achieve the schedule shown in Figure 4. The result is a schedule that beats the completion milestone.

Now that you have fiddled, tweaked, moved, rearranged, reordered and otherwise completely redone your schedule multiple times, and have finally come up with a schedule that optimizes your resources, what do you do with it? If you are one of the fortunate few, your schedule has ample slack time in it, there are no missed milestones, and most – or all – of the critical path is assigned to your most productive team member, then – get to work! However, if you are like the rest of us, your carefully crafted schedule shows what you expected all along: there just aren't enough days in the week to get your project done on time.

This is the point where doing all of this upfront work really pays off. You could have taken your task sheets, added up all of the time estimates, divided the sum by the number of people allocated to the team, and come to a similar conclusion without having expended nearly the same amount of effort. But your carefully crafted schedule is more apt to convince a manager that one of three things needs to be done: the completion date must be pushed out, features need to be cut from the specification, or you need one or more additional people. (A rational manager might ask you if you have cut out all of the fat from the schedule before recognizing the obvious; an irrational manager will just tell you to shorten the schedule, dammit, and don't bother me with your problems.)

On projects such as these, the completion date and specifications are contractual, and not likely to be open to discussion (remember – liquidated damages). So you are most likely going to be faced with adding one or more team members. Unfortunately, what will probably happen is that your manager will tell you to rework the schedule assuming that there are *X* additional players to be named later. Once again you will be faced with the problem of assigning tasks to unknown individuals, and reworking the schedule to meet the projects requirements.

Plan the dive, and dive the plan?

SCUBA divers are taught to plan a dive carefully before getting into the water, and then to 'dive the plan' and not to change the characteristics (maximum depth and time underwater) of the dive while in the water (with the exception of an emergency, of course). Well, the same can be said of your schedule. You have put a lot of time and effort into your schedule; the project team members should be required to stick to it, right? Penalties for missing a deadline should be stiff, tasks should never be performed out of order, heads will roll if major changes are necessary – you get the point.

What a bunch of hooey. In order for your schedule to be effective over time, it has to be flexible. This is simple reality – no project ever goes completely according to plan. Your schedule has to be a work-in-progress at all times.

A scheduler's work is never done

The easiest way to get the maximum results from your schedule is to follow these guidelines:

Print out your schedule using the finest grain you can handle. (I like a printed schedule that goes right down to days.) If your scheduling package

