contents

Copy Deadlines

All articles intended for publication in *Overload 72* should be submitted to the editor by March 1st 2006, and for *Overload 73* by May 1st 2006.

credits & contacts

Overload Editor:
Alan Griffiths
overload@accu.org
alan@octopull.demon.co.uk

Contributing Editor:
Mark Radford
mark@twonine.co.uk

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Advertising:
Thaddaeus Frogley
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: Keeping Up Standards

"Nobody made a greater mistake than he who did nothing because he could do only a little." - Edmund Burke.

There are many conventions that are required to make modern life possible. They take many forms – from social rules, through voluntary agreements and contractual arrangements to legislation. There are also those who seek to circumvent, subvert or exploit these conventions. This implies that there are benefits, real or imagined, from ignoring the conventions – so what is the value of them?

Let's take one example that we are all familiar with: connecting to mains power. This involves a large number of arbitrary choices: voltage, frequency, the shape and materials of the plugs and sockets. The benefits of being able to take any appliance and plug into any socket are obvious. Less obvious are both the mechanisms and activities that support this – the standards process – and the market it creates in compatible components. The former is the cost that suppliers pay to participate in the market, and competition in the market drives improvements in quality and price. Clearly suppliers would like to avoid both the costs of validating conformance to standards and the need for competition, but society (in the form of legislation) has ensured that they cannot access the market without this.

As software developers we all come across standards as part of our work – almost all of you will be working with ASCII or one of the Unicode supersets of it. (Yes, there is also EBCDIC.) Maybe some of you can remember the time before character sets were standardised, and computer manufacturers just did their own thing. The others will have to imagine the additional effort this once required when a program needed to accept data from a new source. Standards didn't solve the whole problem – for example, ASCII had an alternative interpretation of code `0x23` as "national currency symbol" which has largely fallen into disuse since the advent of "extended ASCII" which contains enough "national currency symbols" to handle European countries as well as the USA.

Another implication of differing character sets is that there are occasionally characters represented in one that were not representable in another (for example, the "{" and "}"curly brace characters beloved of the C family of languages are not present in EBCDIC). This means that workarounds are required – for text intended for human consumption this may be as simple as mapping these characters to some arbitrary token (such as a "?"), but for some uses (like the source code of computer programs) more automation is required – hence the C and C++ support for trigraphs and digraphs.

Despite these problems, having a common standard (or two) for interpreting and transmitting text solves a lot of problems and helped make communication between computers a commonplace. Nowadays computers can be plugged into the "world wide web" with almost the same ease that appliances can be plugged into the mains. There is less regulation and certification than with electrical power (and probably more standards), but by and large it does work.

IT standards come in a wide variety of forms. They vary in source: there are bodies that exist to ratify standards like IEEE, ISO and ECMA; there are industry consortia set up to standardise a specific area like Posix, W3C and Oasis; and there are standards published and maintained by a specific vendor like IBM, Microsoft or Sun. They vary in terms: some are financed by charging for access to the standard and some don't charge; some require (and charge for) validation and some don't – there are even some with legislative backing!

My last editorial related to the C++ language standard and the unfortunate treatment it has received at the hands of a major vendor of C++. What I wasn't expecting at the time was that it would sound so familiar to a number of of correspondents. In fact, as I gathered the stories that I'm about to relate together I began to feel that Bonaparte's maxim "never ascribe to malice that which can adequately be explained by incompetence" was being stretched beyond credibility. I now think that Microsoft has either undervalued consensus in the standardisation process, or overestimated the extent to which the its needs are a guide to the needs of the wider community. But please don't take my word for it: decide that for yourself.

Exhibit 1

Many of you will have heard of "C++/CLI" – this is an ECMA standard that originates with Microsoft and has recently been submitted to ISO for a "Fast Track Ballot". The BSI working group responding to this submission, and the following quotes come from this response:

At the time this project was launched in 2003, participants described it as an attempt to develop a "binding" of C++ to CLI, and a minimal (if still substantial) set of extensions to support that environment. C++/CLI is intended to be upwardly compatible with Standard C++, and Ecma TG5 have gone to praiseworthy efforts to guarantee that

standard-conforming C++ code will compile and run correctly in this environment.

Nevertheless, ***we believe C++/CLI has effectively evolved into a language which is almost, but not quite, entirely unlike C++ as we know it. Significant differences can be found in syntax, semantics, idioms, and underlying object model.*** *It is as if an architect said, "we're going to bind a new loft conversion on to your house, but first please replace your foundations, and make all your doors open the other way."* ***Continuing to identify both languages by the same name*** *(even though one includes an all-too-often-dropped qualifier)* ***will cause widespread confusion and damage to the industry and the standard language.***

...

Standard C++ is maintained by WG21, the largest and most active working group in SC22. WG21 meetings, twice a year lasting a week at a time, draw regular attendance by delegates from a number of national bodies and nearly all the important vendors of C++ compilers and libraries, plus a number of people who use the language in their work. By contrast, this ECMA draft was developed by a small handful of people — awesomely competent ones, undoubtedly, but who do not represent the interests of the broad market of vendors and users. With ISO/IEC 14882 maintained by JTC 1 SC 22 WG 21 and C++/CLI maintained by ECMA, the differences between Standard C++ and the C++/CLI variant will inevitably grow wider over time. The document proposes no mechanism for resolving future differences as these two versions of C++ evolve.

For JTC1 to sanction two standards called C++ for what are really two different languages would cause permanent confusion among employers and working programmers.

There is clear evidence that this confusion already exists now...

...

Documentation for Microsoft's Visual C++ product contains many code examples identified as "C++" — NOT "C++/CLI" or even "C++.Net" — which will fail to compile in a Standard C++ environment.

...

C++ already has a reputation as a complicated language which is difficult to learn and use correctly. C++/CLI incorporates lip-service support for Standard C++ but joins to it in shotgun marriage a complete second language, using new keywords, new syntax, variable semantics for current syntax, and a substantially different object model, plus a complicated set of rules for determining which language is in effect for any single line of source code.

If this incompatible language becomes an ISO/IEC standard under the name submitted, it will be publicly perceived that C++ has suddenly become about 50% more complex. The hugely increased intellectual effort would almost certainly result in many programmers abandoning the use of C++ completely.

...

A parallel to this situation can be found in the history of C++ itself. As related by Bjarne Stroustrup in The Design and Evolution of C++, the language in its early days was known as "C with Classes", but he was asked to call it something else: "The reason for the naming was that people had taken to calling C with Classes 'new C' and then C. This abbreviation led to C being called 'plain C', 'straight C' and 'old C'. The last name, in particular, was considered insulting, so common courtesy and a desire to avoid confusion led me to look for a new name."

(Full response at: `http://www.octopull.demon.co.uk/editorial/N8037_Objection.pdf`.)

In short, ISO – the body that standardised C++ is being asked to ratify a "fork" of C++, and the BSI panel is raising some concerns about this. Forking isn't of itself evil: there are often good reasons to fork pieces of work – and on occasions good things come of it. Some of you will remember the edcs fork of gcc – this was a major reworking of the C++ compiler that was infeasible within the normal gcc schedule. This work has since become the basis of current gcc versions. Another, long-lived and useful fork exists between emacs and xemacs – these projects co-exist happily and frequently exchange ideas.

However, in these felicitous cases of forking those concerned were careful to make it clear what they were doing and why. In the case of C++/CLI it is clear that having a language with the power of C++ on the .NET platform is a good thing for those interested in the platform, and it is also clear that the CLR provides many features that cannot be accessed without language support (yes, efforts were made to find a library-based "mapping").

In short, while there may be no explicit rules or laws being broken by Microsoft's promotion of C++/CLI as "Pure C++" (`http://msdn.microsoft.com/msdnmag/issues/05/12/PureC/default.aspx`) it is undoubtedly antisocial.

Exhibit 2

In an earlier editorial I alluded briefly to the notice given by the Commonwealth of Massachusetts' administration that, from 2007, they intended to require suppliers of office applications to support the OpenDocument standard. For those that don't know, OpenDocument is an XML based standard for office applications and has been developed by Oasis (a consortium including all the major players) which has submitted it to ISO for ratification.

Although Microsoft joined Oasis it is apparent that they didn't participate in developing the standard that was produced. I don't know the reasons for this but, inter alia, it is clear that they disagreed with the decision to take an existing, open, XML based formats as a starting point: including that used by both StarOffice and OpenOffice. (Although the resulting OpenDocument format differs substantially from the original OpenOffice format statements from Microsoft continue to refer to it as "OpenOffice format".)

Anyway, after discovering that their own XML based formats didn't meet the eventual criteria that Massachusetts developed during an extended consultation period (and that these criteria were unlikely to change) Microsoft decided to standardise its own XML based formats through ECMA as "Office Open format". (The terms of reference for the ECMA working group are interesting – by my reading it seems that the group doing the standardisation don't have the authority to make changes to address problems discovered in the format!)

The intent is for ECMA to submit "Office Open format" to ISO for "Fast Track submission".

I don't have the expertise to make a technical assessment of alternative standard formats for office documents – especially when one has yet to be published. But when a major customer (Massachusetts) and a range of suppliers, including both IBM and Sun, agree on a standard I think it should be given serious consideration. It is easy to see that OpenDocument is already being adopted and supported by applications suppliers: `<http://en.wikipedia.org/wiki/List_of_applications_supporting_OpenDocument>`).

By playing games with names, Microsoft trivialise the discussion to the point where I doubt that there is any merit to their claim that OpenDocument is seriously flawed or that Office Open format (when it arrives) will be better.

Exhibit 3

In my last editorial, talking about about Microsoft's non-standard "Safe Standard C++ Library" I wrote: *The Microsoft representatives have indicated that the parts of this work applicable to the C standard have already been adopted by the ISO C working group as the basis for a 'Technical Report'*. Since then I've had my attention drawn to the following comments by Chris Hills (who was convener of BSI's C panel around that time):

> *Microsoft are pushing for a new "secure C library" (See* `http://std.dkuug.dk/jtc1/sc22/wg14/www/docs/n1007.pdf` *and* `http://std.dkuug.dk/jtc1/sc22/wg14/www/docs/n1031.pdf`*) for all the library functions, apparently all 2000 of them. I did not think there were 2000 functions in the ISO-C library but MS have included all the MS C/C++ libraries as well in this proposal, which is of no use to the vast majority in the embedded world.*
>
> *The problem for me is that the resultant libraries would be full of MS specific extensions. The trust of the proposal is that there are many holes and leaks in the original libraries that permit buffer over runs, error reporting and parameter validation. Security is the important thing here they stress. One of my BSI panel said that voting against security is like "voting against Motherhood and Apple Pie". However, there is quite some unease on the UK panel re this proposal.*
>
> *The other complaint MS have in their proposal is that the library was designed when computers were "Much simpler and more constrained". This is a common comment from PC programmers who will tell you 16 bit systems died out a while ago and there has not been an 8-bit system since Sinclair or the BBC home computers.*
>
> `http://www.phaedsys.org/papersese0403.html`

This doesn't sound quite like the support for the TR implied by Microsoft's account – and I don't know what has actually happened at the WG14 meeting when this was discussed (maybe I will by next time). However, these ISO groups are manned by individuals and organisations that volunteer their time: if someone volunteers to write a 'Technical Report' then even the most negative response is likely to be something like "it doesn't interest me" – so Microsoft may have talked only to those interested in their idea, not those that thought it misguided. This could have led to an incorrect impression regarding the level of support for their proposal. (Or Chris may have got it wrong – I gather he was unable to attend the ISO meeting where this proposal was discussed.)

We should see later this year if WG13 accepts this 'Technical Report' but, even if that happens, there are some members of the working group that do not anticipate this report becoming a significant input to a future C standard.

Exhibit 4

As I reported on this last time Microsoft has unilaterally decided various usages that a sanctioned by the standard should be reported as 'deprecated' by their C++ implementation and replaced with other non-standard (and non-portable) usages of their own. Beyond saying that the discussions between Microsoft and other WG21 members are ongoing I won't go into more detail at present.

Exhibit n

The above exhibits are not an exhaustive list, I've heard disgruntled remarks about Microsoft's implementation of Kerberos authentication, their implementation of HTTP, their approach to standardising .NET and their work with the SQL Access Group. Such remarks may or may not be justified – I don't know enough about any of these subjects to make informed comment.

Conclusion

Conforming to standards and regulations can be irritating and inconvenient and some, like speed limits, are widely violated – to the extent that 'being caught' is frequently considered the nuisance, not the miscreant. (Other standards, such as prohibitions against kidnapping, are held in higher esteem.)

Part of what governs our attitude to standards is the difference between what we get out of it and the inconvenience it imposes on us. Getting to bed half an hour earlier after a long journey often seems attractive compared to the marginal increase in risk that speeding introduces. (On the other hand, the temptations of kidnapping are, to me at least, more elusive.)

In many regards Microsoft's determination to ensure that standard C++ code works in C++/CLI is a demonstration of respect for the C++ standard. On the other hand, by their actions they appear to hold the process of standardisation, or the needs of other participants, in low regard. On the gripping hand, these standardisation processes are not meeting the needs of an important supplier – is it the process or the supplier that is out of step?

Alan Griffiths
overload@accu.org

Soft Documentation

by Thomas Guest

Introduction

Recently I spent some time working on a user manual.

The existing version of this manual was based on a Microsoft Word [1] master document. From this master the various required output formats were generated in a semi-automated fashion.

I'm guessing anyone who's used a computer will have come across Microsoft Word: it's a popular tool which is easy to get started with and which, by virtue of its WYSIWYG interface, allows even a novice to produce stylish output. It does have its drawbacks though, especially for technical documentation, and these drawbacks were only amplified by the other tools involved in producing the final deliverables.

We'll look more closely at these drawbacks later. I summarise them here by saying the proprietary tools and file formats led to a loss of control. The final outputs were not so much WYSIWYG as WYGIWYG – What You Get is What You're Given.

Producing high quality technical documentation is a difficult problem but it's also a problem which has been solved many times over. Plenty of open source projects provide model solutions. My increasing frustration with the Microsoft Word based documentation toolchain led me to explore one of these alternatives.

This article records the outcome of my exploration. It tells how, in the end, we did regain control over the manual, but at a price.

Requirements

The requirements for the manual were clear enough. It had to look good. It had to fit the corporate style – dictating, in this case, font families, colour schemes, logos and various other presentational aspects. There would be pictures. There would be screen shots. There would be cross references.

Naturally, the contents should provide clear and complete details on how to use the Product.

We needed just two output formats:

- hard copy, printed and bound
- linked online web pages.