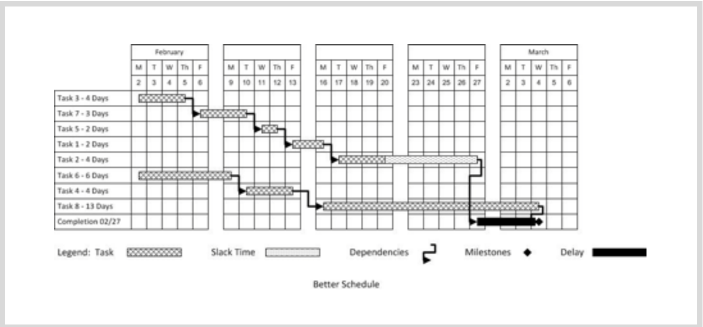


Figure 3

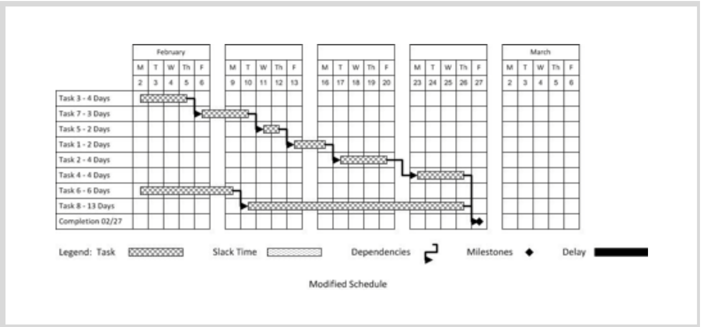


Figure 4

Schedule location

If you decide to post a copy of the schedule, avoid the tendency to turn it into what I call a 'public embarrassment board'. The schedule should not be hung in a high-traffic hallway, conference room or cafeteria.I once had a cubicle around the corner from another project's public embarrassment board. It was located in the hallway next to one of the main employee entrances to the building. To my knowledge it was never updated past a certain point in the project and continued to display missed deadlines and '90% complete' long after the job was finished. Perhaps the worst example of this sort of thing was a defense contractor that had graded the progress of its projects and placed the grades on the wall right in front of the main entrance to the facility. How would you feel if you saw that your project's current grade was a D minus or an F, every time you walked into work? Would you really want your customers to see it?

doesn't do it, print the name of each team member next to the line of tasks to which he or she is assigned.

Hang a copy of your completed schedule in a place that is easily accessible to the project team. (See the 'Schedule Location' box.) Try to display the schedule on one plane, without having to resort to multiple levels. Show your team the relationship between the tasks on the schedule and the tasks definitions. Encourage them to do the following: record the actual start and end dates of a task; mark the actual start and end dates of a task on the schedule; and record any problems they had.

Update the schedule once a month. Start by interviewing all of the project team members and asking them how their current task is going. Ask them how far they have progressed, and how much longer they think it will take to finish. You should also inquire about any other tasks on which they have worked since the previous month's update. Make it clear that you are not asking these questions to nail them on missed deadlines, but rather to make it easier for you to better estimate the rest of the job.

Insofar as the schedule is concerned, you have two goals to accomplish: 1) update the schedule so that it realistically reflects the activities of the team members over the past month; and 2) modify the schedule as necessary for future work. The first step is important because it accurately records the past. This is why you want the team members to mark down real start and end dates for tasks.

The second step is even more important, because it is here that you use the historical data to better predict the future. (If this seems dangerously close to collecting metrics, all the better. I won't presume to call any of this data a metric because the process is too subjective.) The goal is to continuously narrow the 'cone of uncertainty'. [McConnell97]

Use the results of your discussions with team members to estimate the completion of their current tasks. For example, if they say they are 50% done, and they have spent nine days on the task already, then adjust the schedule for the item to read 18 days (regardless of what it started out to be). You may need to adjust this up or down, depending on whether the team member is a strong starter or a strong finisher; use what you have learned of each team member from all of the prior schedules.

You may find that the schedule undergoes drastic changes during some or all of your updates. You may find that a team member is not suited for certain work you have assigned to him or her, requiring a complete shuffling of tasks for the rest of the project.

A schedule update is also the time to add tasks associated with those 'gotchas' I talked about earlier.

Another very good reason to update the schedule every month to accurately reflect past work is that it hides the evidence of missed schedules. This in turn makes it more likely that each team member will annotate the schedule accurately, which will make it a better forecasting tool. I advocate burning, shredding or, at the very least, hiding all previous schedules; you really don't want indicators of past performance lying around. It is also a good idea to keep any type of relative performance data out of the hands of management. If you are perceived as a performance information pipeline to management, the other team members may stop cooperating with your efforts. After all, when updating the schedule it is more important to know a person's total performance on a similar task than whether the person finished the job early or late.

What about Agile? Isn't this just big <something> up front?

What about it? I have heard great things about agile methodologies – Scrum, XP, and the like – and if I ever get to work on an agile project I'll let you know how it worked out. Agile methodologies seem to be short-term oriented – two to four week intervals of planning and execution. I can see how this works for internal development projects, or projects where partial, functional solutions are possible and even desirable. I have never been able to figure out how you would know if your resources are allocated correctly for a long term project if you don't take time to plan for the long term. (I am willing to be corrected on this.)

Although I haven't used it this way, these techniques could be adapted to a quasi-agile form. The schedule you create could be broken up into the agile increments in use by your team. In this case there would be a project milestone at the end of each interval, and the tasks assigned to team members would need to fit not only into the overall, long-term schedule, but each of the intervals as well.

Costs versus benefits

It took me about one full week to create an initial schedule for four or five team members and approximately 100 tasks. I spent one full day a month updating the schedule. I set aside the last work day of the month for this purpose. By keeping to a schedule for updating the schedule I was sure to get the updates done, and the other team members got used to thinking about the questions I would inevitably ask them.

I developed and refined this technique over the course of two projects, a nuclear power plant security system and a retrofit of a waste water treatment plant process control system. The security system project lasted twelve months and was staffed by four programmer/ analysts, including myself. The waste water treatment plant control system project lasted fourteen months and was staffed by four to six programmer/analysts at any given time. Both of these projects were completed and successfully passed their factory acceptance tests on time, a rare occurrence for those types of systems at that company.

I was not the lead analyst on either job, but I had the support of the lead analyst and project manager in both cases. The second project was one third of the way into its overall schedule when I joined the team. After I was given my assignments, I sat down to schedule them out using the techniques developed on the first project. The lead analyst asked what I was doing, and when I told him I was scheduling my own work he directed me to schedule everyone else on the team, too. We discovered there was no way to meet the schedule given the currently allocated resources, and were able to add a person early enough to finish on time.

There were some other benefits to this approach. Team members from the project supported the scheduling process, since they had a say in the time estimates used. We had a small level of process feedback, since I was able to use historical data to better estimate work yet to be done. And, best of all, we were able to complete both jobs with less overtime overall than other projects of their type. ■

Acknowledgements

My thanks to Nicole Wascoe Bauman for her input on the article, and to Don Perkins for supporting my efforts on these projects.

References

[McConnell97] *Software Project Survival Guide*, Steve McConnell

Paper Bag Escapology Using Particle Swarm Optimisation

Some attempts at programming one's way out of a paper bag need an upfront model. Frances Buontempo simplifies things using particle swarm optimisation.

Previously [Buontempo13c], I demonstrated how to use a genetic algorithm to program your way out of a paper bag. This had the main drawback that an initial model was required. In this specific case, we used the well-known equations for the ballistic trajectory of a projectile:

$$x = k + \frac{1}{2}w + vt\cos(\theta)$$

$$y = vt\sin(\theta) - \frac{1}{2}gt^2$$

In this article, I will introduce a method for solving problems which does not require a model up front. This is one of a vast class of optimisation techniques which seems to be well suited to paper bag escapology, though does have other somewhat more practical uses, for example finding training weights for neural networks. Though the genetic algorithm approach worked well, there are often circumstances in which one does not have a model, and even if armed with an upfront model it often requires calibration to find suitable constants. In the projectile equations, we just require a numerical value for the single constant *g*. In the general case more 'constants' may need finding, and frequently models need to be recalibrated since the constants they use are in fact not constant [Buontempo13a].Though, for example, measuring instruments will tend to just need calibrating once, in other areas, such as finance, models are recalibrated weekly or even more frequently, perhaps suggesting that the models are not that close to reality. Specifically, various sources ascribe the quote "When you have to keep recalibrating a model, something is wrong with it," to Paul Wilmott [e.g. Freedman11]. In contrast to approaches requiring a model, particle swarm optimisations still explore the solution space, partially randomly as with genetic algorithms, but exploit the idea of a social sharing of information, via the fitness function, thereby supposedly mimicking swarm or foraging behaviour.

Initial attempt

Starting from first principles, it is not difficult to write code to make one particle move around in space and stop when (and if) it finally escapes from a paper bag. If you will indulge me, I will use JavaScript this time, and draw on the HTML canvas. If you are unfamiliar with this technology there are many online tutorials to get you up to speed (for example, https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Canvas_tutorial). Hopefully the code is intuitive enough that it needs no explanation.

Given a canvas declared in html, with a message for browsers that do not support this

```
<canvas id="myCanvas" width="800" height="400">
  Your browser does not support the canvas element.
</canvas>
```

First, we draw a 2 dimensional bag and place a small rectangle, representing the particle somewhere in it. (See Listing 1.)