Of course, these two versions of the document would have to agree with each other. And the Product itself, a server-based piece of software with a web browser interface, should integrate with the online documentation in a context-sensitive manner: clicking the Help icon next to an item in the UI should pop up the manual opened at the correct location.

Finally, there was the slightly strange requirement that the documentation should be substantial. Somehow, it seemed unreasonable to ask customers to hand over lots of money for nothing more than CD's worth of software; bundling in a weighty manual made the final deliverables more tangible. [1]

The Existing Documentation Toolchain

The existing toolchain was, as already mentioned, based on a Microsoft Word master document.

Producing hard copy was as simple as CTRL+P, followed by a dialog about printer settings and some manual labour involving a ring binder. It's fair to say that the printed output looked pretty much exactly as previewed: the author had good control over pagination, positioning of images, fonts, colours and so on.

The linked online pages took more effort. We'd got a license for a tool which I'll call Word Doctor (not its real name – I'm using an alias because I'm going to moan about it). Generating the linked web pages using Word Doctor involved the following steps:

1. Create a new Project.
2. Point it at the Microsoft Word Master.
3. Select some project options from the Word Doctor GUI.
4. Click the build button (experts, hit 'F5').
5. Make a cup of tea while the pages generate.

All fairly easy – in theory. In practice, there were some other steps which the Word Doctor user manual neglected to mention:

- Exit Microsoft Word. Word Doctor has trouble accessing the document otherwise.
- Restart your PC. For some reason a resource got terminally locked up.
- Rewrite the Microsoft Word master using the Word Doctor document template.
- Don't forget to exit Microsoft Word!
- Create a new project etc.
- Click the build button.
- Click away a few warnings about saving TEMPLATE.DOT and OLE something or other.
- Read the Word Doctor workarounds Wiki page on the intranet.
- Click the build button again.
- Go for lunch. Documentation builds took around half an hour.

I am not exaggerating. The engineering manager admitted that he estimated it took at least two days of struggling to convert a Microsoft Word master into the online form. And nor do I blame Word Doctor. I don't think Microsoft Word comes with a decent developer API. Instead, it tries to do everything itself: from revision control, through styling, to HTML output. It uses an opaque binary file format to deter anyone from trying to develop tools to work with it.

The final irritation was with the Word Doctor output – if you ever got any. The HTML was packed with Internet Explorer specific Javascript, and looked poor in any other browser.

Connecting up to Word Doctor Output

The real downside of Word Doctor was when it came to trying to connect the Product to the Word Doctor web pages. This job fell to me. It was a multi-layered integration task:

- on a team level I would work with the technical author to ensure the documentation content was correct, and contained the required Help topics.
- on the Product side, the web-based user interface would call for help using a text identifier. The Help subsystem would use the identifier to look up an HTML location – a page and an anchor within that page – and it could then pop up a new window viewing this location.
- on the documentation side, I would have to configure Word Doctor to ensure its HTML output included the right locations.

1 This, to me, is a suspect requirement, or at least one we should keep in check, otherwise we run the risk of producing documentation whose sections are cut-and-paste adaptations of similar sections.

Unfortunately, there were problems with each of these layers.

Personally, I got on well with the technical author, but the documentation tools made it extremely hard for us to work on the same file. We had to take it in turns or work with copies. I couldn't even fix a typo directly.

The Word Doctor output was a frame-based collection of static HTML pages. Now, externally referencing a particular location in such a set of pages is tricky – due to the limitations of frames – so the Product's help sub-system had to dynamically generate a framed front page displaying the appropriate left and right pane each time it was called. Not too difficult, but more complex than strictly necessary.

Both pages and anchors were a moving target in the Word Doctor output. Every time you added a new section to the document you broke most of the help references. Thus we found ourselves in a situation where the technical author wanted the Product to stabilise in order to document it and I needed the documentation to stabilise in order to link to it.

Other Problems

Microsoft Word uses a proprietary binary format. This ties you into their product to a degree – effectively, you're relying on Microsoft to look after your data because you simply cannot work with this data without their tool. Of course, the risk of Microsoft collapsing during the lifetime of your document may be one you can live with, but you are also vulnerable to them ceasing to support the version of Word you prefer, or charging an unreasonable amount for an upgrade. It also means:

- it's extremely hard for more than one person to work on a document at a time since changes to binary files cannot be merged together easily.
- revision control becomes more expensive and less useful (how do you view the differences between two versions of the manual?)
- it is very difficult to automate anything. As a trivial example, Word Doctor had no batch interface – it required human input at every stage. Now consider trying to rebadge the manual, perhaps for redistribution of the Product by some partner company. With a decent documentation toolchain this should be as simple as the build 'prepare' target copying the correct logo graphic into place and applying a simple transformation to some text strings.

Resistance to Change

Despite all of these limitations and irritations it was hard to convince anyone a change was necessary or even desirable. The reasons were as much organisational as technical.

- The existing tools had been used to produce acceptable end user documentation in the past for other products shipped by the company.
- Already, considerable effort had been put into the Word master for the new Product (even if much of it would have to be scrapped due to the inevitable changes as the Product developed).
- The engineering team had more work than it could cope with already. At least the user documentation could be outsourced to a contract technical author.
- Setting up a smarter toolchain would need engineering input and, once the tools were in place, would the technical author be able to use them productively?
- The sales team saw the documentation task as non-urgent for much the same reason that they saw user input validation as a nice-to-have rather than a priority. After all, they'd run some promising beta trials at customer sites using a poorly documented and inputs-unchecked version of the Product. They were happy to continue to provide support and tuition as required, either on site, by phone or by email.

I could (and did) argue against all of these points:

- existing documentation was stand-alone: it did not have to integrate with what it documented. Using the existing tools to connect the new Product with its documentation looked like being a continual sink of effort.
- The engineering team probably spent as long telling the technical author what to write as they might have spent writing it themselves.
- Surely the technical author would quickly master a new documentation tool?
- In fact it was more often the engineers than the sales team who provided support, and frequently for problems which could have been avoided with better input checking and more solid documentation.

As software engineers we need to concentrate on the software. That means listening to the sales team; but when it comes to software quality, we know best. I believe the only shortcut is to prune back the feature list and, increasingly, I regard it as wrong to view software documentation as an add-on. Decent documentation is one of the first things I look for when I evaluate a piece of software: the website, the user interface, the README, the FAQ list, and of course the source code itself (if available). Quite simply, I didn't want to deliver a Product with poor documentation. I didn't think it would save us time in the short or long term.

Regaining Control

My frustration with the existing documentation tools set me thinking about alternatives. I looked first to the open source world (I'm using the term loosely here), where there's no shortage of excellent documentation and where the authors are happy to show how they generated it.

I experimented by downloading and attempting to build some open source documentation. This was a part time activity, squeezed into moments when I was waiting for builds to complete or files to check out. If the documentation didn't build or required lots of configuration to get it to build, I moved on.

I was looking for something as simple as:

```
> cd docs ; make
```

To my surprise and disappointment it took several attempts to find something which worked out of the box. Perhaps I was unlucky. No doubt in many cases it was user error on my part and no doubt I could have sought advice from email lists; nonetheless, I kept moving on until I found something which worked first time (my thanks to the Hibernate documentation team [2]). Then I continued to experiment: could I change fonts, include images, replicate the house style? How easy were the tools to use with our own content?

After a Friday afternoon's experimentation I had something worth showing to the engineering manager: an end-to-end solution

which, from a DocBook XML master, generated a skeleton PDF and HTML user manual in something approaching the house style. I suggested to the engineering manager that we should switch the user manual to use the tools I had just demonstrated. I said I'd be happy to do the work. He agreed with me that technically, this seemed the way forwards. However, it wasn't easy for him to give me the go ahead for the reasons already discussed.

Also, it was a hard sell for him to make to the rest of the company: on the one hand, writing end user documentation simply wasn't what the engineers were supposed to be doing; and on the other, it was hard enough persuading the technical author to use the revision control system, let alone edit raw XML.

I confess I had my own doubts too. All I knew at this stage was that DocBook could do the job and that I would happily tinker with it to get it working. I didn't know if I could be productive using it. I don't relish editing XML either.

We both recognised that the single most important thing was content. Full and accurate documentation supplied as a plain README would be of more practical use to our customers than something beautifully formatted and structured but misleadingly inaccurate.

In the end we deferred making a final decision on what to do with the manual.

The results of my experiment did seem worth recording, so I spent a day or so tidying up and checking in the code so we could return to it, if required.

A DocBook Toolchain

I should outline here the basics of the toolchain I had evaluated. It was based on DocBook [3]. A two sentence introduction to DocBook can be found on the front page of the SourceForge DocBook Project [4]. I reproduce it here in full:

> *DocBook is an XML vocabulary that lets you create documents in a presentation-neutral form that captures the logical structure of your content. Using free tools along with the DocBook XSL stylesheets, you can publish your content as HTML pages and PDF files, and in many other formats.*

I would also like to highlight a couple of points from the preface to Bob Stayton's *DocBook XSL: The Complete Guide* [5] – a reference which anyone actually using DocBook is sure to have bookmarked:

> *A major advantage of DocBook is the availability of DocBook tools from many sources, not just from a single vendor of a proprietary file format.*
>
> *You can mix and match components for editing, typesetting, version control, and HTML conversion.*
>
> *...*
>
> *The other major advantage of DocBook is the set of free stylesheets that are available for it... These stylesheets enable anyone to publish their DocBook content in print and HTML. An active community of users and contributors keeps up the development of the stylesheets and answers questions.*

So, the master document is written in XML conforming to the DocBook DTD. This master provides the structure and content of our document. Transforming the master into different output formats starts with the DocBook XSL stylesheets.

Various aspects of the transformation can be controlled by setting parameters to be applied during this transformation (do we want a datestamp to appear in the page footer?, should a list of Figures be included in the contents?), or even by writing custom XSL templates (for the front page, perhaps).

Depending on the exact output format there may still be work for us to do. For HTML pages, the XSL transformation produces nicely structured HTML, but we probably want to adjust the CSS style sheet and perhaps provide our own admonition and navigation graphics. For Windows HTML Help, the DocBook XSL stylesheets again produce a special form of HTML which we must then run through an HTML Help compiler.

PDF output is rather more fiddly: The DocBook XSL stylesheets yield XSL formatting objects (FO) from the DocBook XML master. A further stage of processing is then required to convert these formatting objects into PDF. I used the Apache Formatting Objects Processor (FOP) [6], which in turn depends on other third-party libraries for image processing and so on.

Presentation and Structure

A key point to realise when writing technical documentation is the distinction between structure and presentation. Suppose, for example, our document includes source code snippets and we want these snippets to be preformatted in a monospaced font with keywords emphasized using a bold font style. Here, we have two structural elements (source code, keywords) and two presentational elements (monospaced font, bold style).

Structure and presentation are separate concerns and our documentation chain should enable and maintain this distinction. This means that our master document structure will need to identify source code as "source code" – and not simply as preformatted text – and any keywords within it as "keywords"; and the styling associated with the presentation of this document will make the required mapping from "source code" to "monospace, preformatted" and from "keyword" to "bold"".

We can see this separation in well-written HTML where the familiar element tags (HEAD, BODY, H1, H2, P etc) describe basic document structure, and CLASS attributes make finer structural distinctions. The actual presentation of this structured content is controlled by logically (and usually physically) separate Cascading Style Sheets (CSS).

With a WYSIWYG documentation tool presentation and structure – by definition – operate in tandem, making it all too easy to use a structural hack to fix a presentational issue (for example, introducing a hard page break to improve printed layout, or scaling a font down a point size to make a table look pretty).

DocBook enforces the separation between structure and presentation strictly. This doesn't mean that we can't use a Graphical Editor to work with DocBook documents – indeed, a web search suggests several such editors exist. I chose to work with the raw DocBook format, however, partly because I could continue to use my preferred editor [7] and partly because I wanted to get a better understanding of DocBook. The enforced separation can sometimes be frustrating, however. It took me about an hour to figure out how to disable hyphenation of the book's subtitle on my custom frontpage!

As we can see, there are choices to be made at all stages: which XSL transform software do we use, which imaging libraries; do we go for a stable release of Apache FOP or the development rewrite? Do we spend money on a DocBook XML editor? Since we have full source access for everything in the chain we might also choose to customise the tools if they aren't working for us.
These choices were, to start with, a distraction. I was happy to go with the selection made by the Hibernate team unless there was a good reason not to. I wanted the most direct route to generating decent documentation. I kept reminding myself that content was more important than style (even though the styling tools were more fun to play with).

The Technical Author Departs

We continued on, then, deferring work on the documentation until at least we had frozen the user interface, still pinning our hopes on Word Doctor. Then the technical author left. She'd landed a full-time editing position on a magazine.

Again, I volunteered to work on the documentation. By now the engineering manager had succeeded in selling the idea of switching documentation tools to higher management. It was still hard for him to authorise me to actually write the documentation, though, since we had just recruited a new technical support engineer, based in North America. This engineer had nothing particular lined up for the next couple of weeks. What better way for him to learn about the Product than to write the user manual?

As it turned out it various delayed hardware deliveries meant it took him a couple of weeks to set up a server capable of actually running the Product – and then he was booked up on site visits. He didn't get to spend any time on documentation.

Version 1.0 was due to be released in a week's time. We had four choices:

- Ship with the existing documentation – which was dangerously out of date.
- Stub out the documentation entirely, so at least users wouldn't be misled by it.
- Revise the Microsoft Word document, use Word Doctor to generate HTML, reconnect the HTML to the Product.
- Rewrite the manual using DocBook.

We ruled out the first choice even though it required the least effort. The second seemed like an admission of defeat – could we seriously consider releasing a formal version of the Product without documentation? No-one present had any enthusiasm for the third choice.
So, finally, with less than a week until code freeze, I got assigned the task of finishing the documentation using the tools of my choosing.

Problems with DocBook

Most things went rather surprisingly well, but I did encounter a small number of hitches.

Portability

My first unpleasant surprise with the DocBook toolchain came when I tried to generate the printable PDF output on a Windows XP machine. Rather naively, perhaps, I'd assumed that since all the tools were Java based I'd be able to run them on any platform with a JVM. Not so.

The first time I tried a Windows build, I got a two page traceback (see Figure 1) which sliced through methods in `javax.media.jai`, `org.apache.fop.pdf`, `org.apache.xerces.parsers`, arriving finally at the cause.

I had several options here: web search for a solution, raise a query on an email list, swap out the defective component in the toolchain, roll up my sleeves and debug the problem, or restrict the documentation build to Linux only.