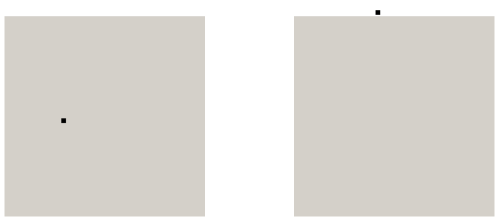
```
//Draw the bag
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.clearRect(0,0,c.width,c.height);
ctx.fillStyle="#E0B044";
var bag_width = 300;
var left = 75;
var right = left + bag_width;
var up = 25;
var down = up + bag_width;
ctx.fillRect(left,up,bag_width,bag_width);

//Draw the particle
ctx.beginPath();
ctx.rect(150, 150, width,width);
ctx.strokeStyle="black";
ctx.stroke();
```

Listing 1

Then, we set up a callback function, allowing the gap between invocations to redraw the canvas, which draws the present position and then moves the particle. If it escapes the bag we stop, otherwise we re-call the callback. Without loss of generality, assuming a square paper bag, we can move as follows: Starting with the particle at position (x, y), indicated by the small black rectangle at some point, say in the middle of the bag, it is allowed to move a little in either direction – vertically, up or down and horizontally, either left or right. The random movements are allowed to continue until the particle finally escapes the paper bag.

```
x += bag_width * 0.2 * (-0.5 + Math.random());
y += bag_width * 0.2 * (-0.5 + Math.random());
```

The particle starts as requested and wends its way round the canvas like an angry ant until it finds its way out of the bag.



When it escapes we cancel the callback using the `id` we remembered:

```
clearInterval(id);
```

In theory it may never escape, but always has done so far. It takes about 3 seconds, though varies greatly, sometimes taking just one second and other times more than 10.

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 15 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com

If they all move towards each other they are likely to swarm, or indeed clump, together and move no further

What have we learnt from this? Aside from how to use the canvas, very little apart from how simply trying some random moves can work. It could possibly be sped up, or optimised in a sense, by allowing several particles to set off on their journey simultaneously and seeing who wins.

Attempt 1 – Every man for himself

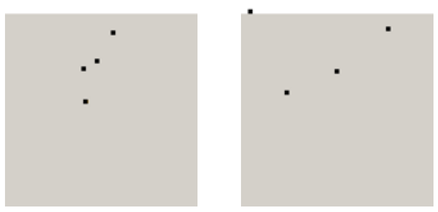
Changing the original code to have an array of 'Beasties' or particles, rather than just tracking the *x* and *y* coordinates of a single item is relatively straightforward. First we need a Beastie, perhaps given a starting *x* and *y* position:

```
function Beasty(x, y, id, index)
{
  this.x = x;
  this.y = y;
  this.id = id;
  this.index = index;
}
```

Then we need to track these, having decided how to start them off. I took the approach of clicking a button to form a new particle, though there are other options.

```
id = setInterval(function()
   { move(index); }, 100);
var beast = new Beasty(x, y, id, index);
ids.push(beast);
```

We store the **id** of the interval in order to cancel all the particles when we're done. As previously, the **move** function moves the given particle by a small random amount. If the particle ends up outside the bag it then stops and freezes the others in their tracks, using their **id**s. The algorithm could be altered to wait for all of them to escape. This is left as an exercise for the reader.



Previously we used a genetic algorithm to allow several attempts at problem solving to 'share knowledge' by combining the angle and velocity from randomly selected better particles of the previous generation to form a new better younger generation. In this approach, the particles each follow their own random walk and do not communicate with each other. If they influence one another we could end with all the particles escaping the paper bag.