I discovered this problem quite early on, before the technical author left – otherwise the Linux-only build restriction might have been an acceptable compromise; several other Product components were by now tied to Linux. (Bear in mind that the documentation build *outputs* were entirely portable, it was only the build itself which didn't work on all platforms). My actual solution was, though, another compromise: I swapped the Java JAI [8] libraries for the more primitive JIMI [9] ones, apparently with no adverse effects.

The incident did shake my confidence, though. It may well be true that open source tools allow you the ultimate level of control, but you don't usually want to exercise it! At this stage I had only tried building small documents with a few images. I remained fearful that similar problems might recur when the manual grew larger and more laden with screenshots.

Volatility

We all know that healthy software tools are in active development, but this does have a downside. Some problems actually arose from the progression of the tools I was using. For example, I started out with the version of the DocBook XSL stylesheets I found in the Hibernate distribution (version 1.65.1). These were probably more than good enough for my needs, but much of the documentation I was using referred to more recent distributions. In this case, fortunately, switching to the most recent stable distribution of the XSL stylesheets resulted in improvements all round. Apache FOP is less mature though: the last stable version (as of December 2005) is 0.20.5 – hardly a version number to inspire confidence – and the latest unstable

```
Caused by: java.lang.IllegalArgumentException: Invalid ICC Profile Data
  at java.awt.color.ICC_Profile.getInstance(ICC_Profile.java:873)
  at java.awt.color.ICC_Profile.getInstance(ICC_Profile.java:841)
  at java.awt.color.ICC_Profile.getDeferredInstance(ICC_Profile.java:929)
  at java.awt.color.ICC_Profile.getInstance(ICC_Profile.java:759)
  at java.awt.color.ColorSpace.getInstance(ColorSpace.java:278)
  at java.awt.image.ColorModel.<init>(ColorModel.java:151)
  at java.awt.image.ComponentColorModel.<init>(ComponentColorModel.java:256)
  at com.sun.media.jai.codec.ImageCodec.<clinit>(ImageCodec.java:561)
  ... 34 more
```

Figure 1: Traceback following an attempted build on Windows XP

```
<section id="hello_world">
  <title>Hello World</title>
  <para>
  Here is the canonical C++ example program.
  </para>
  <programlisting>
  <![CDATA[
  #include <iostream>
  int main() {
   std::cout << "Hello world!" << std::endl;
       return 0;
    }
    ]]>
    </programlisting>
</section>
```

Figure 2: A section of DocBook document.

release, 0.90 alpha 1, represents a break from the past. I anticipate problems if and when I migrate to a modern version FOP, though again, I also hope for improvements.

Verbosity

XML is verbose and DocBook XML is no exception. As an illustration, Figure 2 shows a section of a DocBook document.

XML claims to be human readable, and on one level, it is. On another level, though, the clunky angle brackets and obtrusive tags make the actual text content in the master document hard to read: the syntax obscures the semantics.

Control

The DocBook toolchain gave us superb control over some aspects of the documentation task. In other areas the controls existed but were tricky to locate and operate.

For example, controlling the chunking of the HTML output was straightforward and could all be done using build time parameters – with no modifications needed to the document source. Similarly, controlling file and anchor names in the generated HTML was easy, which meant the integration between the Product and the online version of the manual was both stable and clean.

Some of the printed output options don't seem so simple, especially for someone without a background in printing. In particular, I still haven't really got to grips with fine control of page-breaking logic, and have to hope no-one minds too much about tables which split awkwardly over pages.

The Rush to Completion

In the end, though, all went better than we could have hoped.

I soon had the documentation build integrated with the Product build. Now the ISO CD image had the right version of the User Manual automatically included.

I wrote a script to check the integration between the Product and the User Manual. This script double-checked that the various page/anchor targets which the Product used to launch the pop up Help window were valid. This script too became part of the build. It provided a rudimentary safety net as I rolled in more and more content.

Next, I cannibalised the good bits of the existing manual. We knew what problems we had seen in the field: some could be fixed using better examples in the Help text; I fixed these next. Within a couple of days the new manual had all the good content from the old manual and none of the misleading or inaccurate content; it included some new sections and was fully linked to the Product. It was, though, very light on screen shots.

Screen Captures

In an ideal world we could programatically:

- launch the Product;
- load some data;
- pose the user interface for a number of screen shots;
- capture these screen shots for inclusion in the documentation.

Then this program too could become part of the build and, in theory, the screen shots would never fall out of step with the Product.

Already we had some tools in place to automate data loading and to exercise the user interface. We still have no solution for automatically capturing and cropping the images, so we rely on human/GIMP intervention. So far, this hasn't been a huge issue.

QuickBook

I had a workaround for the verbosity issue. I used QuickBook [10], one of the Boost tools [11]. QuickBook is a lightweight C++ program which generates DocBook (BoostBook, strictly speaking[2]) XML from a WikiWiki style source document.

Using QuickBook, we can write our previous example as:

```
[section Hello World]
Here is the canonical C++ example program.
  #include <iostream>
  int main() {
     std::cout << "Hello world!" << std::endl;
     return 0;
  }

[endsect]
```

QuickBook documents are easy to read and easy to write. QuickBook does fall a long way short of the full expressive richness of DocBook but if all you need are sections, cross-references, tables, lists, embedded images and so on, then it's ideal.

You can even escape back to DocBook from QuickBook. So if you decide your manual needs, for example, a colophon, you can do it!

Build Times

It wasn't going to be hard to beat Word Doctor on build times. Currently, it takes about a minute to generate the full user manual, in PDF and HTML format, from source. A simple dependency check means this build is only triggered when required. The real gain here is not so much that the build is quick, but that it is automatic: not a single button needs clicking.

Conclusions

The real benefits of the new documentation toolchain are becoming more and more apparent.

2 BoostBook [12] extends DocBook to provide greater support for C++ documentation.

SOFT DOCUMENTATION

As a software user I expect software to just work – especially software with a GUI. It should be obvious what to do without needing to read the manual; and preferably without even waiting for tooltips to float into view. By designing a GUI which operates within a web browser we already have a head start here: the user interface is driven like any other web interface – there's no need to document how hyperlinks work or what the browser's address bar does.

What's more, the Manifesto for Agile Software Development explicitly prefers: *Working software over comprehensive documentation.* [13]

These considerations don't mean that the manual is redundant or unwanted, though. There are times when we don't want to clutter the core user interface with reference details. There remain occasions when hardcopy is invaluable.

What's more, when you try and design an intuitive user interface, you realise that the distinction between software and documentation is somewhat artificial: it's not so much that the boundaries blur as that, from a user's point of view, they aren't really there. Suppose, for example, that a form requires an email address to be entered. If the user enters an invalid address the form is redrawn with the erroneous input highlighted and a terse message displayed: *Please enter a valid email address*; there will also be a clickable Help icon directing confused users to the right page of the user manual. Which of these elements of the user interface are software and which are documentation?

Now suppose we are delivering a library designed to be linked into a larger program. The documentation is primarily the header files which define the interface to this library. We must invest considerable effort making sure these header files define a coherent and comprehensible interface: maybe we deliver the library with some example client code and makefiles; maybe we provide a test harness; maybe we generate hyperlinked documentation directly from the source files; and maybe we supply the library implementation as source code. Now which is software and which is documentation?

When we write software, we remember that:

Programs should be written for people to read, and only incidentally for machines to execute. [14]

In other words, software is documentation. Software should also be soft – soft enough to adapt to changing requirements. We must be sure to keep our documentation soft too.

As a simple example, a single text file defines the Product's four part version number. The build system processes this file to make sure the correct version number appears where it's needed: in the user interface, in the CD install script – and, of course, in the documentation.

Another example. If we get a support call which we think could have been avoided had the documentation been better, then we fix the documentation directly. Anyone with access to the revision control system and a text editor can make the fix. The full printed and online documentation will be regenerated when they next do a build, and will automatically be included in the next release.

And a final example. The Product I work on checks file-based digital video: it can spot unpleasant compression artifacts, unwanted black frames, audio glitches and so on. The range and depth of these checks is perhaps the area which changes most frequently: when we add support for a new video codec or container file format, for example. The architecture we have in place means that these low level enhancements disrupt the higher levels of the software only minimally: in fact, the user interface for this part of the Product is dynamically generated from a formal description of the supported checks. Adding a check at this level is a simple matter of extending this formal description. We also need to document the check: perhaps a reference to the codec specification and a precise definition of the metrics used. With an intelligent documentation toolchain the documentation can live alongside the formal description and build time checks confirm the help text links up properly.

From an engineering point of view, documentation is properly integrated into the Product. I finish with another quotation from Stayton [5]:

Setting up a DocBook system will take some time and effort. But the payoff will be an efficient, flexible, and inexpensive publishing system that can grow with your needs.

Thomas Guest
<thomas.guest@gmail.com>

References

1 Microsoft Word: http://office.microsoft.com/
2 Hibernate: http://www.hibernate.org/
3 DocBook: http://docbook.org
4 The DocBook Project: http://docbook.sourceforge.net/
5 Stayton, *DocBook XSL: The Complete Guide* http://www.sagehill.net/docbookxsl/index.html
6 Apache FOP: http://xmlgraphics.apache.org/fop/
7 Emacs: http://www.gnu.org/software/emacs/emacs.html
8 Java Advanced Imaging (JAI) API http://java.sun.com/products/java-media/jai/
9 JIMI Software Development Kit http://java.sun.com/products/jimi/
10 Boost QuickBook: http://www.boost.org/tools/quickbook/
11 Boost: http://www.boost.org/
12 BoostBook: http://www.boost.org/tools/boostbook/
13 Manifesto for Agile Software Development http://agilemanifesto.org/
14 Abelson and Sussman, *Structure and Interpretation of Computer Progams*, Harold Abelson, Gerald Jay Sussman with Julie Sussman MIT Press, 1984; ISBN 0-262-01077-1

Credits

My thanks to Alan Griffiths, Phil Bass and Alison Peck for their help with this article.

Colophon

The master version of this document was written using emacs.

Dead Code

by Tim Penhey

Dead code comes in many forms, and appears in most projects at some time or another. A general definition of dead code would be "code that is unreachable". This may be due to a function never being called, or it may be control paths within a function never being accessible. This article deals with the former.

Functions that are never called happen at two phases during the code's lifecycle: brand new code that is yet to be hooked into an existing framework; or refactoring of some code that removes calls to other functions.

Often in the refactoring of code, functions that are no longer called are left alone as they might still be called from somewhere. This is often the case on larger projects where individuals do not know the entire codebase.

Another reason functions are not deleted is the idea that "it might be useful later", an extension of the hoarder's mentality.

This leads on to an interesting question: How do you know when a function is no longer called? The problem really presents itself when looking at shared object libraries. The normal approach is that an exported function is intended to be called from some other library or executable, however, in many cases there are exported functions that are only ever called from inside the library. Just because a function is exported, does that mean it should be kept? In order to look at shared object libraries, you need also look at all the code that uses that shared object library.

Once you have identified that your code base has dead code in it, why remove it? Probably the biggest factor is to aid the programmers in their understanding. There is no point in spending time reading and understanding code that is never called. Another major factor is having less clutter and cleaner code. This leads to identifying areas for refactoring, which often leads to better abstractions. A minor benefit is speeding up the compile and link time and reducing the size of the libraries and executables.

There are tools available that will do a coverage analysis of the source code. The tool watches the code as it is being executed and will identify parts of the code that are never reached. An advantage of this is that it can also identify unreached code paths in called functions. The disadvantage is the need for either automated or manual testing. If using automated testing, then the tests need to cover all the required use cases, which in itself is hard to do much of the time due to "fluffy", incorrect, or outdated requirements. It is also often hard to "retrofit" on to a large code base. The alternative is manual testing, which means someone sitting in front of the application doing all it can do. Manual testing is probably more error prone than even limited automated testing. If the tests, manual or automated, don't fully cover the actual use cases then it is possible that required code is incorrectly identified as unused.

The impetus behind my looking into this issue was the code base at a previous contract. There was somewhere in the vicinity of two million lines of code and of those it was estimated that somewhere between 20 and 40% is no longer used anywhere. The code was built into approximately 50 shared object libraries and 20 executables. There were only limited regression tests and no user knew everything that the system was supposed to do, which led to the idea of trying to create some tool that would analyse the libraries and executables themselves.

The general approach was to process each of the shared object libraries and extract a list of exported functions to match up with the undefined functions from the shared object libraries and executables – the theory being that whatever was exported and not called was "dead code".

The tools that were chosen for the job were from GNU Binutils [1]: `nm`, `c++filt`, and `readelf`. Primarily because all the code was compiled with g++.

In order to tie `nm`, `c++filt` and `readelf` together, some glue was needed – I chose python.

GNU `nm` lists the symbols from object files. It can also extract the symbols from shared object libraries and executables. nm is capable of giving much more information than is needed for simple function usage. The parameters `--defined-only` and `--undefined-only` were used to reduce the results. These were then parsed using regular expressions to extract the mangled name.

To illustrate we have the following source for a shared object library:

```
--- start shared.hpp
#ifndef DEAD_CODE_SHARED_H
#define DEAD_CODE_SHARED_H

#include <string>

void exported_func(std::string const& param);
void unused_func(std::string const& param);

#endif
--- end shared.hpp
--- start shared.cpp
#include "shared.hpp"
#include <iostream>

void internal_func(std::string const& param)
{
  std::cout << "internal called with "
            << param << "\n";
}

void exported_func(std::string const& param)
{
  std::cout << "exported_called\n";
  internal_func(param);
}

void unused_func(std::string const& param)
{
  std::cout << "never called\n";
}
--- end shared.cpp
g++ -shared -o libshared.so shared.cpp
tim@spike:~/accu/overload/dead-code$
 nm --defined-only libshared.so
00001bd8 A __bss_start
00000740 t call_gmon_start
00001bd8 b completed.4463
00001ac4 d __CTOR_END__
00001abc d __CTOR_LIST__
00000920 t __do_global_ctors_aux
00000770 t __do_global_dtors_aux
00001bd0 d __dso_handle
00001acc d __DTOR_END__
```

```
00001ac8 d __DTOR_LIST__
00001ad4 A _DYNAMIC
00001bd8 A _edata
00001be0 A _end
00000964 T _fini
000007e0 t frame_dummy
00000ab8 r __FRAME_END__
00000900 t _GLOBAL__I__Z13internal_funcRKSs
00001bc0 a _GLOBAL_OFFSET_TABLE_
00000764 t __i686.get_pc_thunk.bx
00000700 T _init
00001ad0 d __JCR_END__
00001ad0 d __JCR_LIST__
00001bd4 d p.4462
000008a4 t __tcf_0
00000886 T _Z11unused_funcRKSs
0000085a T _Z13exported_funcRKSs
0000081c T _Z13internal_funcRKSs
000008bc t _Z41__static_initialization_and
           _destruction_0ii
00001bdc b _ZSt8__ioinit
```

The entries of interest here are the ones where the type (the bit after the hex address) is `T`. These are where the symbol is in the text (code) section.