Attempt 2 – The blind following the blind

Making the particles follow each other is relatively easy though will prove to be a foolish thing to do. There are several options, but obviously we can't have every particle following every other particle otherwise they are likely to freeze. If they all move towards each other they are likely to swarm, or indeed clump, together and move no further. A more fruitful approach might be a variant of the *k* nearest neighbours [k-NN] algorithm. Allowing each particle to take a step independently, but also pulling it towards some of its nearest neighbours will allow the particles to actually move but still tend to swarm together.

```
function knn(items, index, n) {
  var results =[];
  var item = items[index];
  for (var i=0; i<items.length; i++) {
    if (i !==index) {
      var neighbour = items[i];
      var distance = Math.sqrt(item.x*neighbour.x
                    + item.y*neighbour.y);
      results.push( new distance_index
         (distance, i) );
    }
  }
  results.sort( function(a,b) {
    return a.distance - b.distance;
  } );
  var top_n = Math.min(n, results.length);
  return results.slice(0,top_n);
}
```

Listing 2

To find the nearest neighbours of a given particle in our array of particles with index *index* we find its distance from the particle under consideration, order them by distance and just return the top *n* as in Listing 2.

The number of neighbours can either be specified in advance or changed as the simulation runs. I settled for the minimum of 5 and the number of particles, though as you can see the function is flexible. In general, the distance function must be chosen carefully so it is suitable for the domain. In our case, the straightforward Euclidean distance

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

should be suitable since this is inherently a spatial problem. For the mathematically challenged, think Pythagoras. Finding the average *x* and *y* displacement or nudge of these nearest neighbours from each particle can be used in conjunction with a small stochastic (or random) step (Listing 3).

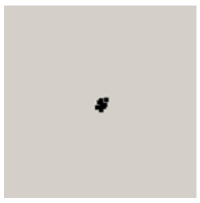
```
x_move += (x_nudge - beast.x)
         * neighbour_weight
         * (-0.5 + Math.random());
y_move += (y_nudge - beast.y)
         * neighbour_weight
         * (-0.5 + Math.random());
```

Listing 3

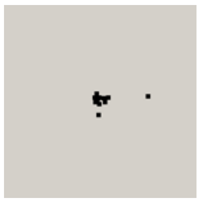
even though we intended to encourage them to swarm together they have tended to each escape from a different spot

Unfortunately, this means the particles do tend to swarm together but if some of them are not doing very well, they can greatly increase the time taken for all of the particles to escape. The particles do all tend to escape eventually but can take an hour or so to finish.

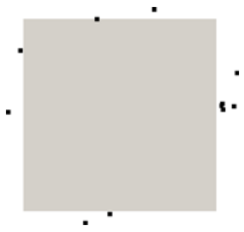
We tend to find the particles clump together initially



Sometimes one starts to escape



However, it can tend to be pulled back with the others. Clearly, its nearest neighbours are in the main clump or swarm of particles, so this is unsurprising. Usually, they do manage to move away from the main swarm.



The simulation stops once all the particles have escaped the paper bag. Notice that even though we intended to encourage them to swarm together they have tended to each escape from a different spot. They did swarm together but it seems that only the individual randomness allow individuals to escape from the mindless herd and then escape from the paper bag. This was not our intention, though it has paved the way for a more successful approach.

Attempt 3 – A swarm with memory

If each particle still tends to move randomly, but also moves towards the best of the rest rather than the nearest few of the rest and also is pulled towards its best position so far, it seems likely things may improve. This will allow the swarm to use what it discovers as it moves, both individually and from the swarm memory. In fact, this is the essence of a particle swarm optimisation [Kennedy95].

The pseudo code is as follows:

```
Choose n
Initialise n particles to a random starting point
  in the bag
While some particles are still in the bag
  Update best global position
  Draw particles current positions
  Move particles - updating each particle's
    current best position
```

In order to move the particles, each has a position and 'velocity'. In the `move` function, each particle's current velocity is first updated based on its current velocity, the particle's local information and global swarm information. Then, each particle's position is updated using the particle's new velocity. In mathematical terms the two update equations are:

$$v_{t+1} = w \times v_t + c_1 \times r_1 \times (p_t - x_t) + c_2 \times r_2 \times (g_t - x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Here w, c_1 and c_2 are weighting constants though variants of the algorithm allow them to change over time. r_1 and r_2 are random variables. p is personal best, and g is the global best. So, this combines the current velocity, a step in the direction of the personal best for each particle and a step towards the current global or swarm best. Choosing the weightings requires care, as we shall see.

The best positions can either be found synchronously or asynchronously, where best will be defined shortly. This paper presents results for synchronous updates, updating the global best after everything has moved.

> The algorithm above implements synchronous updates of particle positions and best positions, where the best position found is updated only after all particle positions and personal best positions have been updated. In asynchronous update mode, the best position found is updated immediately after each particle's position update. Asynchronous updates lead to a faster propagation of the best solutions through the swarm. [Dorigo08]

Unlike several other optimisation methods, this is 'gradientless'. For example, neural networks traditionally find their weights by using the differences between error functions for a change in their value and stepping the weights in the direction of the best value. The steps are always based on the gradient (difference per step size) [Wolfram].In contrast, PSO does not require any calculus to find gradients and use them to infer a step size and direction. In other words, no difficult maths is required to work out optimal ways to find the minimum or maximum of some function or best solution to a problem. We simply try a few things and remember the best so far.

Finally, we need a definition of 'best'. In our initial attempts, the particles were allowed to burst through the sides of the bag. In the case of PSO if we simply find the distance to the edge of the bag it is possible to have two equidistant particles with the current particle to be updated exactly in between them. In this case, it will not move towards either of them if both are allowed to exert influence. There are various ways to tackle this problem. I have decided to concentrate on the definition of 'best',

```
function best(first, second) {
  if (first.y > second.y) {
    return first;
  }
  return second;
}

function updateBest(item, bestGlobal) {
  var i;
  for (i = 0; i < item.length; ++i) {
    bestGlobal = best(item[i], bestGlobal);
    item[i].best = best(item[i].best, item[i]);
  }
  return bestGlobal;
}
```

Listing 4

providing a fitness function which, along with the approach taken in my genetic algorithm escapology, simply measures the distance to the top of the bag. This way all the particles will be given an imperative to move up. See Listing 4.

Note that the canvas has 0 at the top, but I have flipped things to have 0 at the bottom to fit with my mathematical bent. Bigger *y* coordinates are better, though the drawing code in the case must remember to flip the *u* coordinate back again:

```
var particle = item[i];
ctx.fillRect (particle.x,
              canvas.height - particle.y - 2,
              particle_size, particle_size);
```

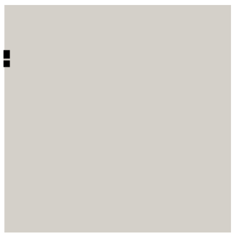
We leave a small gap, here 2, to give the particles space to actually come out of the bag. Care must be taken with the weights, otherwise positions can zoom off to infinity very easily – we need 'sensible' weights. Specifically, since the velocity will tend to make the particles move up due to the chosen fitness function, we can end up with exponential upwards motion. See Listing 5.

The **`move_in_range`** function simply clamps the particle to the edge of the bag or stops it when it peaks above the top. We could adapt the algorithm and allow it just to consider the best of its nearest neighbours rather than the global best, which could give us more of a flock than a swarm. In other words, you will see some overall shape of motion with a flock, rather than a mass of particles moving together in a swarm. We could also allow the particles to escape from the sides of the bag. There are several variants, but we shall just report the one approach outlined so far.