Here is a script that extracts the defined functions and its results for `libshared.so`:

```
-- start
#!/usr/bin/env python
import re, os
exported_func = \
  re.compile('[0-9a-f]{8} T (\S+)')
exported_cmd = 'nm --defined-only %s'
for line in os.popen \
 (exported_cmd % "libshared.so").readlines():
  m = exported_func.match(line)
  if m: print m.group(1)
-- end
_fini
_init
_Z11unused_funcRKSs
_Z13exported_funcRKSs
_Z13internal_funcRKSs
-- end results
```

Mangled names are great for identifying with regular expressions and matching, but not so good for matching with the code. This is where `c++filt` comes in.

```
def unmangle(name):
    return os.popen('c++filt ' \
     + name).readline()[:-1]

-- new results
_fini
_init
unused_func(std::basic_string<char,
 std::char_traits<char>,
std::allocator<char> > const&)
exported_func(std::basic_string<char,
 std::char_traits<char>,
 std::allocator<char> > const&)
internal_func(std::basic_string<char,
 std::char_traits<char>,
 std::allocator<char> > const&)
-- end results
```

You can see with the fully expanded names why the mangled one is easier to parse and match, so both are needed. All libraries also have the `_fini` and `_init` methods, so those can be safely ignored.

In order to identify real usage you need to look at the libraries and executables together, so here is a program which uses the shared object library:

```
-- start main.cpp
#include "shared.hpp"

int main()
{
  exported_func("Hello World\n");
  return 0;
}
-- end main.cpp
-- compile & execute
tim@spike:~/accu/overload/dead-code$ g++
 -o deadtest -l shared -L . main.cpp
tim@spike:~/accu/overload/dead-code$
 ./deadtest
exported_called
internal called with Hello World

tim@spike:~/accu/overload/dead-code$
-- end
```

For the executables you are only interested in the undefined references, and of those ultimately only the ones that correspond to exported functions in the libraries.

```
tim@spike:~/accu/overload/dead-code$ nm --
undefined-only deadtest
         U __cxa_guard_acquire@@CXXABI_1.3
         U __cxa_guard_release@@CXXABI_1.3
         U getenv@@GLIBC_2.0
         w __gmon_start__
         U __gxx_personality_v0@@CXXABI_1.3
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U _Unwind_Resume@@GCC_3.0
         U _Z13exported_funcRKSs
         U _ZNSaIcEC1Ev@@GLIBCXX_3.4
         U _ZNSaIcED1Ev@@GLIBCXX_3.4
         U _ZNSsC1EPKcRKSaIcE@@GLIBCXX_3.4
         U _ZNSsD1Ev@@GLIBCXX_3.4
```

Following is a simplistic script that follows the initial approach defined above.

```
-- start nm2.py
#!/usr/bin/env python
import os, re
exported_func = re.compile \
 ('[0-9a-f]{8} T (\S+)')
unknown_func = re.compile('\s*U (\S+)')
```

```
exported_cmd = 'nm --defined-only %s'
unknown_cmd =  'nm --undefined-only %s'

ignored_funcs = \
 set(['_PROCEDURE_LINKAGE_TABLE_', '_fini',
 '_init'])

def unmangle(name):
 return os.popen('c++filt ' \
 + name).readline()[:-1]
 # return name

class Library(object):
  def __init__(self, name):
    self.fullname = name
    self.name = os.path.basename(name)
    self.exported = []
    for line in os.popen(exported_cmd \
     % self.fullname).readlines():
      m = exported_func.match(line)
      if m:
        if m.group(1) not in ignored_funcs:
          self.exported.append(m.group(1))
      self.unknown = []
      for line in os.popen(unknown_cmd \
       % self.fullname).readlines():
            m = unknown_func.match(line)
            if m:
             self.unknown.append(m.group(1))

class Binary(object):
  def __init__(self, name):
    self.fullname = name
    self.name = os.path.basename(name)
    self.unknown = []
    for line in os.popen(unknown_cmd \
     % self.fullname).readlines():
      m = unknown_func.match(line)
      if m: self.unknown.append(m.group(1))

def main():
  lib = Library('libshared.so')
  bin = Binary('deadtest')
  exported = set(lib.exported)
  for unk in bin.unknown:
    if unk in exported:
      exported.discard(unk)
  print "Unused:"
  for func in exported:
    print "\t%s" % unmangle(func)

if __name__ == "__main__":
  main()
-- end nm2.py
-- executed
tim@spike:~/accu/overload/dead-code$ ./nm2.py
Unused:
 internal_func(std::basic_string<char,
 std::char_traits<char>,
 std::allocator<char> > const&)
 unused_func(std::basic_string<char,
 std::char_traits<char>,
 std::allocator<char> > const&)
-- end executed
```

Here we can see that the function `internal_func` has been shown as not used even though it is called directly from `exported_func`. A tool that was going to give false positives like this was not going to be extremely useful.

Luckily it was pointed out to me that another GNU tool called `readelf` is able to show relocation information. There is a relocation entry for every function that is called.

The relevant lines from the results of `readelf -r --wide libshared.so` are shown in Figure 1.

More regex magic `'[0-9a-f]{8}\s+[0-9a-f]{8}\s+\S+\s+[0-9a-f]{8}\s+(\S+)'` gives a way to identify the function calls. Once these are eliminated from the exported list, we are left with only one function: `unused_func`.

Conclusion

The script ended up taking about 15 - 20 minutes to run (mainly due to an inefficiency in the c++filt calling that I never got around to fixing) but returned around about three or four thousand functions that were no longer called. The script does still show false positives though as it is not able to determine when a function is called through a pointer to a function or pointer to a member function. It did however give a good starting point to reduce the dead code.

Tim Penhey
tim@penhey.net

Thanks

Thanks to Paul Thomas on the accu-general list for pointing out `readelf` to me.

References

1 GNU: http://www.gnu.org/software/binutils/

```
00000830  00001002 R_386_PC32             00000000
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
00000850  00001002 R_386_PC32             00000000
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
0000086e  00001002 R_386_PC32             00000000
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
0000089a  00001002 R_386_PC32             00000000
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
0000083f  00000d02 R_386_PC32             00000000
_ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E
0000087c  00001502 R_386_PC32             0000081c   _Z13internal_funcRKSs
000008b3  00001d02 R_386_PC32             00000000   _ZNSt8ios_base4InitD1Ev
000008e0  00000c02 R_386_PC32             00000000   _ZNSt8ios_base4InitC1Ev
000008f7  00001102 R_386_PC32             00000000   __cxa_atexit
```

Figure 1

How to Shoot Yourself in the Foot. In an Agile Way.

by Giovanni Asproni

Introduction

The project is late and over-budget, the software is bug-ridden and unusable, the customer is furious and doesn't want to pay any more money, the team is burned out and lacks motivation. The Project Manager, looking around for advice, comes across the Agile Alliance web-site [2] and decides that an agile methodology is the way to go to rescue his project…

This is a typical scenario of introduction of an agile methodology in a company; of course it is not the only one – some projects use an agile methodology right from the start. However, no matter how and why an agile approach is chosen, there are some traps and pitfalls that it's better to be aware of.

In this article I'll describe what, in my experience, are the five most common and dangerous mistakes that can make the implementation of an agile methodology fail. I'll give also some hints about how to avoid and/or fix them.

In the rest of the article, I'll refer to the person that has the ultimate decision on what the software should do as the customer, and to the developers as the team.

Finally, the project manager is the person that, in a traditional process, is in charge of planning the activities of the team and the deliverables of the project, and, in an agile one, works more as a facilitator between the team and the customer, and makes sure that the activities of the team run as smoothly as possible. In both cases, she also keeps track of progress made and is instrumental in keeping all the stakeholders focused on the project goals.

So, You Want to be Agile

Nowadays, agile methodologies are in the mainstream, and almost everybody claims to be agile: every time I talk about agile development with some project managers or developers, the first point they frequently make is "in a certain way we are agile as well". (OK, sometimes they are not really that agile, but this is something for another article).

I personally believe that agile methods can give many projects a boost in every respect: quality, productivity, motivation, etc. However, their adoption may be more difficult than many expect. The reason is that, in every agile methodology "Individuals and interactions" are considered more important than "processes and tools" [1], and managing people it is arguably one of the most challenging activities in every organization.

There are many ways to make a project fail, but, in this article, I'll focus on what in my experience are the five most common (and dangerous, since any of them can make a project fail) mistakes that can be made in the adoption of an agile methodology:

- Mandating the methodology from above
- Lack of trust
- Measuring agility by the number of "agile practices" implemented
- Thinking that merely implementing some of the practices will improve quality
- Focusing too much on the process and not enough on the product

Let's have a look at these mistakes in more detail.

Mandating the Methodology from Above

This happens when the project manager (or some other manager) decides that the team must use an agile methodology in order to be more productive, and imposes it on the developers (and sometimes, on the customer as well).

If the project manager is lucky, and the team members already think that an agile methodology is the way to go, this approach might actually work. Unfortunately, imposition very rarely works with programmers, especially with the good ones: programmers are knowledge workers and, as such, they don't like to be patronized, especially about how to do their job properly. So trying to impose them a new methodology can actually have an effect opposite to that which was intended.

I worked in a project where Extreme Programming was imposed from above, and the developers where forced to pair program all the time (with even the pairs often chosen by the project manager), no matter what they thought about the practice, and, as a result, some of the programmers where quite grumpy during pairing sessions making them very unpleasant. Later in the project, after having seen the effects of his decision, the project manager changed his mind, and made pairing optional leaving to the programmers also the choice of whom to pair with. Suddenly something interesting happened: the programmers that used to hate pairing chose to pair most of the time; the only difference was that now they had freedom of choice and decided to choose what they thought was best for their productivity.

If you are a manager willing to adopt an agile methodology in your company and also want to succeed doing it, you should consider involving the programmers and the other stakeholders right from the start, asking for their opinion and help. You may also be willing to consider the books by Linda Rising, and Mary Lynn Manns [3], and Jim Coplien and Neil Harrison [4].

Lack of Trust

Lack of trust is always a bad thing, no matter what is the methodology (agile or traditional) used. In fact, if trust is missing, honest communication becomes very difficult and so is keeping control of the project.

There are different types of lack of trust: between the customer and the team; between the customer and the project manager; between the project manager and the team; and between team members. The symptoms are, usually, not very difficult to spot, for example, when the customer (or the project manager) doesn't trust the team, often insists in giving technical hints and tips; or, when the project manager and the customer don't trust each other, often they insist in very detailed product specifications before starting any development, to be used, by any of them, as a weapon in case of problems. Finally, lack of trust inside the team, usually, manifests itself in the form of gossip at the coffee machine, or finger pointing (and, sometimes, scapegoating, usually against a former team member) when a bug shows up.

The fact that agile methodologies are mainly based on people and their interactions makes them even more sensitive to the lack of trust than traditional ones. For this reason, several "agile practices" such as collective code ownership, face to face communication, co-location, etc., are meant also to foster trust among all the stakeholders, but, unfortunately, they are not sufficient – I've been involved in at least one Extreme Programming project where all the above trust problems where

present, even if we used all the practices suggested by the methodology.

There are no sure recipes for improving trust. However, besides some agile practices, there are some more things you can try.

First of all, everybody can contribute to creating a safe environment. This means that it should be safe for any stakeholder to express her concerns or criticism without having to fear humiliation or retaliation.

If you are a developer, a good starting point is to take responsibility for the code you write, and have the courage to be very clear and honest about what you can and cannot do without giving in to pressure. This last thing can be very difficult, especially at the beginning, but think about what happened the last time you didn't deliver what you were "forced" to promise!

If you are a project manager, a good starting point is to trust your team, give them the resources they need, share the goals with them and allow them to take responsibility for achieving them.

If you are a customer, try to understand that there is a limit to what the team can do and, if you push them hard to do more without listening to their concerns, you will have to blame yourself for the bugs in the resulting product.

Measuring Agility by the Number of "Agile Practices" Implemented

This is a very common mistake. First of all, most of the practices that are considered to be agile – e.g., configuration management, continuous integration, unit testing, collective code ownership, test driven development, etc. – are not specific to agile methodologies, but they are used in more traditional methodologies as well.

Furthermore, practices only make sense in a context, e.g., if the programmers in the team are not comfortable with pair programming, forcing them to do it could be a very big mistake.

Of course using appropriate practices is very important for the success of a project – for example, I couldn't work without having configuration management in place – but the real difference between being, or not being agile is attitude – in an agile project there is a big emphasis on people, communication, trust, and collaboration. The tools and techniques are important only as long as they add value to the work, when they don't add value any more they are discarded or adapted to the new situation.

If you want to introduce new practices in order to make development smoother and/or safer, again, it is better to look for ways to convince everybody involved (developers, customers, project manager, etc.) to buy into your idea.

Thinking that Merely Implementing Some of the Practices will Improve Quality

Unfortunately, this silver bullet view has been promoted also by several people in the agile community. In my opinion (and experience), none of the practices can automatically improve quality of the system, of the code, of the design or testing.

A case in point is Test Driven Development (TDD), which is writing the tests (usually the acceptance or unit ones) before writing any code.

Nowadays, it is often sold as the new silver bullet that will solve all of your code quality problems almost overnight.

This is a technique that, if used properly, can give excellent results in term of code quality, and productivity. However, it is just a tool, and, like any other tool, it can be misused: I worked in a project where TDD was used extensively from the beginning, and yet the code-base was of a very poor quality in terms of bugs and maintainability. The same goes for pair programming and all the other practices.

The bottom line is, good practices are welcome, but you have to use your judgement, experience, and a grain of salt before (and during) the adoption of any of them.

Focusing Too Much on the Process and not Enough on the Product

This is typical of a team using a specific process for the first time. To a certain extent it is normal: after all, when you are learning something new you need to focus on it to see if what you are doing is right or wrong.

However, when the team starts to think along the lines of "the code is not good enough, maybe we are not doing enough <put your preferred practice here>" too often, it could be a sign of something else going wrong.

Of course, good teams think about the process, and change it to fit better their current situation, but they spend only a fraction of their time doing that. In my experience, when too much time is spent on the process, it is a sign that the team is looking for a scapegoat for their shortcomings: blaming the process is easy and safe, since nobody is going to be held responsible for failure.