Unlike our first attempt, we can see all the particles tend to move together and escape in approximately the same place. They do tend to either all move left or all move right, which might indicate inappropriate weightings for the horizontal movement. Further work could be done to investigate the parameter choice.

```
function move(item, w, c1, c2, height, width,
bestGlobal) {
  var i;
  for (i = 0; i < item.length; ++i) {
    var current = item[i];
    var r1 = getRandomInt(0, 5);
    var r2 = getRandomInt(0, 5);
    var vy = (w * current.v.y) +
             (c1 * r1 *
                 (current.best.y - current.y)) +
             (c2 * r2 * (bestGlobal.y -
current.y));
    var vx = (w * current.v.x) +
         (c1 * r1 * (current.best.x - current.x)) +
         (c2 * r2 * (bestGlobal.x - current.x));
    move_in_range(vy, height, item[i], "y");
    move_in_range(vx, width, item[i], "x");
}
```

Listing 5



They do all consistently escape within a few seconds.

Conclusion

This article has considered how to program one's way out of a paper bag without needing an up-front model. This has some obvious advantages over simulations which require a believable model of how a situation might evolve over time. Particle swarm optimisations are part of the more general swarm intelligence algorithms, which allow a collection or swarm of potential solutions to a problem to collaborate, gradually nudging towards a better solution. Other examples include ant colony optimisations [Buontempo13b] or bee foraging algorithms [Quijano10]. In general, the 'points' explored will be values to solve another problem rather than spatial points, but hopefully this demonstration has served as a simple introduction for anyone who wishes to take this further. It would be nice to extend this to other swarming and flocking algorithms, perhaps having flights of birds or similar moving out of the bag. I will leave that as an exercise for the reader. ■

Notes

The code is on github at https://github.com/doctorlove/paperbag

Many thanks to my reviewers.

References

[Buontempo13a] Frances Buontempo 'What's a model' http://accu.org/content/pdf/presentations/accu2011nov/models.pdf

[Buontempo13b] Frances Buontempo (2013) 'How to program your way out of a paper bag' http://accu.org/content/conf2013/Frances_Buontempo_paperbag.pdf

[Buontempo13c] Frances Buontempo 'How to Program Your Way Out of a Paper Bag Using Genetic Algorithms' in *Overload* 118

[Dorigo08] Marco Dorigo, Marco Montes de Oca and Prof. Andries Engelbrecht 'Particle swarm optimization' in *Scholarpedia*, http://www.scholarpedia.org/article/Particle_swarm_optimization

[Freedman11] David Freedman 'Why economic models are always wrong' in *Scientific American*, article dated 26 October 2011 http://www.scientificamerican.com/article/finance-why-economic-models-are-always-wrong/

[k-NN] 'k-nearest neighbors alogorithm' – http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[Kennedy95] J. Kennedy and R. Eberhart 'Particle swarm optimization' in *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, IEEE Press, Piscataway, NJ, 1995 (See http://www.cs.tufts.edu/comp/150GA/homeworks/hw3/_reading6%201995%20particle%20swarming.pdf)

[Quijano10] Nicanor Quijano and Kevin Passino (2010) 'Honey bee social foraging algorithms for resource allocation: Theory and application' in *Engineering Applications of Artificial Intelligence* Volume 23, Issue 6, September 2010, Pages 845–861 (See http://www.sciencedirect.com/science/article/pii/S0952197610001090)

[Wolfram] 'Method of Steepest Descent' at *Wolfram MathWorld*: http://mathworld.wolfram.com/MethodofSteepestDescent.html

Feeding Back

Feedback can be positive and negative. Kevlin Henney contemplates how to make feedback useful.

A slim figure takes to the stage, dressed in orange and black, wreathed in a bandana, his guitar flipped over and strung for left-handed play. It's the close of over three days of hedonistic and culturally shifting psychedelia and sound. Humans have recently and for the first time set foot on another world.

It's 1969. It's Woodstock. It's Jimi Hendrix. During his set he splices metallic whale song into his fluid solos, coaxing sounds from his Stratocaster that guitars simply have no business making.

This is feedback. Not the negative feedback that dampens sound and enthusiasm. Positive feedback. But not gushing and uncontrolled, neither excessive nor insincere. There is an art to feedback.

Feedback, especially positive feedback, is normally a sound engineer's nightmare. A skilled guitarist can make it part of the performance, part of the music. For software engineers, offering and taking feedback, positive or negative, can be just as much a minefield. When there is a problem, it is too easy to resort to silence or complaint. When there isn't a problem, it is too easy to resort to silence.