Conclusion

Implementing an agile methodology can give many advantages in terms of product quality, customer satisfaction, and team satisfaction as well. However, it is not an easy job: customers may fight against it because they have to be more involved and take more responsibility for the outcome; Project Managers need to learn how not to be control freaks and delegate more authority to developers; and developers have to accept more responsibility and be more accountable for what they do.

For these reasons, using a checklist based approach is not going to make the team more or less agile. Even more importantly, don't expect any practice or technique to magically improve the quality of your code-base – they are just tools that, if used wisely, may help; if not, at best won't change anything, and at worst may have disastrous consequences (especially if their use is mandated from above).

The real change happens when all the people involved are given a stake in the product and in the process as well, and they are trusted to do a good job. This is the essence of agility.

However, it is quite simple to shoot yourself in the foot by inadvertently making any of the mistakes described in this article, but can be very difficult to spot them – especially when we are the ones to blame – but, if you keep an open mind, make it safe for others to give you honest feedback, and use it to correct your mistakes, then you are likely to have a very positive impact on the implementation of your agile methodology.

Giovanni Asproni
gasproni@asprotunity.com

References

1 Agile Manifesto: http://www.agilemanifesto.org
2 Agile Alliance: http://www.agilealliance.org
3 Rising, L., Manns, M., L., *Fearless Change: patterns for introducing new ideas*, Addison Wesley, 2004
4 Coplien, J., O., Harrison, N., B., *Organizational Patterns of Agile Software Development*, Prentice Hall, 2004

Friend or Foe!

by Mark Radford

The `friend` keyword in C++ drops the barriers of access control between a class and functions and/or other classes which are named in the `friend` declaration. It is a language feature that introductory tutorial text books seem to have a lot of trouble with. In searching for an example of its use, they often reach for that of declaring freestanding operator functions as `friend`s. In this article, I want to argue the case that using `friend`s in this way is a bad design decision – albeit for a more subtle reason than any that might immediately spring to mind – and also, that `friend` is not inherently evil. I will illustrate the latter with an example of how its use does genuinely make a design more robust.

A Bad Example

First, let's dissect a simple example of using `friend`s to implement `operator<<` (note that the same arguments can be applied to many similar examples). Consider a simple (and self-explanatory) value based class:

```
class seconds
{
public:
  explicit seconds(int initialiser);
  //...
  friend std::ostream&
operator<<(std::ostream& os,
 const seconds& s);
private:
  int val;
};

std::ostream& operator<<(std::ostream& os,
 const seconds& s)
{
  os << s.val;
  return os;
}
```

The use of `friend` in this way is, in my experience, fairly common in C++ production code, probably because it is a traditional example used in C++ text books (indeed, no lesser book than *C++ Programming Language* [1] contains such an example).

The immediately obvious way to give `operator<<` access to the implementation of `seconds` is to make it a member. However, the `operator<<` is something of a language design paradox, because there is no way to define it as a class member, while at the same time allowing its idiomatic use in client code. If `operator<<` were to be defined as a member of the class `seconds`, then it would not possible to write the simple expression:

```
std::cout << seconds(5);
```

This is because in such expressions it is idiomatic for the instance of the class of which `operator<<` is a member to appear on the left hand side of the `<<` operator. If `operator<<` were made into a member of `seconds`, the expression would have to be written the other way around. Therefore, a non-member function must be used to implement this operator.

So, what's wrong with this approach? Well, one concern is that encapsulation has been breached vis-à-vis the use of `friend`, to allow a non-member function to gain access to the implementation of `seconds`. However, that is clearly not really a concern (although in my experience many seem to think it is). After all, what really is the difference between a function having (private) implementation access because it's a member or because it's a `friend`? Another way of looking at it is this: a `friend` function is a member function via a different syntax.

In the above example, the real problem is this: the use of `friend` to implement a non-member operator is only necessary because a natural (and necessary) conversion is absent from `seconds`' interface. In such class designs it makes perfect sense for instances to be convertible to a built-in type representation of their value. The addition of the `value()` member function underpins this, as follows:

```
class seconds
{
public:
  explicit seconds(int initialiser);
  //...
  int value() const;
private:
  int val;
};

std::ostream& operator<<(std::ostream& os,
 const seconds& s)
{
  os << s.value();
  return os;
}
```

Allowing the value to be converted to a built in type representation via a member function is not only a good thing, it is also necessary in order to make the class usable. For example, a class designer can not know in advance of every conversion client code will need. The provision of the `value()` member function allows any required conversion to be added without modifying the definition of `seconds`. Note the analogy with `std::string` which permits conversion to `const char*` via the `c_str()` member function. Note further, the use of a member function rather than a conversion operator, thus requiring the conversion to be a deliberate decision on the part of `seconds`' user.

Now `operator<<` can be implemented as a non-member, and there is no need to use `friend`. However, I now want to describe a short piece of design work, in which `friend` is used for the right reasons…

A Persistent Object Framework

Consider the design of a persistence framework that has the following requirements:

- The state of certain objects must transcend their existence in a C++ program. Such objects are typically those from the problem (real world) domain. When such objects are not in use in the C++ program, their state is stored in some kind of repository, let's say, a relational database.
- Such objects must be loaded into memory when they are needed, and consigned to the database when they are finished with. For

the sake of this example, the loading of objects and their consignment to the database should be transparent to the applications programmer.

The housekeeping of whether the object has been retrieved or not is delegated to a (class template) smart pointer called `persistent_ptr`. `persistent_ptr` delegates the mechanism used to retrieve objects from the database to the implementations of an interface class called `database_query`. The definitions look like this:

```
template <class persistent>
 class database_query
{
public:
  typedef  persistent  persistent_type;
  virtual persistent_type* execute() const = 0;
};

template <typename persistent>
 class persistent_ptr
{
public:
 ~persistent_ptr() {...}
// ...
  persistent* operator->() { return get(); }
  persistent const* operator->()
   const { return get(); }
private:
  persistent* get() const
  {
    if (!loaded(object)) object =
     query->execute();
    return object;
  }
  boost::scoped_ptr
   < database_query<persistent> > const query;
  persistent* object;
};
```

An illustration of `persistant_ptr`'s use looks like this:

```
void f()
{
  persistent_ptr object(…);
     :
     :
}
```

The `object` is instantiated, its state loaded when/if a call to it is made, and when `object` goes out of scope its state (if loaded) goes back into the database.

The interface class `database_query` defines the protocol for loading objects from the database into memory. It has just one member function: the `execute()` function. `persistant_ptr`'s member access operator checks if the object (of type `persistent`) is loaded and if not, calls the `database_query`'s `execute()` function, i.e. *lazy loading* is used and the object is not loaded until it is actually used. Also, the invocation of `persistant_ptr`'s destructor triggers the consigning of `persistent` to the database. So far so good. However, we are now presented with a problem: what if the `database_query`'s `execute()` function was to be called (contrary to the idea of this design) by something other than `persistent_ptr`? One option is just to document that this is not the way this framework is intended to be used. However, it would be much better if this constraint could be stated explicitly by the code itself. There is a way to do this…

A Simple Solution Using `friend`

Like many (most?) of the textbook examples of the use of `friend`, the above example featured its use in accessing private member data. However, member functions too can be made private, and one way to prevent unauthorised parties calling `database_query`'s `execute()` function is to make it private. This leaves us with a problem: `persistent_ptr` can't call it either. One simple solution is to declare `persistent_ptr` a `friend`. Given that `database_query` is an interface class and the `execute()` function is the only member function, this does not cause any problems with exposure of *all* private members – a problem normally associated with `friend`. The interface class `database_query` now looks like this:

```
template <class persistent>
 class database_query
{
public:
    typedef  persistent  persistent_type;

private:
  friend class persistent_ptr<persistent>;
  virtual persistent_type* execute() const = 0;
};
```

Note that there is no impact on derived classes because friendship is not inherited. The use of `friend` in this way has provided one simple way to bring some extra robustness to the design of this framework.

Finally

When designing for C++, occasionally there is a need for two classes to work together closely. In such cases the permitted interactions must often be defined quite specifically, to an extent not covered by the access specifiers public, protected and private. Here, the `friend` keyword can be an asset. I believe many (most?) of its traditional uses – both in textbook examples and production code – are bad, as illustrated by the `seconds` example. However, it should not be rejected as bad in every case, as I believe the example of `persistent_ptr` and its colleague `database_query` shows.

Mark Radford
mark@twonine.co.uk

References

1 Stroustrup,Bjarne (1997) *C++ Programming Language*, 3rd edition, Addison-Wesley.

Recursive Make Considered Harmful

by Peter Miller

Abstract

For large UNIX projects, the traditional method of building the project is to use recursive `make`. On some projects, this results in build times which are unacceptably large, when all you want to do is change one file. In examining the source of the overly long build times, it became evident that a number of apparently unrelated problems combine to produce the delay,but on analysis all have the same root cause.

This paper explores a number of problems regarding the use of recursive `make`, and shows that they are all symptoms of the same problem. Symptoms that the UNIX community have long accepted as a fact of life, but which need not be endured any longer. These problems include recursive `make`s which take "forever" to work out that they need to do nothing, recursive `make`s which do too much, or too little, recursive `make`s which are overly sensitive to changes in the source code and require constant `Makefile` intervention to keep them working.

The resolution of these problems can be found by looking at what `make` does, from first principles, and then analyzing the effects of introducing recursive `make` to this activity. The analysis shows that the problem stems from the artificial partitioning of the build into separate subsets. This, in turn, leads to the symptoms described. To avoid the symptoms, it is only necessary to avoid the separation; to use a single `make` session to build the whole project, which is not quite the same as a single `Makefile`.

This conclusion runs counter to much accumulated folk wisdom in building large projects on UNIX. Some of the main objections raised by this folk wisdom are examined and shown to be unfounded. The results of actual use are far more encouraging, with routine development performance improvements significantly faster than intuition may indicate, and without the intuitvely expected compromise of modularity. The use of a whole project `make` is not as difficult to put into practice as it may at first appear.

Introduction

For large UNIX software development projects, the traditional methods of building the project use what has come to be known as "recursive `make`." This refers to the use of a hierarchy of directories containing source files for the modules which make up the project, where each of the sub-directories contains a `Makefile` which describes the rules and instructions for the `make` program. The complete project build is done by arranging for the top-level `Makefile` to change directory into each of the sub-directories and recursively invoke `make`.

This paper explores some significant problems encountered when developing software projects using the recursive `make` technique. A simple solution is offered, and some of the implications of that solution are explored.

Recursive make results in a directory tree which looks something like figure 1.

This hierarchy of modules can be nested arbitrarily deep. Real-world projects often use two- and three-level structures.

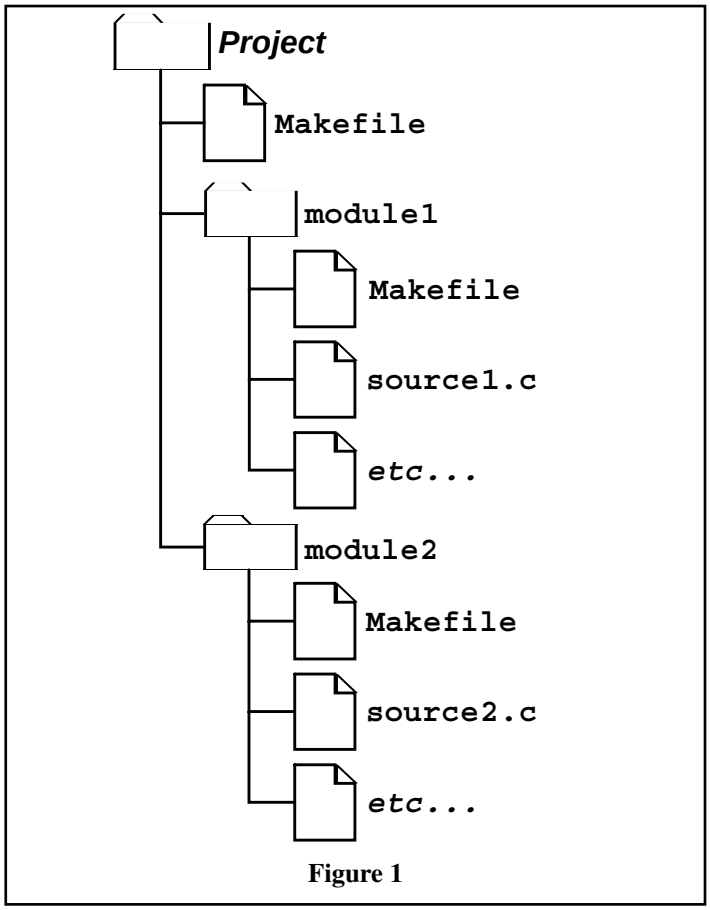

Figure 1

Assumed Knowledge

This paper assumes that the reader is familiar with developing software on UNIX, with the `make` program, and with the issues of C programming and include file dependencies.

This paper assumes that you have installed GNU Make on your system and are moderately familiar with its features. Some features of `make` described below may not be available if you are using the limited version supplied by your vendor.

The Problem

There are numerous problems with recursive `make`, and they are usually observed daily in practice. Some of these problems include:

- It is very hard to get the *order* of the recursion into the sub-directories correct. This order is very unstable and frequently needs to be manually ''tweaked.'' Increasing the number of directories, or increasing the depth in the directory tree, cause this order to be increasingly unstable.
- It is often necessary to do more than one pass over the sub-directories to build the whole system. This, naturally, leads to extended build times.
- Because the builds take so long, some dependency information is omitted, otherwise development builds take unreasonable lengths of time, and the developers are unproductive. This usually leads to things not being updated when they need to be, requiring frequent "clean" builds from scratch, to ensure everything has actually been built.
- Because inter-directory dependencies are either omitted or too hard to express, the `Makefile`s are often written to build *too much* to ensure that nothing is left out.
- The inaccuracy of the dependencies, or the simple lack of dependencies, can result in a product which is incapable of

building cleanly, requiring the build process to be carefully watched by a human.
- Related to the above, some projects are incapable of taking advantage of various "parallel make" impementations, because the build does patently silly things.

Not all projects experience all of these problems. Those that do experience the problems may do so intermittently, and dismiss the problems as unexplained "one off" quirks. This paper attempts to bring together a range of symptoms observed over long practice, and presents a systematic analysis and solution.

It must be emphasized that this paper does not suggest that `make` itself is the problem. This paper is working from the premise that `make` does *not* have a bug, that `make` does *not* have a design flaw. The problem is not in `make` at all, but rather in the input given to `make` – the way make is being used.