When you ride a bicycle, feedback is essential. Sight, hearing and proprioception allow you to navigate and balance, to respond to the bike and the road. You respond when the bike is balanced and on a steady course: you respond by continuing to do what you are doing, preserving your course and your balance. You respond when the bike loses balance, destabilized perhaps by a hole or a bump. You change behaviour, you react to recover and put the bike back on course. And you respond when the situation on the road changes. You avoid pedestrians, cars and other bikes, stop at junctions and red lights, cycle more carefully in the rain.

Part of team leadership involves leading by example, but part involves guidance. For simple systems, guidance is programmatic, a matter of command and control. This doesn't work well for complex systems, and individuals and teams are very complex systems indeed. Feedback is a guidance technique, but there is an art to it that goes beyond the simple presentation of the facts as you see them. To be effective, feedback also needs to be trusted, concrete and constructive.

No matter how upstanding they might be, we do not generally consider people to be objective sources of information in the way that inanimate objects and software tools are. When a piece of code fails a test or doesn't compile, we do not attribute this to a subjective judgement of the test or the emotional state of the compiler (unless we're having a really bad day). When we get feedback from people we are more likely to hear what they are saying through a veil of emotions, cognitive biases and relationships.

If the only feedback you offer is negative and corrective, it is likely to dampen anyone's spirits, independently of whether or not it is factually correct. Negative feedback is likely to breed mistrust and resentment. It is disempowering and demotivating. The absence of any positive feedback, by implication, suggests that there is nothing the person is doing right.

Relentless feedback of any one form does not offer the guidance or build the trust that will help you, the individual or the team. This applies just as much to unconditional positive feedback as it does to negative feedback. Positive feedback is psychologically necessary, otherwise people feel like they're operating in a vacuum — the few humans who have ever been privileged to work in a literal rather than figurative vacuum know that support is a necessity not an option — but there is a balance to be struck: excessive and unwarranted positive feedback simply becomes saccharine and insincere.

Feedback should also be contextual and concrete. Simply saying someone's work is good is a pat on the back, but it's vague and there is little guidance, little they can take away from it beyond feeling congratulated. What is it that is good? Whether we are talking about someone overcoming a personal or technical challenge, meeting a goal or fielding ideas, be specific. Unless you are specific, it is difficult for them to know what it is about what they did that is good so that they can learn from it, repeat it and build on it.

It is this question of learning and allowing someone else to do the learning that highlights the weakness of negative feedback. Even without the question of self-esteem, simply pointing out that something is not good is not helpful, and in this case adding detail doesn't help. Just saying something is not good does not tell someone what is good. There is little they can learn from it. It's like a no-entry sign on a one-way street: you are told which way you should not go, but you are not told which way you should go.

Negative feedback is often given in response to seeing a problem, but it is not intrinsically problem solving and constructive. To be constructive you need to offer a concrete suggestion for improvement or you need to make the giving of feedback part of a problem-solving conversation. If you want someone to learn, create the opportunity and environment for them to discuss and contribute, otherwise the feedback becomes more about the person giving the feedback than the person receiving it. Feedback should have purpose and it should enable purpose.

Feedback, as a term, is often taken to be unidirectional but, as its engineering origins suggest, it is definitely about a relationship. It involves guidance and balance. Steady as she goes. ■

This is based on material previously published in Roy Osherove's *Notes to a Software Team Leader*: https://leanpub.com/teamleader

Kevlin Henney Kevlin is an independent consultant and trainer with an interest in programming, patterns, practice and process. He has been a columnist for a number of magazines and web sites, not all of which have folded, and is the co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*, two volumes in the *Pattern-Oriented Software Architecture* series. He is also the editor of *97 Things Every Programmer Should Know*. Kevlin speaks at conferences, lives online and in transit, and writes short fiction in what perhaps qualifies as his spare time.