Analysis

Before it is possible to address these seemingly unrelated problems, it is first necessary to understand what `make` does and how it does it. It is then possible to look at the effects recursive `make` has on how `make` behaves.

Whole Project Make

`make` is an expert system. You give it a set of rules for how to construct things, and a target to be constructed. The rules can be decomposed into pair-wise ordered dependencies between files. `make` takes the rules and determines how to build the given target. Once it has determined how to construct the target, it proceeds to do so.

`make` determines how to build the target by constructing a *directed acyclic graph*, the DAG familiar to many Computer Science students. The vertices of this graph are the files in the system, the edges of this graph are the inter-file dependencies. The edges of the graph are directed because the pair-wise dependencies are ordered; resulting in an *acyclic* graph – things which look like loops are resolved by the direction of the edges.

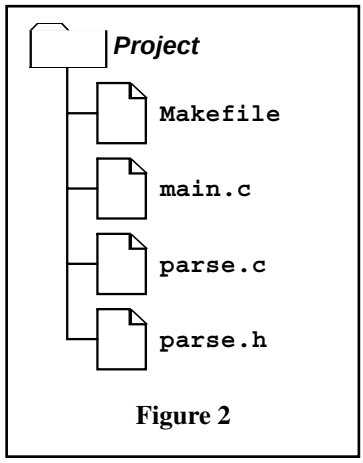

Figure 2

This paper will use a small example project for its analysis. While the number of files in this example is small, there is sufficient complexity to demonstrate all of the above recursive `make` problems. First, however, the project is presented in a non-recursive form (figure 2).

The `Makefile` in this small project looks like this:

```
OBJ = main.o parse.o
prog: $(OBJ)
  $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
  $(CC) -c main.c
parse.o: parse.c parse.h
  $(CC) -c parse.c
```

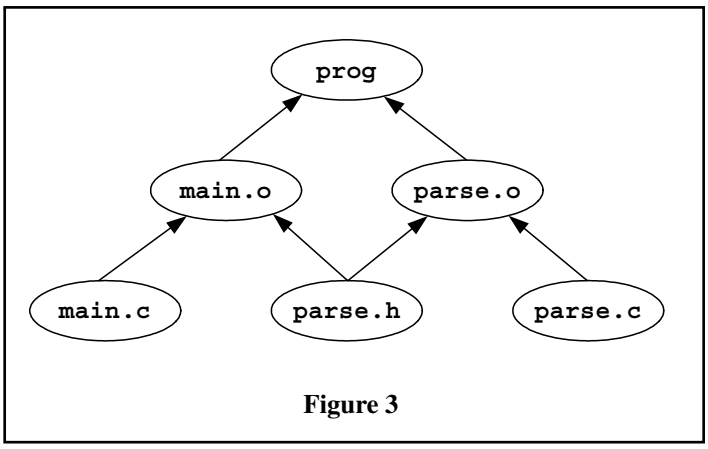

Figure 3

Some of the implicit rules of `make` are presented here explicitly, to assist the reader in converting the `Makefile` into its equivalent DAG.

The above `Makefile` can be drawn as a DAG in the form shown in figure 3.

This is an *acyclic* graph because of the arrows which express the ordering of the relationship between the files. If there *was* a circular dependency according to the arrows, it would be an error.

Note that the object files (`.o`) are dependent on the include files (`.h`) even though it is the source files (`.c`) which do the including. This is because if an include file changes, it is the object files which are out-of-date, not the source files.

The second part of what `make` does it to perform a *postorder* traversal of the DAG. That is, the dependencies are visited first. The actual order of traversal is undefined, but most `make` implementations work down the graph from left to right for edges below the same vertex, and most projects implicitly rely on this behaviour. The last-time-modified of each file is examined, and higher files are determined to be out-of-date if any of the lower files on which they depend are younger. Where a file is determined to be out-of-date, the action associated with the relevant graph edge is performed (in the above example, a compile or a link).

The use of recursive `make` affects both phases of the operation of `make`: it causes `make` to construct an inaccurate DAG, and it forces `make` to traverse the DAG in an inappropriate order.

Recursive Make

To examine the effects of recursive `make`s, the above example will be artificially segmented into two modules, each with its own `Makefile`, and a top-level `Makefile` used to invoke each of the module `Makefiles`.

This example is intentionally artificial, and thoroughly so. However, all "modularity" of all projects is artificial, to some extent. Consider: for many projects, the linker flattens it all out again, right at the end.

The directory structure is as shown in figure 4.

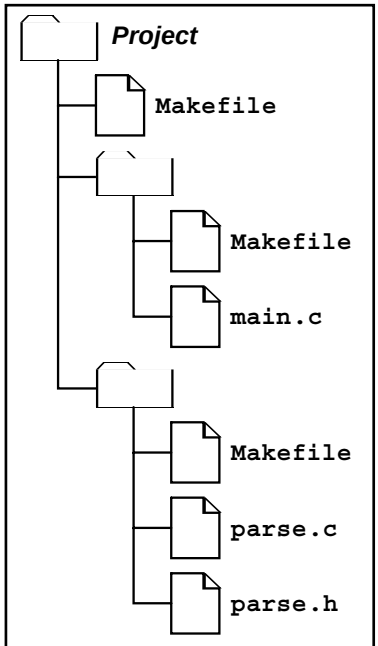

Figure 4

The top-level `Makefile` often looks a lot like a shell script:

```
MODULES = ant bee
all:
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
```

The `ant/Makefile` looks like this:

```
all: main.o

main.o: main.c ../bee/parse.h
  $(CC) -I../bee -c main.c
```

and the equivalent DAG looks like figure 5.

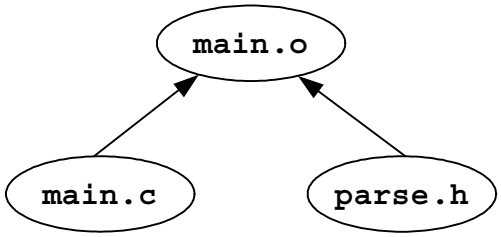

Figure 5

The `bee/Makefile` looks like this:

```
OBJ = ../ant/main.o parse.o
all: prog
prog: $(OBJ)
  $(CC) -o $@ $(OBJ)
parse.o: parse.c parse.h
  $(CC) -c parse.c
```

and the equivalent DAG looks like figure 6.

Take a close look at the DAGs. Notice how neither is complete – there are vertices and edges (files and dependencies) missing from both DAGs. When the entire build is done from the top level, everything will work.

But what happens when small changes occur? For example, what would happen if the `parse.c` and `parse.h` files were generated from a `parse.y` yacc grammar? This would add the following lines to the `bee/Makefile`:

```
parse.c parse.h: parse.y
  $(YACC) -d parse.y
  mv y.tab.c parse.c
  mv y.tab.h parse.h
```

And the equivalent DAG changes to look like figure 7.

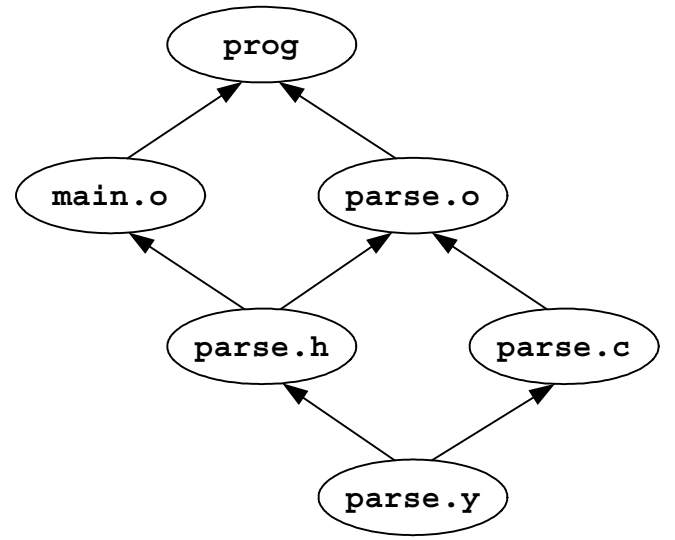

Figure 7

This change has a simple effect: if `parse.y` is edited, `main.o` will *not* be constructed correctly. This is because the DAG for `ant` knows about only some of the dependencies of `main.o`,and the DAG for `bee` knows none of them.

To understand why this happens, it is necessary to look at the actions **`make`** will take *from the top level*. Assume that the project is in a self-consistent state. Now edit `parse.y` in such a way that the generated `parse.h` file will have non-trivial differences. However, when the top-level **`make`** is invoked, first `ant` and then `bee` is visited. But `ant/main.o` is *not* recompiled, because `bee/parse.h` has not yet been regenerated and thus does not yet indicate that `main.o` is out-of-date. It is not until `bee` is visited by the recursive **`make`** that `parse.c` and `parse.h` are reconstructed, followed by `parse.o`. When the program is linked `main.o` and `parse.o` are non-trivially incompatible. That is, the program is *wrong*.

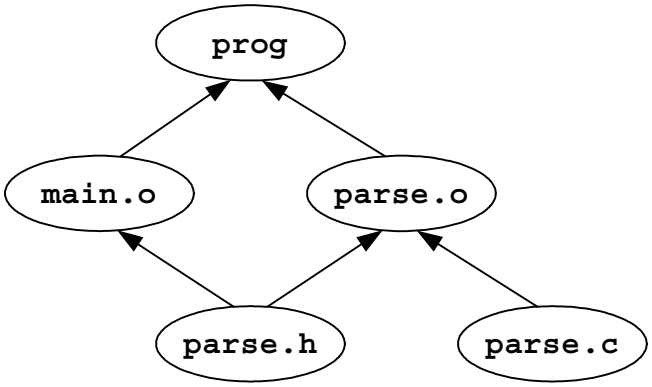

Figure 6

Traditional Solutions

There are three traditional fixes for the above "glitch."

Reshuffle

The first is to manually tweak the order of the modules in the top-level `Makefile`. But why is this tweak required at all? Isn't **`make`** supposed to be an expert system? Is **`make`** somehow flawed, or did something else go wrong?

To answer this question, it is necessary to look, not at the graphs, but the *order of traversal* of the graphs. In order to operate correctly, **`make`** needs to perform a ***postorder*** traversal, but in separating the DAG into two pieces, **`make`** has not been *allowed* to traverse the

graph in the necessary order – instead the project has dictated an order of traversal. An order which, when you consider the original graph, is plain *wrong*. Tweaking the top-level `Makefile` corrects the order to one similar to that which **`make`** could have used. Until the next dependency is added...

Note that **`make -j`** (parallel build) invalidates many of the ordering assumptions implicit in the reshuffle solution, making it useless. And then there are all of the sub-**`make`**s all doing their builds in parallel, too.

Repetition

The second traditional solution is to make more than one pass in the top-level `Makefile`, something like this:

```
MODULES = ant bee
all:
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
```

This doubles the length of time it takes to perform the build. But that is not all: there is no guarantee that two passes are enough! The upper bound of the number of passes is not even proportional to the number of modules, it is instead proportional to the number of graph edges which cross module boundaries.

Overkill

We have already seen an example of how recursive **`make`** can build too little, but another common problem is to build too much. The third traditional solution to the above glitch is to add even *more* lines to `ant/Makefile`:

```
.PHONY: ../bee/parse.h
../bee/parse.h:
    cd ../bee; \
    make clean; \
    make all
```

This means that whenever `main.o` is made, `parse.h` will always be considered to be out-of-date. All of `bee` will always be rebuilt including `parse.h`, and so `main.o` will always be rebuilt, *even if everything was self consistent*.

Note that **`make -j`** (parallel build) invalidates many of the ordering assumptions implicit in the overkill solution, making it useless, because all of the sub-**`make`**s are all doing their builds ("clean" then "all") in parallel, constantly interfering with each other in non-deterministic ways.

Prevention

The above analysis is based on one simple action: the DAG was artificially separated into incomplete pieces. This separation resulted in all of the problems familiar to recursive **`make`** builds.

Did **`make`** get it wrong? No. This is a case of the ancient GIGO principle: *Garbage In, Garbage Out*. Incomplete `Makefiles` are *wrong* `Makefiles`.

To avoid these problems, don't break the DAG into pieces; instead, use one `Makefile` for the entire project. It is not the recursion itself which is harmful, it is the crippled `Makefiles` which are used in the recursion which are *wrong*. It is not a deficiency of **`make`** itself that recursive **`make`** is broken, it does the best it can with the flawed input it is given.

> *But, but, but... You can't do that!"* I hear you cry. *"A single Makefile is too big,it'sunmaintainable,it's too hard to write the rules, you'll run out of memory, I only want to build my little bit, the build will take too long. It's just not practical.'*

These are valid concerns, and they frequently lead **`make`** users to the conclusion that re-working their build process does not have any short- or long-term benefits. This conclusion is based on ancient, enduring, false assumptions.

The following sections will address each of these concerns in turn.

A Single Makefile is Too Big

If the entire project build description were placed into a single `Makefile` this would certainly be true, however modern **`make`** implementations have include statements. By including a relevant fragment from each module, the total size of the `Makefile` and its include files need be no larger than the total size of the `Makefiles` in the recursive case.

A Single `Makefile` Is Unmaintainable

The complexity of using a single top-level `Makefile` which includes a fragment from each module is no more complex than in the recursive case. Because the DAG is not segmented, this form of `Makefile` becomes less complex, and thus *more* maintainable, simply because fewer "tweaks" are required to keep it working.

Recursive `Makefiles` have a great deal of repetition. Many projects solve this by using include files. By using a single `Makefile` for the project, the need for the "common" include files disappears – the single `Makefile` is the common part.

It's Too Hard To Write The Rules

The only change required is to include the directory part in filenames in a number of places. This is because the **`make`** is performed from the top level directory; the current directory is not the one in which the file appears. Where the output file is explicitly stated in a rule, this is not a problem.

GCC allows a **`-o`** option in conjunction with the **`-c`** option, and GNU Make knows this. This results in the implicit compilation rule placing the output in the correct place. Older and dumber C compilers, however, may not allow the **`-o`** option with the **`-c`** option, and will leave the object file in the top-level directory (i.e. the wrong directory). There are three ways for you to fix this: get GNU Make and GCC, override the built-in rule with one which does the right thing, or complain to your vendor.

Also, K&R C compilers will start the double-quote include path (**`#include "filename.h"`**) from the current directory. This will not do what you want. ANSI C compliant C compilers, however, start the double-quote include path from the directory in which the source file appears; thus, no source changes are required. If you don't have an ANSI C compliant C compiler,you should consider installing GCC on your system as soon as possible.

I Only Want To Build My Little Bit

Most of the time, developers are deep within the project tree and they edit one or two files and then run **make** to compile their changes and try them out. They may do this dozens or hundreds of times a day. Being forced to do a full project build every time would be absurd.

Developers always have the option of giving **make** a specific target. This is always the case, it's just that we usually rely on the default target in the `Makefile` in the current directory to shorten the command line for us. Building "my little bit" can still be done with a whole project `Makefile`, simply by using a specific target, and an alias if the command line is too long.

Is doing a full project build every time so absurd? If a change made in a module has repercussions in other modules, because there is a dependency the developer is unaware of (but the `Makefile` is aware of), isn't it better that the developer find out as early as possible? Dependencies like this *will* be found, because the DAG is more complete than in the recursive case.

The developer is rarely a seasoned old salt who knows every one of the million lines of code in the product. More likely the developer is a short-term contractor or a junior. You don't want implications like these to blow up after the changes are integrated with the master source, you want them to blow up on the developer in some nice safe sand-box far awayfrom the master source.

If you want to make "just your little" bit because you are concerned that performing a full project build will corrupt the project master source, due to the directory structure used in your project, see the "Projects *versus* Sand-Boxes" section below.

The Build Will Take Too Long

This statement can be made from one of two perspectives. First, that a whole project **make**, even when everything is up-to-date, inevitably takes a long time to perform. Secondly, that these inevitable delays are unacceptable when a developer wants to quickly compile and link the one file that they have changed.

Project Builds

Consider a hypothetical project with 1000 source (`.c`) files, each of which has its calling interface defined in a corresponding include (`.h`) file with defines, type declarations and function prototypes. These 1000 source files include their own interface definition, plus the interface definitions of any other module they may call. These 1000 source files are compiled into 1000 object files which are then linked into an executable program. This system has some 3000 files which **make** must be told about, and be told about the include dependencies, and also explore the possibility that implicit rules (`.y` ➔ `.c` for example) may be necessary.

In order to build the DAG, **make** must "stat" 3000 files, plus an additional 2000 files or so, depending on which implicit rules your **make** knows about and your `Makefile` has left enabled. On the author's humble 66MHz i486 this takes about 10 seconds; on native disk on faster platforms it goes even faster. With NFS over 10MB Ethernet it takes about 10 seconds, no matter what the platform.

This is an astonishing statistic! Imagine being able to do a single file compile, out of 1000 source files, in only 10 seconds, plus the time for the compilation itself.

Breaking the set of files up into 100 modules, and running it as a recursive **make** takes about 25 seconds. The repeated process creation for the subordinate **make** invocations take quite a long time.

Hang on a minute! On real-world projects with less than 1000 files, it takes an awful lot longer than 25 seconds for **make** to work out that it has nothing to do. For some projects, doing it in only 25 minutes would be an improvement! The above result tells us that it is not the number of files which is slowing us down (that only takes 10 seconds), and it is not the repeated process creation for the subordinate **make** invocations (that only takes another 15 seconds). So just what *is* taking so long?

The traditional solutions to the problems introduced by recursive **make** often increase the number of subordinate **make** invocations beyond the minimum described here; e.g. to perform multiple repetitions (see 'Repetition', above), or to overkill cross-module dependencies (see 'Overkill', above). These can take a long time, particularly when combined, but do not account for some of the more spectacular build times; what else is taking so long?

Complexity of the `Makefile` is what is taking so long. This is covered, below, in the 'Efficient Makefiles' section.

Development Builds

If, as in the 1000 file example, it only takes 10 seconds to figure out which one of the files needs to be recompiled, there is no serious threat to the productivity of developers if they do a whole project **make** as opposed to a module-specific **make**. The advantage for the project is that the module-centric developer is reminded at relevant times (and only relevant times) that their work has wider ramifications.

By consistently using C include files which contain accurate interface definitions (including function prototypes), this will produce compilation errors in many of the cases which would result in a defective product. By doing whole-project builds, developers discover such errors very early in the development process, and can fix the problems when they are least expensive.

You'll Run Out Of Memory

This is the most interesting response. Once long ago, on a CPU far, far away, it may even have been true. When Feldman [1] first wrote **make** it was 1978 and he was using a PDP11. Unix processes were limited to 64KB of data.

On such a computer, the above project with its 3000 files detailed in the whole-project `Makefile`, would probably *not* allow the DAG and rule actions to fit in memory.

But we are not using PDP11s any more. The physical memory of modern computers exceeds 10MB for *small* computers, and virtual memory often exceeds 100MB. It is going to take a project with hundreds of thousands of source files to exhaust virtual memory on a *small* modern computer. As the 1000 source file example takes less than 100KB of memory (try it, I did) it is unlikely that any project manageable in a single directory tree on a single disk will exhaust your computer's memory.

Why Not Fix The DAG InThe Modules?

It was shown in the above discussion that the problem with recursive **make** is that the DAGs are incomplete. It follows that by adding the missing portions, the problems would be resolved without abandoning the existing recursive **make** investment.

- The developer needs to remember to do this. The problems will not affect the developer of the module, it will affect the developers of *other* modules. There is no trigger to remind the developer to do this, other than the ire of fellow developers.

- It is difficult to work out where the changes need to be made. Potentially every `Makefile` in the entire project needs to be examined for possible modifications. Of course, you can wait for your fellow developers to find them for you.
- The include dependencies will be recomputed unnecessarily, or will be interpreted incorrectly. This is because **`make`** is string based, and thus "`.`"and "`../ant`" are two different places, even when you are in the `ant` directory. This is of concern when include dependencies are automatically generated – as they are for all large projects.

By making sure that each `Makefile` is complete, you arrive at the point where the `Makefile` for at least one module contains the equivalent of a whole-project `Makefile` (recall that these modules form a single project and are thus inter-connected), and there is no need for the recursion anymore.

Efficient Makefiles

The central theme of this paper is the *semantic* side-effects of artificially separating a `Makefile` into the pieces necessary to perform a recursive **`make`**. However, once you have a large number of `Makefiles`, the speed at which **`make`** can interpret this multitude of files also becomes an issue.

Builds can take "forever" for both these reasons: the traditional fixes for the separated DAG may be building too much *and* your `Makefile` may be inefficient.

Deferred Evaluation

The text in a `Makefile` must somehow be read from a text file and understood by **`make`** so that the DAG can be constructed, and the specified actions attached to the edges. This is all kept in memory.

The input language for `Makefiles` is deceptively simple. A crucial distinction that often escapes both novices and experts alike is that **`make`**'s input language is *text based*, as opposed to token based, as is the case for C or AWK. **`make`** does the very least possible to process input lines and stash them away in memory.

As an example of this, consider the following assignment:

```
OBJ = main.o parse.o
```

Humans read this as the variable OBJ being assigned two filenames `main.o` and `parse.o`. But **`make`** does not see it that way. Instead OBJ is assigned the *string* **`"main.o parse.o"`**. It gets worse:

```
SRC = main.c parse.c
OBJ = $(SRC:.c=.o)
```

In this case humans expect **`make`** to assign two filenames to OBJ, but **`make`** actually assigns the string **`''$(SRC:.c=.o)''`**. This is because it is a *macro* language with deferred evaluation, as opposed to one with variables and immediate evaluation.

If this does not seem too problematic, consider the following Makefile shown at the top of the next column.

How many times will the shell command be executed? **Ouch!** It will be executed *twice* just to construct the DAG,and a further *two* times if the rule needs to be executed.

If this shell command does anything complexor time consuming (and it usually does) it will take *four* times longer than you thought.

```
SRC = $(shell echo 'Ouch!' \
  1>&2 ; echo *.[cy])
OBJ = \
  $(patsubst %.c,%.o,\
    $(filter %.c,$(SRC))) \
  $(patsubst %.y,%.o,\
    $(filter %.y,$(SRC)))
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
```

But it is worth looking at the other portions of that OBJ macro. Each time it is named, a huge amount of processing is performed:

- The argument to **`shell`** is a single string (all built-in-functions take a single string argument). The string is executed in a sub-shell, and the standard output of this command is read back in, translating new lines into spaces. The result is a single string.
- The argument to **`filter`** is a single string. This argument is broken into two strings at the first comma. These two strings are then each broken into sub-strings separated by spaces. The first set are the patterns, the second set are the filenames. Then, for each of the pattern substrings, if a filename sub-string matches it, that filename is included in the output. Once all of the output has been found, it is re-assembled into a single space-separated string.
- The argument to **`patsubst`** is a single string. This argument is broken into three strings at the first and second commas. The third string is then broken into sub-strings separated by spaces, these are the filenames. Then, for each of the filenames which match the first string it is substituted according to the second string. If a filename does not match, it is passed through unchanged. Once all of the output has been generated, it is re-assembled into a single space-separated string.

Notice how many times those strings are disassembled and re-assembled. Notice how many ways that happens. *This is slow*. The example here names just two files but consider how inefficient this would be for 1000 files. Doing it *four* times becomes decidedly inefficient.

If you are using a dumb **`make`** that has no substitutions and no built-in functions, this cannot bite you. But a modern **`make`** has lots of built-in functions and can even invoke shell commands on-the-fly.The semantics of **`make`**'s text manipulation is such that string manipulation in **`make`** is very CPU intensive,compared to performing the same string manipulations in C or AWK.

Immediate Evaluation

Modern **`make`** implementations have an immediate evaluation `:=` assignment operator.The above example can be re-written as

```
SRC := $(shell echo 'Ouch!' \
  1>&2 ; echo *.[cy])
OBJ := \
  $(patsubst %.c,%.o,\
    $(filter %.c,$(SRC))) \
  $(patsubst %.y,%.o,\
    $(filter %.y,$(SRC)))
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
```

Note that *both* assignments are immediate evaluation assignments. If the first were not, the shell command would always be executed twice. If the second were not, the expensive substitutions would be performed at least twice and possibly four times.

As a rule of thumb: always use immediate evaluation assignment unless you knowingly want deferred evaluation.

Include Files

Many `Makefiles` perform the same text processing (the filters above,for example) for every single **`make`** run, but the results of the processing rarely change. Wherever practical, it is more efficient to record the results of the text processing into a file, and have the `Makefile` include this file.

Dependencies

Don't be miserly with include files. They are relatively inexpensive to read, compared to `$(shell)`, so more rather than less doesn't greatly affect efficiency.

As an example of this, it is first necessary to describe a useful feature of GNU Make: once a `Makefile` has been read in, if any of its included files were out-of-date (or do not yet exist), they are re-built, and then **`make`** starts again, which has the result that **`make`** is now working with up-to-date include files. This feature can be exploited to obtain automatic include file dependency tracking for C sources. The obvious way to implement it, however, has a subtle flaw.

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include dependencies
dependencies: $(SRC)
  depend.sh $(CFLAGS) \
    $(SRC) > $@
```

The `depend.sh` script prints lines of the form:

```
file.o: file.cinclude.h...
```

The most simple implementation of this is to use GCC, but you will need an equivalent awk script or C program if you have a

```
#!/bin/sh
gcc -MM -MG "$@"
```

different compiler:

This implementation of tracking C include dependencies has several serious flaws, but the one most commonly discovered is that the `dependencies` file does not, itself, depend on the C include files. That is, it is not re-built if one of the include files changes. There is no edge in the DAG joining the `dependencies` vertex to any of the include file vertices. If an include file changes to include another file (a nested include), the dependencies will not be recalculated, and potentially the C file will not be recompiled, and thus the program will not be re-built correctly.

A classic build-too-little problem, caused by giving **`make`** inadequate information, and thus causing it to build an inadequate DAG and reach the wrong conclusion.

The traditional solution is to build too much:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include dependencies
.PHONY: dependencies
dependencies: $(SRC)
  depend.sh $(CFLAGS) \
    $(SRC) > $@
```

Now, even if the project is completely up-do-date, the dependencies will be re-built. For a large project, this is very wasteful, and can be a major contributor to **`make`** taking "forever" to work out that nothing needs to be done.

There is a second problem, and that is that if any *one* of the C files changes, *all* of the C files will be re-scanned for include dependencies. This is as inefficient as having a `Makefile` which reads

```
prog: $(SRC)
  $(CC) -o $@ $(SRC)
```

What is needed, in exact analogy to the C case, is to have an intermediate form. This is usually given a `.d` suffix. By exploiting the fact that more than one file may be named in an include line, there is no need to "link"all of the `.d` files together:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include $(OBJ:.o=.d)
%.d: %.c
  depend.sh $(CFLAGS) $* > $@
```

This has one more thing to fix: just as the object (`.o`) files depend on the source files and the include files, so do the dependency (`.d`) files.

`file.d file.o: file.c include.h`

This means tinkering with the `depend.sh` script again:

```
#!/bin/sh
gcc -MM -MG "$@" |
sed -e 's@^\(.*\)\.o:@\1.d \1.o:@'
```

This method of determining include file dependencies results in the `Makefile` including more files than the original method, but opening files is less expensive than rebuilding all of the dependencies every time. Typically a developer will edit one or two files before re-building; this method will rebuild the *exact* dependency file affected (or more than one, if you edited an include file). On balance, this will use less CPU, and less time.

In the case of a build where nothing needs to be done, **`make`** will actually do nothing, and will work this out very quickly.

However, the above technique assumes your project fits enitrely within the one directory. For large projects, this usually isn't the case.

This means tinkering with the `depend.sh` script again:

```
#!/bin/sh
DIR="$1"
shift 1
case "$DIR" in
"" | ".")
gcc -MM -MG "$@" |
sed -e 's@^\(.*\)\.o:@\1.d \1.o:@'
;;
*)
gcc -MM -MG "$@" |
sed -e "s@^\(.*\)\.o:@$DIR/\1.d $DIR/\1.o:@"
;;
esac
```

And the rule needs to change, too, to pass the directory as the first argument, as the script expects.

```
%.d: %.c
  depend.sh `dirname $*` $(CFLAGS) $* > $@
```

Note that the `.d` files will be relative to the top level directory. Writing them so that they can be used from any level is possible, but beyond the scope of this paper.

Multiplier

All of the inefficiencies described in this section compound together. If you do 100 `Makefile` interpretations, once for each module, checking 1000 source files can take a very long time - if the interpretation requires complex processing or performs unnecessary work, or both. A whole project **`make`**, on the other hand, only needs to interpret a single `Makefile`.

Projects versus Sand-boxes

The above discussion assumes that a project resides under a single directory tree, and this is often the ideal. However, the realities of working with large software projects often lead to weird and wonderful directory structures in order to have developers working on different sections of the project without taking complete copies and thereby wasting precious disk space.

It is possible to see the whole-project **`make`** proposed here as impractical, because it does not match the evolved methods of your development process.

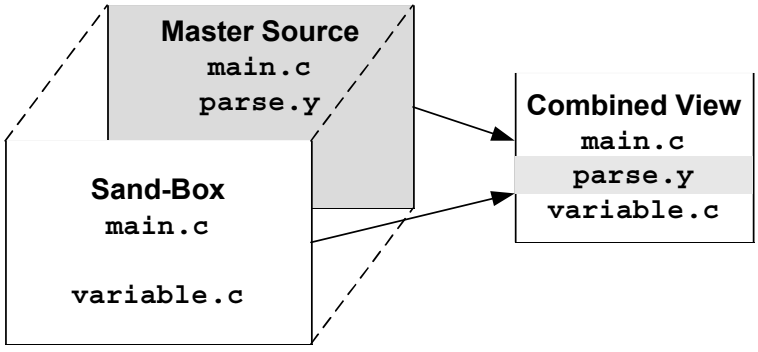

Figure 8

The whole-project **`make`** proposed here does have an effect on development methods: it can give you cleaner and simpler build environments for your developers. By using **`make`**'s VPATH feature, it is possible to copy only those files you need to edit into your private work area, often called a *sandbox*.

The simplest explanation of what VPATH does is to make an analogy with the include file search path specified using `-I path` options to the C compiler. This set of options describes where to look for files, just as VPATH tells **`make`** where to look for files.

By using VPATH, it is possible to "stack" the sand-box *on top of* the project master source, so that files in the sand-box take precedence, but it is the union of all the files which **`make`** uses to perform the build (see Figure 8).

In this environment, the sand-box has the same tree structure as the project master source. This allows developers to safely change things across separate modules, e.g. if they are changing a module interface. It also allows the sand-box to be physically separate – perhaps on a different disk, or under their home directory. It also allows the project master source to be read-only, if you have (or would like) a rigorous check-in procedure.

Note: in addition to adding a **`VPATH`** line to your development `Makefile`, you will also need to add `-I` options to the **`CFLAGS`** macro, so that the C compiler uses the same path as **`make`** does. This is simply done with a 3-line `Makefile` in your work area – set a macro, set the VPATH, and then include the `Makefile` from the project master source.

VPATH Semantics

For the above discussion to apply, you need to use GNU Make 3.76 or later. For versions of GNU Make earlier than 3.76, you will need Paul Smith's VPATH+ patch. This may be obtained from `ftp://ftp.wellfleet.com/netman/psmith/gmake/`.

The POSIX semantics of `VPATH` are slightly brain-dead, so many other **`make`** implementations are too limited. You may want to consider installing GNU Make.

The Big Picture

This section brings together all of the preceding discussion, and presents the example project with its separate modules, but with a whole-project `Makefile`. The directory structure is changed little from the recursive case, except that the deeper `Makefiles` are replaced by module specific include files (see Figure 9).

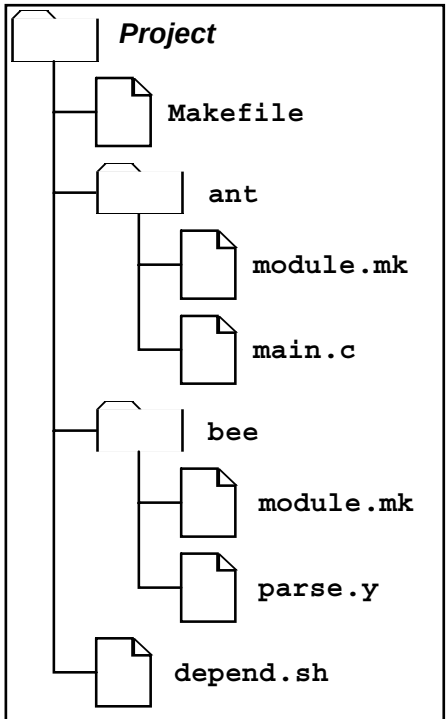

Figure 9

The `Makefile` looks like this:

```
MODULES := ant bee
#look for include files in
# each of the modules
CFLAGS += $(patsubst %,-I%,\
  $(MODULES))

#extra libraries if required
LIBS :=

#each module will add to this
SRC :=

#include the description for
# each module
include $(patsubst %,\
    %/module.mk,$(MODULES))

#determine the object files
OBJ := \
  $(patsubst %.c,%.o, \
    $(filter %.c,$(SRC))) \
  $(patsubst %.y,%.o, \
    $(filter %.y,$(SRC)))

#link the program
prog: $(OBJ)
  $(CC) -o $@ $(OBJ) $(LIBS)

#include the C include
# dependencies
include $(OBJ:.o=.d)

#calculate C include
# dependencies
%.d: %.c
  depend.sh `dirname $*.c` $(CFLAGS) $*.c > $@
```

This looks absurdly large, but it has all of the common elements in the one place, so that each of the modules' **make** includes may be small.

The `ant/module.mkfile` looks like:

```
SRC += ant/main.c
```

The `bee/module.mkfile` looks like:

```
SRC += bee/parse.y
LIBS += -ly
%.c %.h: %.y
  $(YACC) -d $*.y
  mv y.tab.c $*.c
  mv y.tab.h $*.h
```

Notice that the built-in rules are used for the C files, but we need special yacc processing to get the generated `.h` file.

The savings in this example look irrelevant, because the top-level `Makefile` is so large. But consider if there were 100 modules, each with only a few non-comment lines, and those specifically relevant to the module. The savings soon add up to a total size often *less than* the recursive case, without loss of modularity.

The equivalent DAG of the `Makefile` after all of the includes looks like figure 10

The vertexes and edges for the include file dependency files are also present as these are important for make to function correctly.

Side Effects

There are a couple of desirable side-effects of using a single `Makefile`.

- The GNU Make `-j` option, for parallel builds, works even better than before. It can find even more unrelated things to do at once, and no longer has some subtle problems.
- The general **`make -k`** option, to continue as far as possible even in the face of errors, works even better than before. It can find even more things to continue with.

LiteratureSurvey

Howcan it be possible that we have been mis-using **make** for 20 years? How can it be possible that behaviour previously ascribed to **make**'s limitations is in fact a result of mis-using it?

The author only started thinking about the ideas presented in this paper when faced with a number of ugly build problems on utterly different projects, but with common symptoms. By stepping back from the individual projects, and closely examining the thing they had in common, **make**, it became possible to see the larger pattern. Most of us are too caught up in the minutiae of just getting the rotten build to work that we don't have time to spare for the big picture. Especially when the item in question "obviously" works, and has done so continuously for the last 20 years.

It is interesting that the problems of recursive **make** are rarely mentioned in the very books Unix programmers rely on for accurate, practical advice.

The Original Paper

The original **make** paper [1] contains no reference to recursive **make**, let alone any discussion as to the relative merits of whole project **make** over recursive **make**.

It is hardly surprising that the original paper did not discuss recursive **make**, Unix projects at the time usually did fit into a single directory.

It may be this which set the "one `Makefile` in every directory" concept so firmly in the collective Unix development mind-set.

GNU Make

The GNU Make manual [2] contains several pages of material concerning recursive `make`, however its discussion of the merits or otherwise of the technique are limited to the brief statement that

> *This technique is useful when you want to separate makefiles for various subsystems that compose a larger system.*

No mention is made of the problems you may encounter.

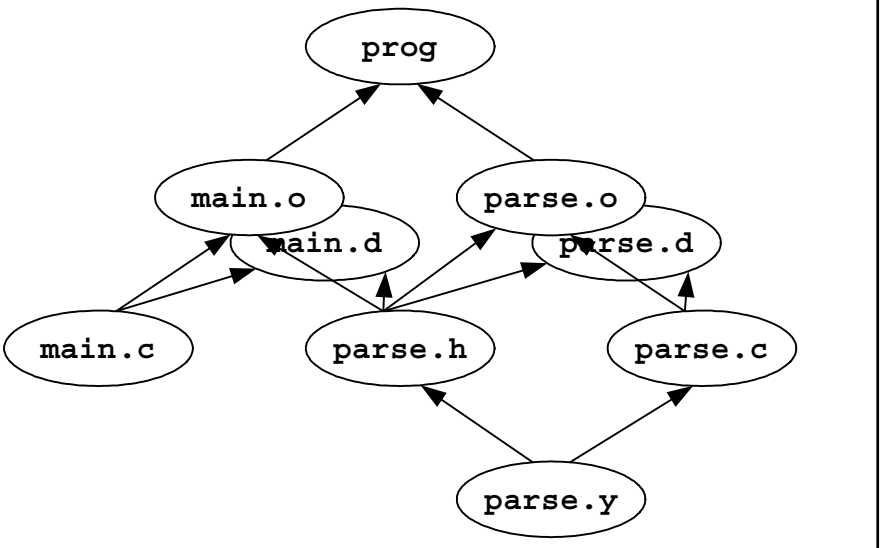

Figure 10

Managing Projects with Make

The Nutshell Makebook [3] specifically promotes recursive **make** overwhole project **make** because:

> *The cleanest way to build is to put a separate description file in each directory, and tie them together through a master description file that invokes make recursively. While cumbersome, the technique is easier to maintain than a single, enormous file that covers multiple directories.* (p. 65)

This is despite the book's advice only two paragraphs earlier that

> *make is happiest when you keep all your files in a single directory.* (p. 64)

Yet the book fails to discuss the contradiction in these two statements, and goes on to describe one of the traditional ways of treating the symptoms of incomplete DAGs caused by recursive **make**.

The book may give us a clue as to why recursive **make** has been used in this way for so many years. Notice how the above quotes confuse the concept of a directory with the concept of a `Makefile`.

This paper suggests a simple change to the mind-set: directory trees, however deep, are places to store files; `Makefiles` are places to describe the relationships between those files, however many.

BSD Make

The tutorial for BSD Make [4] says nothing at all about recursive **make**, but it is one of the few which actually described, however briefly, the relationship between a `Makefile` and a DAG (p. 30). There is also a wonderful quote

> *If make doesn't do what you expect it to, it's a good chance the makefile is wrong.* (p. 10)

Which is a pithy summary of the thesis of this paper.

Summary

This paper presents a number of related problems, and demonstrates that they are not inherent limitations of **make**, as is commonly believed, but are the result of presenting incorrect information to **make**. This is the ancient *Garbage In, Garbage Out* principle at work. Because **make** can only operate correctly with a complete DAG, the error is in segmenting the `Makefile` into incomplete pieces.

This requires a shift in thinking: directory trees are simply a place to hold files, `Makefiles` are a place to remember relationships between files. Do not confuse the two because it is as important to accurately represent the relationships between files in different directories as it is to represent the relationships between files in the same directory. This has the implication that there should be exactly one `Makefile` for a project, but the magnitude of the description can be managed by using a **make** include file in each directory to describe the subset of the project files in that directory. This is just as modular as having a `Makefile` in each directory.

This paper has shown how a project build and a development build can be equally brief for a whole-project **make**. Given this parity of time, the gains provided by accurate dependencies mean that this process will, in fact, be faster than the recursive **make** case, and more accurate.

Inter-dependent Projects

In organizations with a strong culture of re-use, implementing whole-project **make** can present challenges. Rising to these challenges, however, may require looking at the bigger picture.

- A module may be shared between two programs because the programs are closely related. Clearly, the two programs plus the shared module belong to the same project (the module may be self-contained, but the programs are not). The dependencies must be explicitly stated, and changes to the module must result in both programs being recompiled and re-linked as appropriate. Combining them all into a single project means that whole-project **make** can accomplish this.
- A module may be shared between two projects because they must inter-operate. Possibly your project is bigger than your current directory structure implies. The dependencies must be explicitly stated, and changes to the module must result in both projects being recompiled and re-linked as appropriate. Combining them all into a single project means that whole-project **make** can accomplish this.
- It is the normal case to omit the edges between your project and the operating system or other installed third party tools. So normal that they are ignored in the `Makefiles` in this paper, and they are ignored in the built-in rules of **make** programs.

Modules shared between your projects may fall into a similar category: if they change, you will deliberately re-build to include their changes, or quietly include their changes

whenever the next build may happen. In either case, you do not explicitly state the dependencies, and whole-project `make` does not apply.

- Re-use may be better served if the module were used as a template, and divergence between two projects is seen as normal. Duplicating the module in each project allows the dependencies to be explicitly stated, but requires additional effort if maintenance is required to the common portion.

How to structure dependencies in a strong re-use environment thus becomes an exercise in *risk management*. What is the danger that omitting chunks of the DAG will harm your projects? How vital is it to rebuild if a module changes? What are the consequences of *not* rebuilding automatically? How can you tell when a rebuild is necessary if the dependencies are not explicitly stated? What are the consequences of forgetting to rebuild?

Return On Investment

Some of the techniques presented in this paper will improve the speed of your builds, even if you continue to use recursive `make`. These are not the focus of this paper, merely a useful detour.

The focus of this paper is that you will get more accurate builds of your project if you use whole-project `make` rather than recursive `make`.

- The time for `make` to work out that nothing needs to be done will not be more, and will often be less.
- The size and complexity of the total `Makefile` input will not be more, and will often be less.
- The total `Makefile` input is no less modular than in the recursive case.
- The difficulty of maintaining the total `Makefile` input will not be more, and will often be less.

The disadvantages of using whole-project `make` over recursive `make` are often un-measured. How much time is spent figuring out why `make` did something unexpected? How much time is spent figuring out that `make` *did* something unexpected? How much time is spent tinkering with the build process? These activities are often thought of as "normal" development overheads.

Building your project is a fundamental activity. If it is performing poorly, so are development, debugging and testing. Building your project needs to be so simple the newest recruit can do it immediately with only a single page of instructions. Building your project needs to be so simple that it rarely needs any development effort at all. Is your build process this simple?

Peter Miller
miller@canb.auug.org.au

References

1 Feldman, Stuart I. (1978). Make- APro gramfor Maintaining Computer Programs. Bell Laboratories Computing Science Technical Report 57

2 Stallman, Richard M. and Roland McGrath (1993). GNU Make: A Programfor Directing Recompilation.Free Software Foundation, Inc.

3 Talbott, Steve (1991). Managing Projects with Make, 2nd Ed.O'Reilly & Associates, Inc.

4 de Boor, Adam (1988). PMake- ATutorial.University of California, Berkeley

About the Author

Peter Miller has worked for many years in the software R&D industry, principally on UNIX systems. In that time he has written tools such as Aegis (a software configuration management system) and Cook (yet another make-oid), both of which are freely available on the Internet. Supporting the use of these tools at many Internet sites provided the insights which led to this paper.

Please visit `http://www.canb.auug.org.au/~millerp/` if you would like to look at some of the author's free software.